UNIVERSITY OF OSLO
Department of Informatics

Extending DataPool: A Tool for Handling Input Data in Scientific Computing, Using Python Web Frameworks

Master thesis

Fredrik B. Fjeld

August 2009

Extending DataPool: A Tool for Handling Input Data in Scientific Computing, Using Python Web Frameworks

Fredrik B. Fjeld

August 2009

Abstract

Simulation programs frequently need a large amount of input data. Code dealing with reading input data and initializing data structures can be very tedious to write, especially if the data are to be defined in graphical interfaces. When using general-purpose frameworks, the idea is that the application code should be short, but this is often difficult because managing input may still require a huge effort. Hence, a tool for managing input data is needed such that the application programmer can quickly put together pieces of code that handle all aspects of supplying data to the program. A tool was implemented to address these challenges, and the result is a package called DataPool, which can greatly simplify the creation of user interfaces in Python programs. DataPool is a configurable Python package and tool for managing and controlling input data in simulation programs. DataPool can only handle input for a set of *parameters*. It cannot be used to create interactive drawings, advanced widgets, or fancy layout of a GUI. Nonetheless, DataPool may use these elements to let the user adjust a large number of physical and numerical parameters by offering a fancy layout and interactivity in simulation programs where it is necessary. This is achieved by using the available interfaces to DataPool, as well as developing new interfaces.

The motivation for this master thesis was to find a way of increasing the efficiency and usability when performing computational simulations, by extending and equipping the GUI module of the package DataPool with a highly visual, easy-to-use and powerful user interface. The angling of a possible solution was set on realizing this with the use of Python web frameworks. The thesis investigates the feasibility of obtaining a satisfying solution abiding loose coupling, extendability and being generic in a framework-based implementation. The starting point of evaluated Python web frameworks is Django. Other framework for Python like TurboGears was also up for evaluation and compared and reflected with the former. The thesis makes an in-depth investigation and evaluation of the Django framework. The result is the user interface DataPool Web. A web-based menu system designed to present the internal tree structure defined through the DataPool package in the most usable and effective way. It focuses on user interaction, practical functions and visual communication in order to make the use of DataPool package easy and time saving. The interface has the ability to present large amounts of data in an effective and lucid manner for the user.

Keywords: computational science, scientific computing, Django, Python, TurboGears, simulation, visualization, web, user interface

Acknowledgments

First, I would like to express my gratitude to my supervisor, professor Hans Petter Langtangen at Simula Research Laboratory. Thank you for showing me the world of Python, and giving me the opportunity of pursuing this passion even further with this master thesis. Your patience, guidance, contributions and inspiring discussions during this time were priceless.

I would like to thank Rustam Mehmandarov for the valuable team-work proving the concept of extreme programming, the interesting and fun discussions and the never-ending sessions of coding and xkcd humor.

I also wish to express my gratitude to everybody at Simula Research Laboratory for a fantastic environment, both academically and socially. I also thank everybody in ProsIT for making my studies a very active, fun and memorable time.

A special thank to Christian Mikalsen for the support in my startup company during this period. The Grunderskolen stay in Shanghai would never been realized without your help.

I thank my friends for the understanding, encouragement and standing by me during this period. You know who you are.

Finally, I thank my family for endless support and care, and for giving me free wings.

Thank you.

Oslo, Norway
August 2009
Fredrik B. Fjeld

Contents

1 Introduction **11**

1.1 Motivation . . . 11
1.2 Contribution . . . 12
1.3 Problem Statement . . . 13
1.4 Specifying the thesis . . . 14
1.5 Collaboration and overlap . . . 14
1.6 Outline . . . 15
1.7 Conventions in the thesis . . . 15

2 Using DataPool: User Manual **17**

2.1 Introduction . . . 17
2.2 Basic Ideas of DataPool Usage . . . 17
2.2.1 The Three Key Steps . . . 18
2.2.2 Working with Submenus . . . 18
2.2.3 Specifying Data Items . . . 19
2.2.4 Starting the User Interface . . . 22
2.2.5 Extracting Parameter Values . . . 23
2.2.6 Specifying and Getting Multiple Parameters . . . 23
2.3 An Introductory Worked Example . . . 24
2.3.1 The Basic Program . . . 24
2.3.2 Adding DataPool Functionality . . . 25
2.3.3 DataPool Command-Line Options . . . 29
2.3.4 Operating the Command Line Interface . . . 30
2.3.5 Operating the Interactive Command Interface . . . 31
2.3.6 Operating the File Interface . . . 32
2.3.7 Unit Conversion . . . 32
2.3.8 Writing the Menu Tree to File . . . 32
2.3.9 Automatic Generation of Documentation . . . 34
2.3.10 More Advanced Specification of Data Items . . . 35
2.3.11 Traversing Menu Tree . . . 38
2.3.12 Add a Menu with Minimally Intrusive Approaches . . . 38
2.3.13 Data Items for Output Data . . . 39
2.4 A Class-Based Example . . . 40
2.4.1 Adding a Menu to a Class . . . 41
2.4.2 Add a Menu with a Non-Intrusive Approach . . . 43
2.4.3 Another Non-Intrusive Approach . . . 45
2.5 A Complete *ball* Demo . . . 46
2.6 A More Advanced Example . . . 49

2.6.1 The Problem 49
2.6.2 The Existing Simulation Code 50
2.6.3 The Menu Tree 53
2.6.4 Creating a Menu 54
2.6.5 Examples on Using the Interfaces 57

3 Using DataPool Web: User Manual 59
3.1 Introduction 59
3.2 Menu system 59
3.3 Fundamentals 60
3.3.1 Structure 60
3.3.2 Administration Panel 60
3.3.3 Menu Items (submenus) 61
3.3.4 Data Items 61
3.4 Concepts 62
3.4.1 Hovering 62
3.4.2 Expand & Collapse 62
3.4.3 Linking & Scrolling 63
3.4.4 Menu Item Dependency 65
3.4.5 Menu Chooser 65
3.4.6 Enabled & Disabled Items 66
3.5 Usage 67
3.5.1 Changing Data Item Values 67
3.5.2 Administration Panel 70
3.5.3 Choosing Menu Items (submenus) 73
3.5.4 Value Unit Conversion 78
3.5.5 Balloon Help 80
3.6 Input Error Handling 81
3.6.1 Python Values 81
3.6.2 Wrong Input 81
3.7 Output 83
3.7.1 Plot Visualization 83
3.7.2 Simulation data 84
3.7.3 Combined 87

4 Web frameworks 89
4.1 The role of Python 90
4.2 First Approach 90
4.3 Inter- and intra crosscutting 91
4.4 Introducing Django 92
4.5 Loosely coupled Django 93
4.6 TurboGears 93
4.7 Choice of technologies 94
4.8 Summary 95

5 Technicalities 96
5.1 DPW Contents 96
5.2 Implementation 97
5.2.1 Python 97
5.2.2 Django Templates 97

5.2.3 HTML 98
5.2.4 CSS 98
5.2.5 JavaScript 98
5.2.6 Design Elements 98
5.3 System Structure 99
5.4 Design Philosophy 100
5.5 Core 101
5.5.1 Django 102
5.5.2 DataPool API 103
5.6 Program Flow 104
5.6.1 Approach 104
5.6.2 Initial Browser View 105
5.6.3 Browser Interaction 106
5.6.4 System State Handling 111
5.6.5 Form Creation 113
5.6.6 Form Validation 114
5.6.7 Menu Tree Rendering 115
5.6.8 Interaction: JQuery 117
5.6.9 Alternatives 118
5.7 Variations in DataPool 118
5.7.1 MenuAPI 119
5.7.2 Settings 119

6 Alternative Framework: TurboGears 120
6.1 Intro 120
6.2 Approach 121
6.3 Challenges 121
6.4 Conclusion 123

7 Conclusion 125
7.1 Summary 125
7.2 Contributions 125
7.3 Future Work 127
7.3.1 Visualization of Results 127
7.3.2 Web Service 127
7.3.3 Integration with the FEniCS Project 127

Bibliography 128

List of Figures

2.1 Interactive Command Interface. 31
2.2 Automatically generated HTML documentation for the whole DataPool menu tree. 35
2.3 Web interface for ball2_menu1.py 48

3.1 Initial start-up of DPW 60
3.2 Hover effect on data items 62
3.3 Hover effect on a folded submenu 62
3.4 A collapsed data item showing its belonging attributes 63
3.5 An expanded submenu showing its first level of children submenus 63
3.6 Automatically slided Quick-menu 64
3.7 Parent link for a data item 65
3.8 Anchor indicates dependency rule on a submenu level 65
3.9 A Menu Chooser data item (single dependency) 66
3.10 A Menu Chooser data item (multiple dependency) 66
3.11 A disabled submenu 67
3.12 A disabled data item 67
3.13 A float field 67
3.14 An integer field 68
3.15 Representation of a Python string 68
3.16 A drop-down from the valuelist attribute 69
3.17 Representation of a Python list 69
3.18 Representation of a Python tuple 69
3.19 Representation of a Python tuple (minmax) 70
3.20 Error message using minmax 70
3.21 Error message when defining minmax 70
3.22 Administration Panel 71
3.23 Administration Panel - Status - Saved parameters 72
3.24 Administration Panel - Status - started simulation 72
3.25 Administration Panel - Status - Simulation done 73
3.26 Menu item Quick-menu dependency specification 74
3.27 Dependency: Submenus not chosen, but expanded 75
3.28 Single dependency: Choosing submenu 76
3.29 Single dependency: Submenu chosen 76
3.30 Multiple dependency: Both submenus chosen 77
3.31 Multiple dependency: One submenu chosen 77
3.32 Multiple dependency error: Limits 78
3.33 Multiple dependency error: None value 78

3.34 Data item with defined unit attribute 79
3.35 Changing unit of the input value 79
3.36 Changing both data item value and the unit of the input 80
3.37 Data item value converted 80
3.38 Balloon help 80
3.39 Input error: Float 81
3.40 Input error: List 81
3.41 Input error: Integer 81
3.42 Input error: Tuple 81
3.43 Administration panel error links 82
3.44 Input field errors 83
3.45 Simulation plot results 84
3.46 Ball simulation in DPW 85
3.47 Simulation data results 86
3.48 Combined simulation results 88

5.1 The Django project structure 97
5.2 Program flow - Initial Browser View 106
5.3 Program flow - Browser Interaction - Validated 108
5.4 Program flow - Browser Interaction - Operations 110
5.5 Program flow - Browser Interaction - Errors 111
5.6 Program flow - System State Handling 112
5.7 Program flow - Form Creation 113
5.8 Program flow - Form Validation 115
5.9 Program flow - Menu Tree Rendering 117

6.1 Form generated with TurboGears. 123

Chapter 1

Introduction

1.1 Motivation

In science and engineering, it is common to use software for simulating how nature or technical devices work. Such software is commonly called simulation programs. Most simulation programs are tailored to a particular type or class of problems. Because development of these tailored programs is very expensive, it has been popular to create general-purpose frameworks to simplify the development of simulation programs by assembling various high-level objects from the framework. The purpose of the tool described in this document is to equip a given framework, programmable in Python, with support for easy creation of user interfaces to simulation programs. Many types of user interfaces can be automatically generated by a minimum of code: highly interactive web pages, command line options, XML file input, or a plain text command format.

Simulation programs frequently need a large amount of input data. For example, several hundred parameters are not uncommon in complicated problems. Code dealing with reading input data and initializing data structures can be very tedious to write, especially if the data are to be defined in graphical interfaces. Quite often, managing input demands much more lines of code than the mathematical calculations involved in the simulations. When using general-purpose frameworks, the idea is that the application code should be short, but this is often difficult because managing input may still require a huge effort. Hence, a tool for managing input data is needed such that the application programmer can quickly put together pieces of code that handle all aspects of supplying data to the program.

Quite surprisingly, tools for managing input to simulation programs has received very little attention in the literature. It seems that code related to this issue is widely accepted to be special-purpose – and lengthy. The Diffpack programming environment [13] was one of the first general-purpose frameworks for simulation that contained tools for simplifying the coding of flexible input. Diffpack introduced the concept of a *menu system*. Before going into its details, we need to briefly describe how the Diffpack framework is designed.

Diffpack contains a large number of classes, each class doing some general-purpose operation arising in a simulation, or more precisely, when solving partial differential equations. Each class knows its required input. For example, a class for solving an ordinary differential equation may need (at least) three parameters: the type of numerical method to be used ("method"), the time step (Δt), and the time interval for the solution (T). The class would then, in Diffpack, have a static C++ function defining basic information about these input parameters: the name of the parameter, its default value, a description of the parameter, an optional specification of legal input values, etc. This information can be collected by general-purpose code to create, together with similar information from many other classes, a graphical user interface (GUI) or an interface controlled by commands (for file or "shell" input). Diffpack offers different types of input: command line options, a GUI, commands in a file, and questions and answers in the terminal window. The application programmer provides only basic information about a menu item and decides at run time what type of user interface that is wanted for the current simulation.

To initialize data structures in the class, a function is needed for asking the menu system the value of the parameters. In the case of input from a GUI, the menu system would find the widget corresponding to a parameter and get the user input from this widget. In the case of providing input on the command line, the menu system would search for an option and a value given as a command line argument.

The idea is that the definition and reading of parameters in a class is coded completely independently of whether the user wants to provide input in a GUI, from the command line, or in a file. Each program unit (class) is only concerned with its own input. Moreover, the application code can limit itself to defining the physical parameters in the problem, because most numerical parameters are defined and read in various classes already available in the general-purpose framework. The application code typically applies some of these classes, and these classes often involve many other classes, which again involve other classes in the framework. In this way, one can build the total information about input parameters in a recursive manner. Just a few lines in the application code may result in a menu with hundreds of parameters, arising from a large number of layers of classes in the framework.

DOLFIN [31] and deal.II [3] share some of the Diffpack functionality. However, they have more limited mechanisms for automatically generating various types of user interfaces and for combining parameters from different program units into a coherent menu with a coherent user interface. Another tool called Boost Program Options [23], a part of the Boost C++ Libraries, also have some similar functionality, but is very primitive compared to the solution and outcome of this thesis.

1.2 Contribution

The convenience of specifying input to Diffpack-based simulators has been one of the important reasons for Diffpack's popularity. Unfortunately, Diffpack's

menu system is specific to Diffpack and cannot be used in other programming environments. Moreover, several technical aspects of Diffpack's menu system can clearly be improved. We decided therefore to adopt the fundamental ideas of Diffpack's menu system, but to implement them in a more general way in Python. The result is a package called DataPool [20], which can greatly simplify the creation of user interfaces in Python programs.

We remark that DataPool is not a general tool for creating GUIs. DataPool can only handle input for a set of *parameters*. It cannot be used to create interactive drawings, advanced widgets, or fancy layout of a GUI. Nonetheless, DataPool may use these elements to let the user adjust a large number of physical and numerical parameters by offering a fancy layout and interactivity in simulation programs where it is necessary. This is achieved by using the available interfaces to DataPool, as well as developing new interfaces.

Imagine that we have a simulation code that requires several hundred variables. With DataPool we can easily review and update some of the values while leaving the rest with their default values. We can also select between several user interfaces for updating the values. We can use command line arguments like `--radius 12`, load them from a file, or update them via a graphical web interface. The changes can also be saved to a file that can be edited and loaded back to DataPool next time we want to run the simulation. This is just a short glimpse into DataPool's functionality that will be described in the next chapters.

1.3 Problem Statement

In the book Python scripting for Computational Science by Langtangen [14], some real examples are shown related to this aspect in simulation and visualization. It is particular the simviz1.py script [14, p. 46] for automating simulation and visualization that are basically used in the examples. The section 11.4.2 explains a class hierarchy for holding an input parameter and a class for the generation of CGI scripts [14, p. 553]. The Parameter class [14, p. 563] is used for a data structure that holds the involved parameters. A Parameter instance are passed along to an instance of the AutoSimVizCGI class for automatic generation of a CGI form with the corresponding parameters [14, p. 568]. The CGI scripts generate the appropriate web page with input fields for the parameters to be edited. From this page it is also possible to start the simulation, and the resulting Gnuplot and animation are presented on the same page [14, p. 573].

Parallels can be drawn from the examples to this master thesis. The Parameter class is implemented differently and taken care of with the core of DataPool with the creation of a tree structure for the parameters. Further, the CGI creation can be compared to the main goal of this thesis, which is the creation of a web interface for DataPool with Python web frameworks. In addition, the use of files as input for the generation of GUIs [14, p. 587] is also a part of the DataPool project. Langtangen is also stating the limitation of the tools presented and suggests an approach for bigger systems:

> Applications with a large number of parameters may naturally sort

> these in classes and use menu tree with nested submenus in the interface. Extensions of AutoSimvisGUI and AutoSimVizCGI to menu trees could make use of a directory-tree-like widget for navigation and the parameter setting part of the present version of the classes for each submenu [14, pp. 568 569].

Also, a larger extent of user control of the layout [14, p. 569], could be done with the solution with expanding/collapsing widgets and the power of Ajax.

1.4 Specifying the thesis

The motivation for a master thesis regarding these issues, would be the development of a system which increases the efficiency of the task of handling, starting and accomplishing computational simulations. One important aspect of the thesis should be to focus on making the system as generic as possible. In this way make it highly usable for scientists which are performing simulations in their research. With generic in mind, it should be both practical in use for different types of simulations, but also easy to expand in the future. The expanding could apply new functionality and more built-in support with different middleware. To achieve a loosely coupled system would strengthen the system's position in the future for further development and collaboration from the community, in the spirit of research.

The master thesis are concentrated on the development of a web interface for the DataPool module. Since the DataPool is written in Python, the core of my master thesis will be to realize the web interface focusing on web frameworks for Python. The weight will be on the web framework Django and the other one evaluated is the TurboGears framework. We will compare and reflect on the different approaches. The creation of the web interface for DataPool will in this thesis be the case of using a web browser locally with the interface provided through the GUI module.

1.5 Collaboration and overlap

Rustam Mehmandarov started the development of the core of DataPool as a part of his master thesis, and had the lead of the process from day one. Later, this master thesis was initiated in parallel, supposed to be built on top of the DataPool core as en extension. In the early phase this process mainly consisted of user feedback for DataPool, while developing the web interface DataPool Web ("DPW"), which is the result of this thesis.

However, this process evolved into a tight cooperation, a development process in order to realize the creation of DPW on top of DataPool. Significant and central portions were developed in collaboration in the *Extreme Programming* spirit, and others were developed individually. The process spanned from technical code, tailoring of the API and defining new concepts for the module.

In addition to the implementation of real computer simulation examples, which were used to optimize the usability of DataPool.

The user manual for DataPool is written in cooperation with Hans Petter Langtangen and Rustam Mehmandarov, this also includes the example codes.

1.6 Outline

The next chapter contains the user manual for DataPool. The chapter is introducing the terminology, principles and the functionality. Different examples, variating greatly in complexity, are used to show and cover the usage of the system. Other interfaces are also examined in this user manual, and the last parts show an example of a demo walkthrough involving the most aspects provided by the system.

Chapter 3 contains the user manual for DPW. All fundamentals, concepts and usage of the web interface are covered. The user manual mainly uses the Oscillator simulation throughout the chapter, but other simulations are also involved to show the functionality and system behavior.

Chapter 4 introduces the world of web frameworks for Python and discuss the role of Python, Django and the choice of technologies for DPW.

Chapter 5 describes all the technicalities of the DPW module regarding to implementation, design philosophy, program flow and structure.

Chapter 6 describes the use of TurboGears as an alternative web framework for implementing DPW.

Chapter 7 takes a look at the road for future development and conclude the thesis.

1.7 Conventions in the thesis

The thesis are using different typefaces in order to indicate programming code, command and etc.

Code snippets are shown with a light blue box:

```
print('This is a python code snippet')
```

We indicate a complete runnable program like this, with a thin dark blue border on the left side:

```
import sys

def msg():
        msg = 'this is a complete program'
        print msg
```

```
        print sys.version

msg()
```

Other denoting of code and commands are done with monospace:

`This is the monospace for code and commands.`

Terminal commands are presented like this:

Terminal

```
This is a command in a terminal window.
```

Data in files are denoted like this:

```
<?xml version="1.0" ?>
<menu name="/">
   <menu name="/Hard Kick/">
      <data_item name="/Hard Kick/C_D">
         <attribute name="widget">entry</attribute>
```

Chapter 2

Using DataPool: User Manual

2.1 Introduction

2.2 Basic Ideas of DataPool Usage

DataPool manages a (possibly large) set of input parameters in simulation programs. These input parameters are grouped in various submenus in a tree-like fashion. For example, one may have a main menu with a submenu for numerical parameters and another submenu for physical parameters entering the mathematical model. The submenu for numerical parameters may have different submenus for parameters in different numerical algorithms used in the simulator. Similarly, the submenu for physical parameters may contain various submenus for the parameters that enter various physical models.

Each submenu consists of two types of objects: data items and submenus. A *data item* holds information about a parameter, typically the name of the parameter, its default value, its current value, plus optional information about legal values, the associated unit (meter, second, etc.), and what kind of widget that is suitable for setting the parameter in a graphical user interface. A *submenu* object has a name and a list of data items and other submenu objects. Submenus are also sometimes referred to as *menu items* in DataPool (especially in the source code).

The idea behind DataPool is that various parts of a big program can make their own nested submenus, or *menu trees* as we denote them, and attach their trees to a global menu tree for the whole simulation code. In this way, a comprehensive user interface can be built from many small, independent menu trees.

To provide an image of a menu tree in DataPool, we can think of a file and directory structure on a computer. A directory may contain files and other directories. The files correspond to data items, while the directories correspond to the submenus. Having a directory tree, we know how easy it is to copy or move it to a directory in another directory tree and extend that tree. DataPool's menu trees are of the same nature.

As soon as the global menu tree is built, the user of the program can specify values in various interfaces: in a web page, on the command line, or in a file. Thereafter, the program can load the parameter values into variables and perform computations. Examples will show how this is done in practice.

2.2.1 The Three Key Steps

Using DataPool consists of three phases:

- Define submenus and data items.
- Start user interface.
- Load data from DataPool into variables used for computation.

DataPool's application programming interface (API) allows a variety of syntax for performing the first and third steps. Some syntax is tailored to small programs that rapidly need to be equipped with a user interface, while other parts of the syntax are better suited for larger software systems with a lot of parameters and submenus scattered around in numerous program units.

Here we briefly present a simple subset of the DataPool API needed in the first examples. Using DataPool starts with an `import` and is followed by grabbing its *global menu tree*[1]:

```
import DataPool
menu = DataPool.menu # global menu tree
```

With the `menu` variable we can add data items and submenus to the global menu tree. Different parts of a Python application may access the global menu tree as a global variable in the application.

2.2.2 Working with Submenus

Initially, the tree will consist of a root submenu containing all other submenus and data items that can be added by the user. The root submenu, like the root directory in Unix-like operating systems, is designated by a forward slash (`/`). It is possible to set this submenu delimiter to another character[2].

Creating a new submenu with its name in a string `name` is done by

```
menu.submenu(name)
```

When creating new submenus and data items, they will automatically be added to the current submenu (also called *current working menu*). The `submenu` command is also used for navigating in the menu tree[3]. As an example, think of

[1] Global menu tree is a global object in the DataPool package. Different parts of a system can access this object and add submenus, and thereby contribute to building a (possibly large) common menu tree.

[2] The forward slash is used as a default path delimiter. However, it can be modified in `settings.py` located at the root directory of the package.

[3] In Unix terms, using a directory tree as a model of DataPool's menu tree, one may say that `menu.submenu` corresponds to a `cd` command, but `mkdir` is automatically run first if one moves to a non-existing directory.

moving to submenu s3 that is in submenu s2, again contained in submenu s1. Being in the s1 submenu we can use a local path:

```
menu.submenu('s2/s3')
```

Being anywhere in the menu tree, we can also use an absolute path. It will always have a slash (/) at the beginning of the string:

```
menu.submenu('/s1/s2/s3')
```

The local, or relative, path is interpreted relatively to the current working menu. After each menu.submenu command, DataPool updates its current working menu and sets it to be the specified menu.

This usage of paths can remind users of handing paths in the most common operating systems. As expected, it is possible to use one or several ".." in the relative paths for moving to a parent submenu. For instance:

```
menu.submenu('../../')
menu.submenu('../s4/s5')
```

The former statement will move us two levels up in the menu hierarchy (i.e. to s1), while the latter will first move us up to submenu s2, and then go to, and if necessary create, s4 and s5.

2.2.3 Specifying Data Items

As we will see, the add command offers various ways of adding input data which can be used interchangeably depending on the problem at hand.

Method 1: Positional Parameters

Suppose we have a parameter called "time step". Its default value is 0.1, and it is measured in seconds. The minimal specification of this data item is to set its name and default value:

```
menu.add('time step', 0.1)
```

However, we may specify more information if desired:

```
menu.add('time step', 0.1, unit='s',
         help='time discretization parameter in the ODE solver',
         widget='entry')
```

Now we specify the unit as 's' for second. If a graphical user interface is available, we want to use an entry widget ('entry') in that user interface (entry means a simple one-line text box). We also specify a help or documentation string that will appear in automatically generated documentation of the whole menu tree. In graphical user interfaces this help string will typically pop up on mouse-overs for the item.

It is worth noticing that the `add` command will always add data items to the current working menu, which is set to the menu tree root by default on initialization of a new tree. To add data items to the correct submenu, this command should be used in combination with the `menu.submenu` command.

An alternative to specify input parameters is to use a Python list notation to add several data items with one command:

```
menu.add([('density', 1.2, 'density of air', 'kg*m**(-3)'),
          ('radius', 0.11, 'radius of ball', 'm'),
          ('mass', 0.43, 'mass of ball', 'kg'),
         ])
```

This way of adding data items requires the information to be given in a specific order: name, default value, help string, unit, widget type, tuple containing minimum and maximum values. There are other attributes that can be set, and these must be specified as keyword arguments of the form `name=value`.

Method 2: Dictionaries

More flexibility in defining data items can be achieved by using Python dictionaries:

```
# create dictionaries:
dict1 = dict(name='density', value=1.2, help='density of air',
             unit='kg*m**(-3)'),
dict2 = dict(name='radius', value=0.11, help='radius of ball',
             unit='m'),
dict3 = dict(name='mass', value=0.43, help='mass of ball', unit='kg')

# single add:
menu.add(dict1)

# multiple add:
menu.add([dict2, dict3])
```

Method 3: DataItem Objects

Not surprisingly, data items are found in the DataPool source code as instances of class `DataItem`. We may define such instances directly when creating menus:

```
# create objects:
obj1 = DataItem('density', 1.2, 'density of air', 'kg*m**(-3)')
obj2 = DataItem('radius', 0.11, help='radius of ball', unit='m')
obj3 = DataItem('mass', 0.43, unit='kg', help='mass of ball', )

# single add:
menu.add(obj1)

# multiple add:
menu.add([obj2, obj3])
```

Method 4: Nested Dictionaries

Sometimes menu subtrees can be easier to define as nested dictionaries. For programs written in languages not supporting Python objects, nested dictionaries

might facilitate the generation of DataPool menu subtrees. Let us see how this can be done. Assume that we have a subtree with the following structure:

```
Submenu 'main'/
        Data item 'x', 'value':1, 'unit':'cm'
        Data item 'y', 'value':0.1
        Data item 'z', 'value':0.2
        Submenu 'sub1'/
                Data item 'a', 'default':'3'
                Data item 'b', 'value':0.01
                Data item 'c', 'value':0.11
                Submenu 'sub2'/
                        Data item 'd', 'value':10
        Submenu 'sub3'/
                Data item 'e', 'value':'e1'
                Data item 'f', 'value':'f1'
```

This can be represented as nested dictionary and added to the global menu tree:

```
# the subtree to be added
tree = [ 'main', [{'name':'x', 'value':1, 'unit':'cm'},
                  {'name':'y','value':0.1},
                  {'name':'z','value':0.2},
                  'sub1', [{'name':'a', 'default':'3'},
                           {'name':'b','value':0.01},
                           {'name':'c','value':0.11},
                            'sub2', [{'name':'d','value':10},
                                     ],
                           ],
                   'sub3', [{'name':'e','value':'e1'},
                            {'name':'f','value':'f1'},
                            ],
                    ],
        ]

# add to the global menu tree:
menu.add(tree)
```

This solution also provides more flexibility with respect to the number and the position of the attributes that each data item may have, as well as the ability to define submenu structures in a simple way.

Method 5: Menu Trees

Imagine that you have several Python modules with a menu tree each, and you want to add these trees somewhere in a global tree. In this case you can simply move to the correct submenu, and do the following for each of the subtrees you wish to add:

```
# single add:
menu.add(submenu_tree_obj)
```

It is also possible to use lists to add several objects:

```
# multiple add:
menu.add([submenu_tree_obj1, submenu_tree_obj2, submenu_tree_obj3])
```

The `submenu_tree_obj` objects are the instances of the MenuItem class, and can be instantiated as shown in this example:

```
submenu_tree_obj1 = MenuItem(menu, 'test_menu1', None)
submenu_tree_obj2 = MenuItem(menu, 'test_menu2', submenu_tree_obj1)
submenu_tree_obj3 = MenuItem(menu, 'test_menu3', submenu_tree_obj2)
```

2.2.4 Starting the User Interface

When all submenus and data items are specified, the program must ask the user for input. DataPool offers several different types of user interfaces. For some of them, like the graphical user interface, the control is handed over to the user, who can examine submenus and data items and modify parameter values. Other interfaces, like the command line or a file with menu commands, just reads user-provided information to update the values in data items.

To read user input, or "start the user interface" as we say in this tutorial, the program must have a call on the form

```
menu.start_ui('desired_ui', <other args>)
```

The command expects a string defining which user interface it will be running, and other user interface-specific arguments. This command performs actions that strongly depend on the user interface that was chosen when the program was started.

Operating each interface will be described in detail in Section 2.3. Now, let us have a brief look at the available interfaces.

Command-Line Arguments

The command-line interface allows the values of the data items to be specified through command-line arguments, provided when the program is started. The specific `menu.start_ui` call for this user interface reads

```
# command line interface:
menu.start_ui('cmd')
```

Interactive Command Interface

The interactive command line interface is built on the principle of an interactive dialog in a terminal window where the user can view the menu tree, move around, and assign values to data items (the interface works much like a Unix shell).

```
# interactive command interface:
menu.start_ui('shell')
```

File Interface

It is also possible to write a file with commands for setting values of data items. This file can be given to the `menu.start_ui` call, and DataPool will then read the file and update values in the menu tree:

```
# interactive command interface with
# commands saved in a text file:
menu.start_ui('shell', file_name.i)
```

The commands in the file have the same syntax as the commands in the interactive command-line interface (i.e., if the latter is thought of as a Unix shell, the file interface is a Unix shell script).

Web Interface

DataPool has a graphical user interface, which is operated through a web browser. The user is presented with a web page containing a graphic representation of the menu tree, with a possibility to navigate and update data item's values. This interface is started with the following options:

```
# web interface (default port number and URL):
menu.start_ui('web')

# web interface (customized):
menu.start_ui('web', 8010, "http://localhost:8010/new_datapool/")
```

2.2.5 Extracting Parameter Values

After the user of the program has provided information about parameter values in the menu tree, it is possible to access a particular value by its name:

```
dt = menu.get('time step')
```

This command is successful as long as we are in the right submenu or if there is only one data item with name "time step" in the whole menu tree. Otherwise we have to navigate to the right submenu with the `menu.submenu` command before calling `menu.get`, or use a relative or a full path to the data item.

```
# example of get() with a relative path:
dt = menu.get('../another_submenu/time step')
```

2.2.6 Specifying and Getting Multiple Parameters

Sometimes it is tedious to write a set of separate `add` and `get` calls to specify and read several parameters. The DataPool API offers a shortcut in this case. For instance, we may specify a list of data items, where each item can be a tuple with elements for the name, the default value, the help string, the unit, the widget, and the tuple with minimum and maximum allowed values – in that order. As a minimum, the name and the default value must be provided. All possible ways of adding multiple parameters are described in Section 2.2.3. Here is a short example:

```
data_items = [
  ('C_D', 0.2, 'drag coefficient'),
  ('rho', 1.2),
  ('V', 1, 'velocity', 'm/s')]

menu.add(data_items)
```

Loading the parameter values associated with these items can also be done with a single call:

```
C_D, rho, V = menu.get(['C_D', 'rho', 'V'])

# or:
C_D, rho, V = menu.get([d[0] for d in data_items])
```

It is also worth noticing that the names in the list sent to the function can consist of a simple item name or a full or a relative path to this item. For instance:

```
C_D, rho, V = menu.get(['../C_D', 'rho', '/volume/V'])
```

The return value will always be a tuple with all values, or just a single value if only one parameter was specified.

2.3 An Introductory Worked Example

After having reviewed the basic ideas of DataPool usage, it is time to apply the methods in an example. This section presents a very simple program having a few input parameters and trivial mathematical computations. We first describe the program with hardcoded values of all parameters. Then we show alternative ways of using DataPool to equip the program with a menu for flexible reading of input. The first version of the program is just a simple, "flat" code. A class-based version is treated in Section 2.4. The basic usage of DataPool explained in Sections 2.3 and 2.4 is demonstrated in a more complicated and realistic simulation code in Section 2.6 where a quite general ordinary differential equation describing nonlinear oscillations with many parameters is solved.

The purpose of our introductory simulation program is to compute the drag and gravity forces on a body moving through air. Such computations help to determine if the drag can be neglected in certain applications. A rough formula for the drag force is $\frac{1}{2}C_D\rho AV^2$, where C_D is a dimensionless drag coefficient for the body, ρ is the density of the surrounding fluid (e.g., air), A is the cross-section area of the body normal to the velocity direction, and V is the velocity of the body. In case of a ball of radius a, we have that $A = \pi a^2$. The gravity force is mg, where m is the mass of the body and g is the acceleration of gravity.

2.3.1 The Basic Program

Let us make a simple program that computes the drag and gravity forces in the case of a soft and hard kick of a football[4]:

```
def drag(C_D, rho, A, V):
    return 0.5*C_D*rho*A*V**2
```

[4] For the fun of it we may run this code and realize that the drag force is negligible for a soft kick (10 km/h), while in a really hard kick (120 km/h) the drag force is as important as gravity. This is important information for deciding whether one can neglect air resistance or not.

```
def gravity(m):
    g = 9.81  # m*s**(-2)
    return m*g

C_D = 0.2        # drag coefficient, dimensionless
rho = 1.2        # density of air, kg*m**(-3)
a = 0.11         # radius of a ball (standard football), m
from math import pi
A = pi*a**2      # cross section area normal to movement, m^2
m = 0.43         # mass of body, kg
V_hi = 120       # velocity, hard football kick, km/h
V_hi = V_hi/3.6  # velocity in m/s
V_lo = 10        # velocity, soft football kick, km/h
V_lo = V_lo/3.6  # velocity in m/s


# "simulate":
hard_kick_drag = drag(C_D, rho, A, V_hi)
soft_kick_drag = drag(C_D, rho, A, V_lo)
gravity_force = gravity(m)

print """
Gravity force = %.2f N
Drag force, hard kick = %.2f N
Drag force, soft kick = %.2f N
""" % (gravity_force, hard_kick_drag, soft_kick_drag)
```

This program is stored in the file `ball1.py` in the `doc/samples/ball` directory of the DataPool source code.

2.3.2 Adding DataPool Functionality

Let us add statements to the `ball1.py` code so that we can set and get parameters via the DataPool tool instead of hardcoding values in the program.

Working with Separate Calls for Each Data Item. We start with filling a main menu with data items by calling `menu.add` as outlined in Section 2.2.3:

```
import DataPool
menu = DataPool.menu

menu.add('C_D', 0.2, help='drag coefficient', minmax=[0,1])
menu.add('density', 1.2, help='density of air', unit='kg*m**(-3)')
menu.add('radius', 0.11, 'help=radius of ball', unit='m')
menu.add('mass', 0.43, help='mass of ball', unit='kg')
menu.add('V high', 120, help='velocity of hard football kick',
         unit='km/h')
menu.add('V low', 10,   help='velocity of soft football kick',
         unit='km/h')
```

Now the menu contains all necessary information about the input parameters to the program. Note that the `help`, `unit`, and `minmax` keywords constitute optional information that we could have skipped.

The next step is to ask the user for input,

```
menu.start_ui()
```

This enables the user to provide values for the various data items, but we postpone to Sections 2.3.4–2.3.6 to show how to operate the various interfaces and

set the parameter values.

After this prompt phase we are ready to load the user-given data into variables and continue with the mathematical computations. Retrieval of parameter values is done with `menu.get`:

```
C_D = menu.get('C_D')
rho = menu.get('density')
a =  menu.get('radius')
m = menu.get('mass')
V_hi = menu.get('V high')
V_lo = menu.get('V low')
```

At this stage we have filled the same variables as in the original program and we can perform the same computations. The complete program utilizing DataPool (and stored in the file `ball1_sep1.py`) looks as follows:

```
def drag(C_D, rho, A, V):
    return 0.5*C_D*rho*A*V**2

def gravity(m):
    g = 9.81  # m*s**(-2)
    return m*g

import DataPool
menu = DataPool.menu

menu.add('C_D', 0.2, help='drag coefficient', minmax=[0,1])
menu.add('density', 1.2, help='density of air', unit='kg*m**(-3)')
menu.add('radius', 0.11, 'help=radius of ball', unit='m')
menu.add('mass', 0.43, help='mass of ball', unit='kg')
menu.add('V high', 120, help='velocity of hard football kick',
         unit='km/h')
menu.add('V low', 10,   help='velocity of soft football kick',
         unit='km/h')

# menu ready, prompt user:
menu.start_ui()

# extract user's input:
C_D = menu.get('C_D')
rho = menu.get('density')
a =  menu.get('radius')
m = menu.get('mass')
V_hi = menu.get('V high')
V_lo = menu.get('V low')

from math import pi
A = pi*a**2         # cross section area normal to movement, m^2
V_hi = V_hi/3.6     # velocity in m/s
V_lo = V_lo/3.6     # velocity in m/s

# "simulate":
hard_kick_drag = drag(C_D, rho, A, V_hi)
soft_kick_drag = drag(C_D, rho, A, V_lo)
gravity_force = gravity(m)

print """
Gravity force = %.2f N
Drag force, hard kick = %.2f N
Drag force, soft kick = %.2f N
""" % (gravity_force, hard_kick_drag, soft_kick_drag)
```

Putting the Data Items in a Submenu. The above example stored all the data items in the nameless root menu. We can explicitly create a submenu, say "Ball Simulation", and let all data items be members of this submenu. This is easily done by inserting a `menu.submenu` call right before the `menu.add` calls:

```
menu.submenu('Ball Simulation')

menu.add('C_D', 0.2, help='drag coefficient', minmax=[0,1])
menu.add('density', 1.2, help='density of air', unit='kg*m**(-3)')
...
```

Different submenus may have data items with the same name, so we sometimes need the *full path* of a data item, which is the name prefixed by all submenu names separated by a forward slash. For example, the name `density` has the full path `Ball Simulation/density` in the current example. This is the same naming convention as used for files and directories in Unix systems. However, in DataPool, we may skip the submenu prefix if the data item name is unique. Since we in the present example have only one item with the name `density`, we can use the short name `density` instead of the full path `Ball Simulation/density`.

In the `menu.get` calls we must either use full paths, or move to the right submenu first, or be sure that the short name is unique:

```
C_D = menu.get('/Ball Simulation/C_D') # full path

# or move to the right submenu,
menu.submenu('/Ball Simulation')
C_D = menu.get('C_D')

# or, if C_D is a unique name,
C_D = menu.get('C_D')
```

The complete program that has all data items on a submenu can be found in the file `ball1_sep2.py`.

Working with Bulk Calls for Multiple Data Items. Instead of making separate calls to `menu.add` and `menu.get` we can perform bulk operations on a list of items. As mentioned in Section 2.2.6, we first create a list of tuples or lists holding information about each data item. The elements making up the data about one item are (in order): The name of the item, the default value, and optionally a help string, a unit specification, a data type specification, and a widget specification. Our six input parameters can be specified as follows:

```
data_items = [
    ('C_D', 0.2, 'drag coefficient'),
    ('density', 1.2, 'density of air', 'kg*m**(-3)'),
    ('radius', 0.11, 'radius of ball', 'm'),
    ('mass', 0.43, 'mass of ball', 'kg'),
    ('V high', 120, 'velocity of hard football kick', 'km/h'),
    ('V low',  10 , 'velocity of soft football kick', 'km/h'),
    ]

menu.add(data_items)
```

Note that we here just add the items to the root menu. We could add them to a submenu by calling `menu.submenu` prior to `menu.add`. (Recall that with

separate `menu.add` calls, or with a list of dictionaries, or a list of `DataItem` objects, we can provide more optional information.)

The retrieval of all parameter values at once is done by

```
names = [item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)
```

Accuracy is important here: The sequence of variables on the left-hand side of the `menu.get` call must exactly match the right order and number of names in the `data_items` list! If we add the data items to a submenu, we need in the general case to specify full paths, i.e., prefix the item names by their submenu paths:

```
names = ['/Ball Simulation/%s' % item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)
```

In the present examples the submenu prefix is not required since the short names are unique within the whole menu.

The rest of the program is the same as in the previous examples. For completeness we show the whole code (found in `ball1_bulk.py`):

```
def drag(C_D, rho, A, V):
    return 0.5*C_D*rho*A*V**2

def gravity(m):
    g = 9.81  # m*s**(-2)
    return m*g

import DataPool
menu = DataPool.menu

data_items = [
    ('C_D', 0.2, 'drag coefficient'),
    ('density', 1.2, 'density of air', 'kg*m**(-3)'),
    ('radius', 0.11, 'radius of ball', 'm'),
    ('mass', 0.43, 'mass of ball', 'kg'),
    ('V high', 120, 'velocity of hard football kick', 'km/h'),
    ('V low',  10 , 'velocity of soft football kick', 'km/h'),
    ]

menu.add(data_items)

# menu ready, prompt user:
menu.start_ui()

# extract user's input:
names = [item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)

from math import pi
A = pi*a**2         # cross section area normal to movement, m^2
V_hi = V_hi/3.6     # velocity in m/s
V_lo = V_lo/3.6     # velocity in m/s

# "simulate":
hard_kick_drag = drag(C_D, rho, A, V_hi)
soft_kick_drag = drag(C_D, rho, A, V_lo)
gravity_force = gravity(m)
```

```
print """
Gravity force = %.2f N
Drag force, hard kick = %.2f N
Drag force, soft kick = %.2f N
""" % (gravity_force, hard_kick_drag, soft_kick_drag)
```

2.3.3 DataPool Command-Line Options

The `--datapool-ui` command-line option can always be used for setting the type of interface. By default, its value is set to command line interface. It is important to note that this argument will always override the interface specified in a `start_ui()` call. For example, say we want to run our program with a web-based graphical interface:

Terminal

```
python ball1_sep2.py --datapool-ui "web"
```

Alternatively, we can set data item values through commands in a file with path `mypath/somefile.i`:

Terminal

```
python ball1_sep2.py --datapool-ui "mypath/somefile.i"
```

Another important command-line option is `--datapool-file`, which allows default values for data items to be specified in a file. This file has the same syntax as the file or shell interface. Values in this file override default values set when data items are defined in the code. Say we have created a file `values.i` with the contents

```
submenu Ball Simulation
C_D = 0.3
density = 1.3
```

Running

Terminal

```
python ball1_sep2.py --datapool-file values.i --datapool.ui web
```

will set the `C_D` and `density` items to have default values 0.3 and 1.3 (instead of 0.2 and 1.2). That is, when the web-based graphical interface appears on the screen, we see the values 0.3 and 1.3 in the text fields for `C_D` and `density`.

The default values set by `menu.add` commands are hardcoded. Some applications may require changes of hundreds of default values in a large menu. We can then have more relevant default values in a file and load this file with the `--datapool-file` option before operating an interface. In that interface we can perhaps change just a few values for the particular run.

To see the different options that are available, we can type

Terminal

```
python ball1_sep2.py --help
```

The result will be as follows:

Terminal

```
rustamm@ubuntu:~$ python ball1_sep2.py --help
Usage:

scriptname.py [--data_item_name1 value1 ...] [--datapool-ui]  [--datapool-file]

datapool-file -- is the path to a file with DataPool commands
datapool-ui -- is the desired UI.
```

A more comprehensive help is printed by the `--help-all` option. This command will show all command-line arguments and the full list of data items that can be adjusted on the command line.

2.3.4 Operating the Command Line Interface

DataPool's simplest user interface reads option-value pairs from the command line. A data item with name "name" gives rise to an option `--name`, and the proceeding command-line argument represents the user-specified value of the item. In case of item names with blanks, the blanks are substituted by underscores in the option name. For example, the name "V high" implies the option `--V_high`, and to set this parameter to 100, we specify `--V_high 100`.

In our example in the file `ball1_sep1.py` from Section 2.3.2, DataPool creates six command-line options: `--C_D`, `--density`, `--radius`, `--mass`, `--V_high`, and `--V_low`.

When the data items are put on a submenu "Ball Simulation", as in the program `ball1_sep2.py`, the option names equal the full path prefixed by a double hyphen: `--/Ball_Simulation/C_D`, `--/Ball_Simulation/density`, and so forth.

However, if the data item name is unique in the menu tree, the submenu prefix can be skipped, so `--density` is a legal shortcut for `--/Ball_Simulation/density`.

Here are two examples running three of our sample codes with a command line interface:

Terminal

```
python ball1_sep1.py --radius 12 --V_high 100 --V_low 8
python ball1_sep2.py --/Ball_Simulation/V_high 100
python ball1_bulk.py --density 1000 --V_high 10 --V_low 1
```

We only set the options where we want to change the default value.

To see the full list of available command-line arguments one must issue a `--help-all` option.

2.3.5 Operating the Interactive Command Interface

The interactive command interface is an text-based shell-like user interface that can interpret a set of commands necessary for navigating the menu tree and updating data item's values. A screenshot of this interface is presented in Figure 2.1.

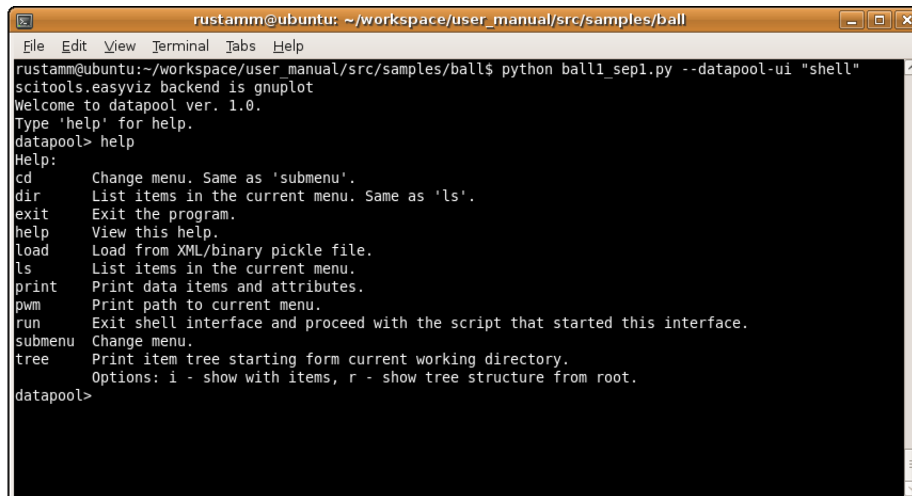

Figure 2.1: Interactive Command Interface.

The interface can be started with the `start_ui('shell')` command. A list of all available commands can be seen when the `help` command is issued. It is also possible to type help <desired command> that will show a more detailed description of each command:

Terminal

```
datapool> help tree
Print item tree starting form current working directory. Options:
i - show with items, r - show tree structure from root.
Usage: tree [-i] [-r]
datapool>
```

To update a value we need to move to the correct submenu using the `submenu` command, and type the data item name followed by the value. Here is a sample session:

Terminal

```
datapool> tree

|-- /
   |-- Ball Simulation/
datapool> submenu Ball Simulation
datapool> ls
./
../
C_D
density
```

```
radius
mass
V high
V low

Total: 0 menu(s) and 6 item(s).
datapool> V high = 100
datapool> print V high
DataItem: name=V high (default_value=120)
{'enumerator': None,
 'help': 'velocity of hard football kick',
 'minmax': None,
 'str2type': <type 'int'>,
 'uncertainty': None,
 'unit': 'km/h',
 'validate': None,
 'value': 100,
 'valuelist': None,
 'widget': 'entry'}
```

2.3.6 Operating the File Interface

The file-based interface consists of ASCII text in a file, using the shell-like syntax explained in the previous section. (The interactive command interface resembles an interactive Unix shell, while the file interface is the counterpart to a Unix shell script.) The file interface can be started with the

```
start_ui('shell', path='data.i')
```

command. Alternatively, it can be started by adding the `--datapool-ui data.i` command line arguments (as shown in Section 2.3.3).

2.3.7 Unit Conversion

Mixing units in input is known to be a frequent and serious type of error in engineering, so DataPool's support of automatic unit conversion is a very important feature [15]. For example, we may in the file write

```
radius = 10 cm
```

or we may on the command line set `--radius '10 cm'`, or we may in a graphical user interface fill in `10 cm` in the text field for the `radius` item. If a unit was provided along with a value, the value is automatically converted if the unit differs from the unit specified for this data item in the `menu.add` call. In the example above, the value `10 cm` becomes 0.1 since `radius` was specified to have unit measured in meters (`'m'`).

2.3.8 Writing the Menu Tree to File

It may be convenient to dump the menu tree, with new values filled out by the user, to a file. The values in the file can then be used as default values in a future run of the program.

We can write out the definition of the data items and submenus in several formats, using the `write` command:

```
# plain file:
menu.write('data.i')
# or:
menu.write('data.i', format='shell')
```

With the `'shell'` as format we get a file `data.i` with the shell-like syntax, so we can set `--datapool-file data.i` in a future run and hence start the (say) web interface in that run by the set of values from the current run. This feature may minimize the need for adjusting values in the interface, which is important in simulation programs with a large number of data items and hence a potentially large number of edits in the interface.

Here is an example of a file with `'shell'` syntax, created by the `menu.write` command above:

```
submenu /Hard Kick/
C_D = 0.2
density = 1.2 kg*m**(-3)
radius = 0.11 m
mass = 0.43 kg
velocity = 120 km/h
submenu /Soft Kick/
C_D = 0.2
density = 1.2 kg*m**(-3)
radius = 0.11 m
mass = 0.43 kg
velocity = 10 km/h
```

DataPool offers several other file formats: a list of command-line arguments, XML code, and pure (DataPool) Python code. The files in all formats, except XML, contain each data item's path, name, value, and unit (if any). The XML format contains all attributes of all data items.

For example, the following call generates a complete set of all command-line options and values from the current menu:

```
# command-line options and values:
menu.write('data.i', format='cmd')
```

The generated `data.i` file contains the text

```
--/Hard Kick/C_D 0.2 --/Hard Kick/density '1.2 kg*m**(-3)' \
--/Hard Kick/radius '0.11 m' --/Hard Kick/mass '0.43 kg' \
--/Hard Kick/velocity '120 km/h' --/Soft Kick/C_D 0.2 \
--/Soft Kick/density '1.2 kg*m**(-3)' --/Soft Kick/radius '0.11 m' \
--/Soft Kick/mass '0.43 kg' --/Soft Kick/velocity '10 km/h'
```

This text, or parts of it, can be cut and pasted into a terminal window in a future run of the program.

The Python code format is specified by a call like

```
# Python code for updating the values in the menu:
menu.write('data.i', format='python')
```

The resulting `data.i` file now contains code for creating the current menu with DataPool calls:

```
import DataPool
menu = DataPool.menu
menu.submenu('/Hard Kick/')
menu.set('C_D', '0.2')
menu.set('density', '1.2 kg*m**(-3)')
menu.set('radius', '0.11 m')
menu.set('mass', '0.43 kg')
menu.set('velocity', '120 km/h')
menu.submenu('/Soft Kick/')
menu.set('C_D', '0.2')
menu.set('density', '1.2 kg*m**(-3)')
menu.set('radius', '0.11 m')
menu.set('mass', '0.43 kg')
menu.set('velocity', '10 km/h')
```

Running

```
# XML format:
menu.write('data.i', format='xml')
```

generates an XML file with has a look similar to the following text (the output was simplified to contain only one data item in each menu):

```
<?xml version="1.0" ?>
<menu name="/">
   <menu name="/Hard Kick/">
      <data_item name="/Hard Kick/C_D">
         <attribute name="widget">entry</attribute>
         <attribute name="enumerator">None</attribute>
         <attribute name="help">drag coefficient</attribute>
         <attribute name="uncertainty">None</attribute>
         <attribute name="value">0.2</attribute>
         <attribute name="valuelist">None</attribute>
         <attribute name="minmax">None</attribute>
         <attribute name="validate">None</attribute>
         <attribute name="str2type">&lt;type 'float'&gt;</attribute>
         <attribute name="unit">None</attribute>
      </data_item>
   </menu>
   <menu name="/Soft Kick/">
      <data_item name="/Soft Kick/density">
         <attribute name="widget">entry</attribute>
         <attribute name="enumerator">None</attribute>
         <attribute name="help">density of air</attribute>
         <attribute name="uncertainty">None</attribute>
         <attribute name="value">1.2</attribute>
         <attribute name="valuelist">None</attribute>
         <attribute name="minmax">None</attribute>
         <attribute name="validate">None</attribute>
         <attribute name="str2type">&lt;type 'float'&gt;</attribute>
         <attribute name="unit">kg*m**(-3)</attribute>
      </data_item>
   </menu>
</menu>
```

The mentioned file formats can also be used to let another program read the menu and generate an interface. The user can operate this interface, and the program can write the data items and their values to file using the shell-like syntax. This file can be loaded into a program using DataPool. In this way, one can let another program supply the user interface, but define the menu with DataPool and get the values also by asking a DataPool menu tree.

2.3.9 Automatic Generation of Documentation

For large menu trees containing several hundreds of parameters it is essential to have a proper detailed documentation for each data item and its attributes.

Although one can use, e.g., the graphical web interface to browse the whole tree and learn about data items and submenus, many will prefer to look up information about all input parameters to a program in a separate manual, either on the screen or on paper. Such a manual can be generated by a call to the `menu.write()` function with the `format='*-doc'` argument, as in this example:

```
menu.write('/home/myusername/documents/', format='html-doc')
```

A complete documentation of all data items and submenus is now written to documents in the specified folder. The present call results in an HTML page[5] as shown in Figure 2.2. As we can see from the figure, the page contains an overview of the menu tree with all the submenus and their corresponding data items, and the right-hand side contains links to ease the navigation between submenus.

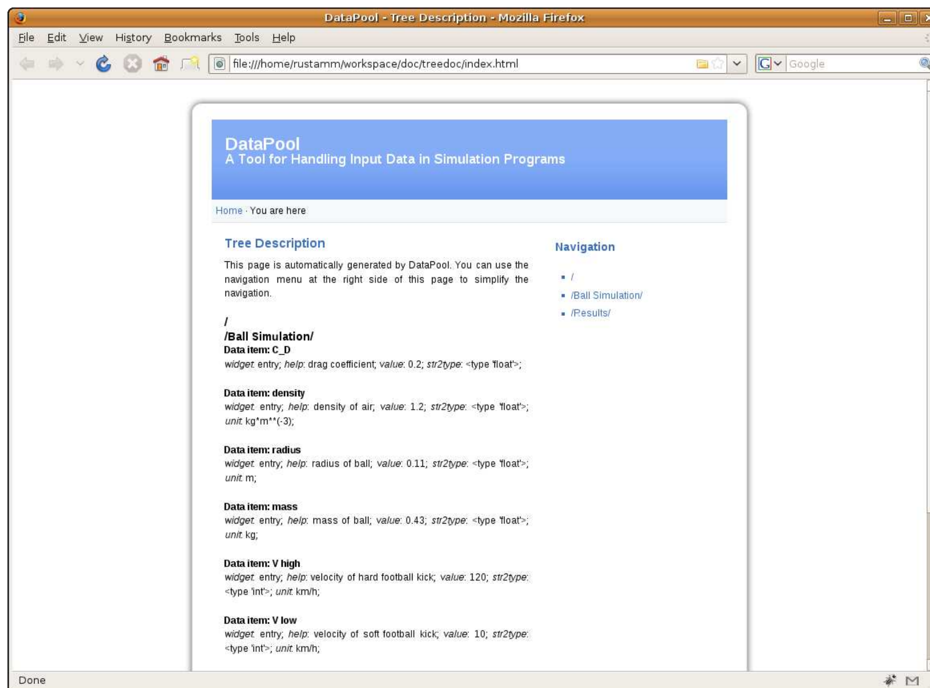

Figure 2.2: Automatically generated HTML documentation for the whole DataPool menu tree.

2.3.10 More Advanced Specification of Data Items

There is even more to the DataPool API than shown so far. Each data item is, quite naturally, an instance of type `DataItem`. We can build such instances directly:

[5]Currently, the only format available is HTML. However, this function is designed to support several formats that could be implemented in a future version.

```
data_items = [
    DataItem('C_D', 0.2, help='drag coefficient', minmax=[0,1],
             widget='slider'),
    DataItem('density', 1, help='density of air',
             unit='kg*m**(-3)', str2type=float),
    DataItem('radius', 0.11, help='radius of ball', unit='m'),
    DataItem('mass', 0.43, help='mass of ball', unit='kg'),
    DataItem('V high', 120,
             help='velocity of hard football kick', unit='km/h'),
    DataItem('V low',  10 ,
             help='velocity of soft football kick', unit='km/h'),
    ]
```

Note that some new keyword arguments are introduced here. These are also available in `menu.add` calls (but not when making simple lists of tuples of information as in the `ball1_bulk.py` code). One argument, `minmax`, specifies an interval in which the parameter value must lie. The argument `widget` specifies the type of graphical element (widget) that is used in a graphical user interface to set values. In the present sample call to `DataItem`, we specify "slider", which leads to a slider that can be dragged to the right value. The slider is only meaningful if we have an associated keyword argument `minmax` with the limits of the slider. The default `widget` value is `entry`, which gives a simple one-line text entry where the user can write the parameter value. The following widget types are valid in DataPool:

- `entry` – a simple entry field element that allows the user to enter text
- `textbox` – an entry field element that allows the user to enter multi-line text
- `slider` – an element that allows the user to choose a value within a defined interval by moving an indicator
- `radiobutton` – an element that allows the user to choose one option from a predefined set of options
- `checkbox` – an element that allows the user to choose one or several option from a predefined set of options
- `optionlist` – a drop-down list that allows the user to choose one option from a predefined set of options
- `fileupload` – a dialog that allows the user to upload a file and save the path to the file in the data item as a value.

It is also worth noticing that the `minmax` attribute will always be ignored if the data item also has the `valuelist` attribute. When set or updated, the value will always be checked to satisfy rules defined by those two attributes.

Another argument, `str2type`, specifies a Python function that turns a string with the user's input into the right Python object. By default, `str2type` is `str`, but is replaced with a proper function when the data item is created. For example, a default value of `3.5` makes DataPool set `str2type` to `float`. The user can of course assign any function taking a string as argument and returning a

python object.

Giving `bool` as the `str2type` argument could seem natural if the value is supposed to be boolean, but the plain Python `bool` causes trouble. For example, `bool('False')` or `bool('false')` is `True` since `'False'` and `'false'` are non-empty strings and therefore evaluate to `True`. It would be convenient to write `false` as a data item value, so to obtain this functionality, the function `scitools.misc.str2bool` is used as the default choice of `str2type` for boolean values. This functions turns case insensitive strings "false", "true", "on", "off", "yes", and "no" to the right boolean value.

To retrieve values from the menu in a bulk call, we need a list of the data item names as before. Each `DataItem` instance has an attribute `name` holding the name. A list of item names is therefore easily constructed as

```
names = [item.name for item in data_items]
```

Other `DataItem` attributes have the same name as those that can be used for keyword arguments in the `DataItem` constructor: `default_value`, `help`, `unit`, `str2type`, `widget`, `minmax`.

The `name` and `default_value` are required attributes for any data item. The other attributes are optional, with `None` as default value.

In addition, the programmer can supply an unlimited set of other keyword arguments, each leading to an attribute with the same name. These attributes are called *meta attributes* and can be used for any purpose by the programmer. For example, one may use a meta attribute to attach an object for processing a parameter value, say convert a filename to an interpretation of the file contents. Meta attributes can also be used to add functionality missing in DataPool.

Imagine that we have a data item with name `data_file` that has a file name we want to process as a value. To extend DataPool with a function for processing this particular type of data files we define an attribute (say) `process`. It will contain a function that processes the file contents, and returns a desired result:

```
def do_stuff(filename):
    # Pseudo code:
    # 1. read the file
    # 2. process data
    # 3. create and return the data structure
    return data_structure

menu.add('data_file', 'mypath/datafile.ext', help='data file', \
         process=do_stuff)

# later, at some other place:
my_func = menu.get_attr('data_file', 'process')
value = my_func(menu.get('data_file'))
```

2.3.11 Traversing Menu Tree

DataPool's Python API also allows users to traverse the menu tree starting from a desired submenu:

```
menu.walk(submenu)
```

This is a generator function that for each data item (submenu) rooted at *submenu* (including *submenu* itself) returns a tuple consisting of a full path to each item in the tree structure, a list of child menu items (submenus) and a list of data items[6].

For instance, we can use this function to traverse the menu tree to generate documentation using a custom user-defined format, or search the menu tree for data items that have some particular attribute defined.

Let us look at a simple snippet that collects the path, name and value for each submenu under the starting submenu in a list. Later, this list can be processed to generate a LaTeX or HTML table that can be used as documentation:

```
item_list = []
for path, menu_items, data_items in self.walk(submenu):
    for item in data_items:
        item_list.append([path, item.name, item.attributes['value']])
```

2.3.12 Add a Menu with Minimally Intrusive Approaches

The above examples add quite some new code to the original program. Sometimes we do not want major modifications to a program that works. We simply want to add a menu with a minimum of new statements in existing program files. To this end, one can define data items in a separate file, say the file has the name `ball1_dataitems.py`:

```
data_items = [
    ('C_D', 0.2, 'drag coefficient'),
    ('density', 1.2, 'density of air', 'kg*m**(-3)'),
    ('radius', 0.11, 'radius of ball', 'm'),
    ('mass', 0.43, 'mass of ball', 'kg'),
    ('V high', 120, 'velocity of hard football kick', 'km/h'),
    ('V low',  10 , 'velocity of soft football kick', 'km/h'),
    ]
```

In the original code we now need to add the following set of quite generic statements to make a menu, ask the user for input, and load the user-given values into variables:

```
import DataPool
menu = DataPool.menu
from ball1_dataitems import data_items
menu.submenu('/Ball Simulation')
```

[6]You might have noticed resemblance between DataPool's `walk()` and `os.walk()`. The behavior was designed to be identical. The only difference is that the former traverses the menu tree, while the latter is written for directory trees.

```
menu.add(data_items)
menu.start_ui()
names = [item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)
```

The file `ball_import1.py` contains a complete program using this strategy.

The idea can be extended further to put *all* DataPool related code externally in a file:

```
import DataPool
menu = DataPool.menu

data_items = [
    ('C_D', 0.2, 'drag coefficient'),
    ('density', 1.2, 'density of air', 'kg*m**(-3)'),
    ('radius', 0.11, 'radius of ball', 'm'),
    ('mass', 0.43, 'mass of ball', 'kg'),
    ('V high', 120, 'velocity of hard football kick', 'km/h'),
    ('V low',  10 , 'velocity of soft football kick', 'km/h'),
    ]

menu.submenu('Ball Simulation')
menu.add(data_items)
menu.start_ui()
names = [item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)

# remove variables not necessary for the computational code:
del names, data_items
```

This file must be constructed as a module such that we can import it. During the import, the code will be executed. The idea is to do a

```
from module import *
```

type of import. We therefore clean up variables that are not useful in the application code doing the computations. All variables for data item values and the `menu` variable are useful to have in the present case (but not `names` and `data_items`).

The original `ball1.py` code now needs only *one* statement (!):

```
from ball1_menu import *
```

if the module with the menu functionality is called `ball1_menu.py`. This is the non-intrusive way of adding menu functionality to our original program. The modified `ball1.py` file is called `ball1_import.py`.

To summarize, DataPool makes it possible to take our original program with hardcoded parameters and equip it with fancy user interfaces by (i) adding only one statement and (ii) creating a new module with eight statements!

2.3.13 Data Items for Output Data

We can define a set of data items representing results of computations, i.e., output data from the program. In the present case, the drag and gravity forces

constitute output data items. By including them on the menu, we can in a graphical interface view their values after the computations are performed. In this way, DataPool can be used for both input and output in a program, and in particular, graphical interfaces may become a more problem solving environment than just a way of setting input.

This is described in details in section 3.7 in the chapter *Using DataPool Web: User Manual*.

2.4 A Class-Based Example

Our next example is a refined version of the simple program `ball1.py` from Section 2.3. Instead of a "flat" program, we create a class for holding the physical parameters and computing the drag and gravity force. The complete code is found in the file `ball2.py`:

```
"""
Class version of the code in ball1.py.
"""
from math import pi

class BallForces:
    """
    Compute gravity and drag force on a ball moving in a fluid.
    m: mass in kg
    a: radius of ball in m
    C_D: drag coefficient (dimensionless)
    rho: density of surrounding fluid in kg/m**3 (1.2 for air)
    V: velocity of ball in km/h
    """
    def __init__(self, m, a, C_D=0.2, rho=1.2, V=1):
        self.m = m
        self.a = a
        self.C_D = C_D
        self.rho = rho
        self.set_velocity(V)
        self.A = pi*a**2
        self.g = 9.81

    def set_velocity(self, V):
        """Set velocity V in km/h."""
        self.V = V/3.6  # convert to m/s

    def gravity(self):
        """Compute, store and return gravity force."""
        self.gravity_force = self.m*self.g
        return self.gravity_force

    def drag(self):
        """Compute, store and return drag force."""
        self.drag_force = 0.5*self.C_D*self.rho*self.A*self.V**2
        return self.drag_force

    def compute(self):
        """Compute drag and gravity force and their ratio."""
        self.gravity()
        self.drag()
        self.force_ratio = self.drag_force/self.gravity_force

    def __str__(self):
```

```
        """Print data and the gravity and drag forces."""
        s = repr(self) + '\n' # print out input parameters
        try:
            s += 'Gravity force: %.2f N\nDrag force: %.2f N\n'\
                 'ratio (drag to gravity): %.4f' % \
                 (self.gravity_force, self.drag_force,
                  self.force_ratio)
            return s
        except AttributeError:
            raise Exception, \
            'You must call compute before you can print results'


    def __repr__(self):
        """Return string s such that eval(s) recreates the instance."""
        V = 3.6*self.V   # convert m/s to km/h
        return 'BallForces(m=%s, a=%s, C_D=%s, rho=%s, V=%s)' % \
               (self.m, self.a, self.C_D, self.rho, V)

if __name__ == '__main__':
    m = 0.43
    a = 0.11
    hard = BallForces(m, a, V=120)
    soft = BallForces(m, a, V=10)
    hard.compute()
    soft.compute()
    print 'Hard kick of a football:\n', hard
    print 'Soft kick of a football:\n', soft
```

2.4.1 Adding a Menu to a Class

Our first attempt to add a DataPool-based menu to our `BallForces` class will be based on three modifications:

1. Allow the constructor to take no arguments (no input). This is necessary if we want to create empty instances that are later filled with data from the menu.
2. Introduce a method `define` for defining all data items in the class. We put the data items in a submenu with the same name as the class.
3. Introduce a method `scan` for retrieving user-given data from the menu and loading the data into class attributes.

There are two basic strategies for defining and reading data items: we can do separate calls for each parameter or we can collect parameter information in lists and do a bulk call. In a class, the latter approach may be the simplest one since the list of data items is handy to have as a static variable in the class. This list and the modified constructor look as follows:

```
import DataPool
menu = DataPool.menu

class BallForces:
    """
    Compute gravity and drag force on a ball moving in a fluid.
```

```
    m: mass in kg
    a: radius of ball in m
    C_D: drag coefficient (dimensionless)
    rho: density of surrounding fluid in kg/m**3 (1.2 for air)
    V: velocity of ball in km/h
    """

    data_items = [
    ('C_D', 0.2, 'drag coefficient'),
    ('density', 1.2, 'density of air', 'kg*m**(-3)'),
    ('radius', 0.11, 'radius of ball', 'm'),
    ('mass', 0.43, 'mass of ball', 'kg'),
    ('velocity', 120, 'velocity of a football kick', 'km/h'),
    ]

    def __init__(self, m=0.5, a=0.1, C_D=0.2, rho=1.2, V=1):
        self.m = m
        self.a = a
        self.C_D = C_D
        self.rho = rho
        self.set_velocity(V)
        self.A = pi*a**2
        self.g = 9.81
```

Note that we now define a menu without a low and high velocity. Instead we create all parameters for one ball in one submenu. If we want to compare the drag force for a soft and a hard kick, we create two submenus, one for each ball, and fill in velocity values in both submenus. The menu becomes less tailored, but the class with its own menu is a more reusable piece of code for other applications.

The definition of items is now just a short method, and the `scan` method should also be straightforward to understand:

```
    def define(self, submenu_name=None):
        if submenu_name is None:
            submenu_name = self.__class__.__name__
        self.submenu_name = "/"+submenu_name
        menu.submenu(self.submenu_name)
        menu.add(BallForces.data_items)

    def scan(self):
        menu.submenu(self.submenu_name)
        names = [item[0] for item in BallForces.data_items]
        self.C_D, self.rho, self.a, self.m, self.V = menu.get(names)
```

The main program is affected by the presence of a menu as we need to call the `define` and `scan` methods, plus `menu.start_ui`. We put all the code in a function `main`:

```
def main():
    hard = BallForces()
    soft = BallForces()
    hard.define('Hard Kick')
    soft.define('Soft Kick')
    menu.set("/Hard Kick/velocity", 100)
    menu.set("/Soft Kick/velocity", 8)
    menu.start_ui()
    hard.scan()
    soft.scan()
    hard.compute()
```

```
    soft.compute()
    print 'Hard kick of a football:\n', hard
    print 'Soft kick of a football:\n', soft
if __name__ == '__main__':
    main()
```

The complete code is found in the file `ball2_bulk.py`. Let us try out the program with various types of user interfaces. The goal is to set values for a hard and soft kick of a football with velocities 100 km/h and 8 km/h, respectively. All default values, except for the velocities, are then correct. Note that we have two `velocity` items, in separate submenus. We therefore need to use the full path to reach a particular velocity item: `Soft Kick/velocity` and `Hard Kick/velocity`.

We start with the command line interface:

Terminal

```
python ball2_bulk.py --/Soft_Kick/velocity 8 \
                     --/Hard_Kick/velocity 100
```

A file interface requires us to make a file, say `data.i`,

```
submenu /Soft Kick
velocity = 8 km/h
submenu /Hard Kick
velocity = 100 km/h
```

The file interface is required by running

Terminal

```
python ball2_bulk.py --datapool-ui data.i
```

2.4.2 Add a Menu with a Non-Intrusive Approach

The next step is to modify the code in the previous section such that we minimize the number of modifications in the original `BallForces` class. We follow the ideas from Section 2.3.12. In the present case, it is wise to leave `ball2.py` as it is since the module may be used in other applications. The additional menu-related code is added in a separate, second file. A third file can import this second file and perform an execution, or the execution can be realized as a test block in the second (module) file.

We remark that the code we end up with in this example requires quite some advanced Python constructs, but the example shows the power of modifying code in one file, without touching the text in this file, but instead adding new code in a separate file.

The file with menu-related code is given the name `ball2_menu1.py`. It first contains the list of data item information, either as a plain list of tuples, or a list of dictionaries, or a list of `DataItem` instances. The two latter approaches

are needed if we want to set some of the more advanced data item attributes such as `str2type`. Here we exemplify the use of a list of dictionaries[7]:

```
import DataPool
menu = DataPool.menu

data_items = [
    dict(name='C_D', default=0.2, help='drag coefficient'),
    dict(name='density', default=1.2,
         help='density of air', unit='kg*m**(-3)'),
    dict(name='radius', default=0.11,
         help='radius of ball', unit='m'),
    dict(name='mass', default=0.43,
         help='mass of ball', unit='kg'),
    dict(name='velocity', default=120,
         help='velocity of a football kick', unit='km/h'),
    ]

from ball2 import BallForces
BallForces.data_items = data_items
```

The next step is to add a `define` and a `scan` method to the original `BallForces` class in the file `ball2.py` *without* editing the `ball2.py` file. This can be done by defining new methods as functions and then attaching them to the class object. The following code in `ball2_menu1.py` does the job:

```
def define(self, submenu_name=None):
    if submenu_name is None:
        submenu_name = self.__class__.__name__
    self.submenu_name = "/"+submenu_name
    menu.submenu(self.submenu_name)
    menu.add(BallForces.data_items)

def scan(self):
    menu.submenu(self.submenu_name)
    names = [item['name'] for item in BallForces.data_items]
    self.C_D, self.rho, self.a, self.m, self.V = menu.get(names)
    self.A = pi*self.a**2
    self.g = 9.81

from ball2 import BallForces
from scitools.misc import func_to_method
func_to_method(define, BallForces)
func_to_method(scan, BallForces)
```

We also want to replace the constructor in class `BallForces` by a constructor that does not need any input data, because we provide the input data elsewhere and call `scan` to initialize attributes. Contrary to the previous example where we just added default values to positional arguments in the constructor and initialized all input data attributes, we now simply provide an empty constructor:

```
def dummy_constructor(self):
    pass

func_to_method(dummy_constructor, BallForces, '__init__')
```

Now, we basically have the same modified class `BallForces` as in the example file `ball2_bulk.py`. It remains to write a `main` function that does the

[7]Every dictionary is just passed as argument to the `DataItem` constructor, so the keys in the dictionaries correspond to the keyword arguments in the `DataItem` constructor.

same main steps as the `main` in `ball2_bulk.py`. We could import that function from `ball2_bulk.py`, but since this example is meant to be separate from and alternative to the code in `ball2_bulk.py`, we copy the function. The code can be found in `ball2_menu1.py`.

2.4.3 Another Non-Intrusive Approach

The previous example let a `BallForces` object build a submenu, and then we constructed a global menu from two such submenus. This principle is useful in large program systems. An alternative way, which is closer to the starting example in `ball1_bulk.py`, is to separate the menu and the original class code completely. We build a separate menu, prompt the user, and then we load data from the menu and create the relevant objects of type `BallForces` and use these as in the initial example `ball2.py`. In a way, this is a non-intrusive example combining the menu in `ball1_bulk.py` with the class definition in `ball2.py`. The Python code is simpler than in the previous example and should speak for itself without further explanation.

```
import DataPool
menu = DataPool.menu

data_items = [
    dict(name='C_D', default=0.2, help='drag coefficient'),
    dict(name='density', default=1.2,
         help='density of air', unit='kg*m**(-3)'),
    dict(name='radius', default=0.11,
         help='radius of ball', unit='m'),
    dict(name='mass', default=0.43,
         help='mass of ball', unit='kg'),
    dict(name='V high', default=120,
         help='velocity of a hard football kick', unit='km/h'),
    dict(name='V low', default=10,
         help='velocity of a hard football kick', unit='km/h'),
    ]

def scan():
    """Load menu data into hard and soft objects (BallForces)."""
    names = [item['name'] for item in data_items]
    C_D, rho, a, m, V_hi, V_lo = menu.get(names)
    from ball2 import BallForces
    hard = BallForces(m, a, C_D, rho, V_hi)
    soft = BallForces(m, a, C_D, rho, V_lo)
    return hard, soft


def main():
    menu.add(data_items)
    hard, soft = scan()
    hard.compute()
    soft.compute()
    print 'Hard kick of a football:\n', hard
    print 'Soft kick of a football:\n', soft

if __name__ == '__main__':
    main()
```

2.5 A Complete *ball* Demo

Now that we have seen different functions of DataPool applied to a simple program that computes drag and gravity forces in the case of a soft and hard kick of a football, we will try to sum up some of the central DataPool functions in this demo walkthrough.

In this demo, we will start our "simulation" program, define a desired user interface, and update the velocity for a soft kick of the football from the command line. Afterwards, we will review and update some of the values in the web interface. At the end, we will proceed with the simulation code, and save our new parameter data to a file in DataPool's "shell" format.

Before we start our demo, we have to copy and modify the file

`samples/ball/ball2_menu1.py`

Copy the file to another folder, and rename it `ball2_menu1_demo.py`. Then, open it in your favorite text editor, and at the end of the `main` function add the `menu.write()` with correct parameters. At the end, the `main` function should look like this:

```
def main():
    hard = BallForces()
    soft = BallForces()
    hard.define('Hard Kick')
    soft.define('Soft Kick')
    menu.set('velocity', 10)
    menu.start_ui()
    hard.scan()
    soft.scan()
    hard.compute()
    soft.compute()
    print 'Hard kick of a football:\n', hard
    print 'Soft kick of a football:\n', soft
    menu.write("<your/path>/ball2\_menu1_demo.py", "shell")
```

Now, let us start our demo!

To update the velocity value for a soft kick we have to use the correct command line option with the full path to the data item[8]. If we are not sure about the path to this parameter we may issue the following command

Terminal

```
python ball2_menu1.py --help-all
Usage:

scriptname.py [--data_item_name1 value1 ...] [--datapool-ui] [--datapool-file]

datapool-file -- is the path to a file with DataPool commands
datapool-ui -- is the desired UI.

List of the available data items:
--/Hard_Kick/C_D
```

[8]We have to use full paths since we have two different velocities: one for soft kick, and another for hard kick.

```
--/Hard_Kick/density
--/Hard_Kick/radius
--/Hard_Kick/mass
--/Hard_Kick/velocity
--/Soft_Kick/C_D
--/Soft_Kick/density
--/Soft_Kick/radius
--/Soft_Kick/mass
--/Soft_Kick/velocity
```

This will show us a list of all available parameters. Now, we know that we should update `--/Soft_Kick/velocity`:

Terminal

```
python ball2_menu1.py --datapool-ui web --/Soft_Kick/velocity 25
```

A second later, we will be presented with a web interface as seen in Figure 2.3. Here we can observe that the velocity has been updated with the new value (as we can see from the code, it was previously set to 10 in the `main()`).

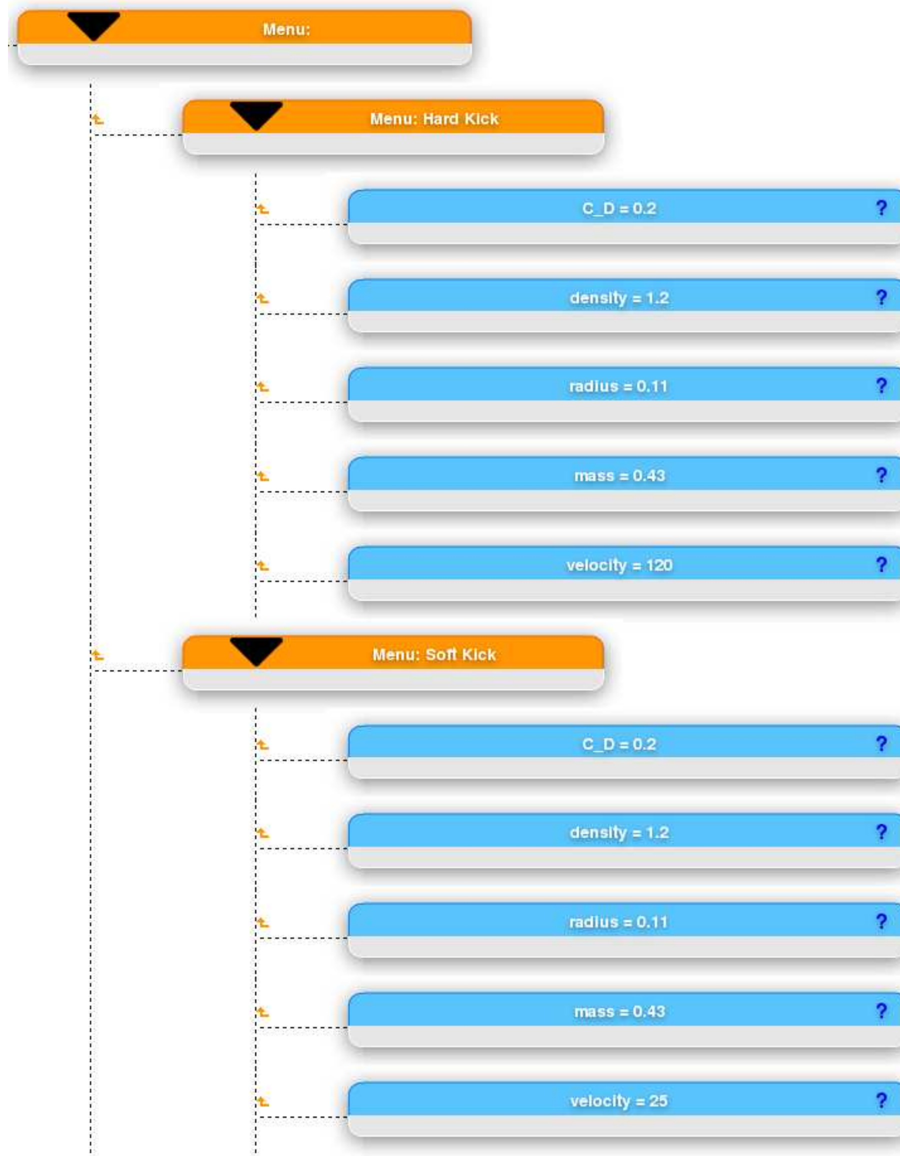

Figure 2.3: Web interface for ball2_menu1.py

Also, we can review and update some of the parameters, for instance, we can update the mass for both kicks to 0.7. Now, when we are satisfied with all values, we can start the simulation. When the simulation is done, proceeding with the `ball2_menu1_demo.py` script, and save the values with their units (if any) to the file in DataPool "shell" format.

Here we have seen how we can easily modify parameters from both the command line and the graphical user interface, save the new values to a file, and run the simulation. The new values can later be loaded into DataPool via its file interface as described in Section 2.3.6. Note that all interfaces were

generated automatically from the submenu and data item definitions in the script.

2.6 A More Advanced Example

Our next example is more comprehensive as it needs a real tree of submenus and performs a real simulation.

2.6.1 The Problem

The purpose is to solve the following differential equation describing the motion of oscillating systems:

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad t > 0, \quad u(0) = U_0,\ \dot{u}(0) = V_0$$

This equation is nothing but Newton's 2nd law of motion, where m is the mass, u is a displacement of a body, $\dot{u}$ is the body's velocity, $\ddot{u}$ its acceleration, U_0 is the initial displacement, and V_0 the initial velocity. The terms f, s, and F are forces acting on the body.

Different physical problems lead to different choices of the friction force term $f(\dot{u})$, the spring (restoring) force term $s(u)$, and the external force term $F(t)$. Some common choices are listed below.

1. Linear friction force (low velocities): $f(\dot{u}) = 6\pi\mu R\dot{u}$ (Stokes drag), where R is the radius of a spherical approximation to the body's geometry, and μ is the viscosity of the surrounding fluid.
2. Quadratic friction force (high velocities): $f(\dot{u}) = \frac{1}{2}C_D\varrho A|\dot{u}|\dot{u}$, where C_D is a drag coefficient, A is the area of a cross section of the body normal to the motion, and ϱ is the density of the surrounding fluid.
3. Linear spring force: $s(u) = ku$, where k is a constant.
4. Sinusoidal "spring" force (pendulum): $s(u) = k\sin u$, where k is a constant.
5. Cubic spring force: $s(u) = k(u - \frac{1}{6}u^3)$, where k is a spring constant.
6. Sinusoidal external force: $F(t) = F_0 + A\sin\omega t$, where F_0 is the mean value of the force, A is the amplitude, and ω is the frequency.
7. "Bump" force: $F(t) = H(t-t_1)(1-H(t-t_2)F_0$, where $H(t)$ is the Heaviside function (0 if $t < 1$, otherwise 1), t_1 and t_2 are two given time points, and F_0 is the size of the force. This $F(t)$ is zero for $t < t_1$ and $t > t_2$, and F_0 for $t \in [t_1, t_2]$.
8. Random force 1: $F(t) = F_0 + A\cdot U(t;B)$, where F_0 and A are constants, and $U(t;B)$ denotes a function whose value at time t is random and uniformly distributed in the interval $[-B, B]$.
9. Random force 2: $F(t) = F_0 + A\cdot N(t;\mu,\sigma)$, where F_0 and A are constants, and $N(t;\mu,\sigma)$ denotes a function whose value at time t is a random, Gaussian distributed number with mean μ and standard deviation σ.

A program for simulating the system described by the shown differential equation must be able to deal with all these choices of input, plus input parameters related to the mass parameter m, the type of differential equation solver, the solver's time step (Δt), and the end time of the simulation (T). We will now show how the concept of a menu system can be used to quickly define all information needed to create a menu tree which can be displayed as a web interface or operated on the command line or in a file.

2.6.2 The Existing Simulation Code

We assume that there exists some kind of a simulator with solver classes and classes for the various force models, etc. The application code will put instances of these classes together for solving a particular problem.

We have made a minimalistic general-purpose framework consisting of two modules: `functions` and `oscillator`. The `functions` module contains classes for the different force models above:

```
from math import *
import random

class Zero:
    def __call__(self, x):
        return 0.0


class LinearFriction:  # Stokes drag
    def __init__(self, mu, R):
        self.mu, self.R = mu, R

    def __call__(self, dudt):
        return 6*pi*self.mu*self.R*dudt


class QuadraticFriction:
    def __init__(self, C_D, rho, A):
        self.C_D, self.rho, self.A = C_D, rho, A

    def __call__(self, dudt):
        return 0.5*self.C_D*self.rho*self.A*abs(dudt)*dudt


class Spring:
    """Base class for all springs (holds the spring constant k)."""
    def __init__(self, k):
        self.k = k

class LinearSpring(Spring):
    def __call__(self, u):
        return self.k*u

class CubicSpring(Spring):
    def __call__(self, u):
        return self.k*(u - 1./6*u**3)

class SineSpring(Spring):
    def __call__(self, u):
        return self.k*sin(u)
```

```
class SineForce:
    def __init__(self, F0, A, omega):
        self.F0, self.A, self.omega = F0, A, omega

    def __call__(self, t):
        return self.F0 + self.A*sin(self.omega*t)


class BumpForce:
    def __init__(self, F0, t1, t2):
        self.F0, self.t1, self.t2 = F0, t1, t2

    def __call__(self, t):
        if self.t1 <= t <= self.t2:
            return self.F0
        else:
            return 0.0

class RandomForce1:
    def __init__(self, F0, A, B):
        self.F0, self.A, self.B = F0, A, B

    def __call__(self, t):
        return self.F0 + self.A*random.uniform(-self.B, self.B)


class RandomForce2:
    def __init__(self, F0, A, mu, sigma):
        self.F0, self.A, self.mu, self.sigma = F0, A, mu, sigma

    def __call__(self, t):
        return self.F0 + self.A*random.gauss(self.mu, self.sigma)
```

The `oscillator` module contains a `Problem` class and a `Solver` class. The class `Problem` holds all information about the physical problem to be solved, but it knows nothing about *how* the problem can be technically solved. The `rhs` method applies the information about the problem and defines the right-hand side of a system of ordinary differential equations. This is the only information about the problem that is needed for a solver.

The class `Solver` knows how to solve the problem, but it knows nothing about the problem itself, it only has access to a function (in our case `Problem.rhs`) that can define the system of ordinary differential equations to be solved. This design gives a clear distinction between the particular physical problem and the general mathematical methods used for solving equations.

An implementation of classes `Problem` and `Solver` may read:

```
from ODESolver import *
from functions import *
from scitools.all import *
from scitools.misc import read_cml, read_cml_func

class Problem:
    def initialize(self):
        """Read option-value pairs from sys.argv."""
        self.m = eval(read_cml('--m', 1.0))
        self.friction = read_cml_func(
            '--friction', lambda dudt: 0, 'dudt', globals())
        self.spring = read_cml_func(
            '--spring', lambda u: u, 'u', globals())
```

```
        self.external = read_cml_func(
            '--external', lambda t: 0, 't', globals())
        self.initial_u = eval(read_cml('--initial_u', 1.0))
        self.initial_dudt = eval(read_cml('--initial_dtdu', 0))

    def rhs(self, u, t):
        """Define the right-hand side in the ODE system."""
        m, f, s, F = \
           self.m, self.friction, self.spring, self.external
        u, dudt = u
        return [dudt,
                (1./m)*(F(t) - f(dudt) - s(u))]

class Solver:
    def initialize(self):
        self.T = eval(read_cml('--T', 4*pi))
        self.dt = eval(read_cml('--dt', pi/20))
        self.method = read_cml('--method', 'RungeKutta4')

    def solve(self, problem):
        self.solver = eval(self.method)(problem.rhs, self.dt)
        self.N = int(self.T/self.dt)
        ic = [problem.initial_u, problem.initial_dudt]
        self.solver.set_initial_condition(ic)
        self.u, self.t = self.solver.solve(self.N)
```

Both classes have an `initialize` function whose purpose is to initialize attributes by reading information from the command line. The `read_cml` and `read_cml_func` functions are utilities for reading command-line arguments in a flexible way. Typically, we can specify the `--spring` option as, for example, `CubicSpring(2.5)` (i.e., we make an instance of `CubicSpring` and this can be called as a function since the instance has a `__call__` special method). The details of the command line reading utilities are not important here – this code is just from a first version of the simulator and we want to replace this code by constructions based on DataPool.

We may also want to visualize the $u(t)$ and $u'(t)$ curves. This can be done by a `Visualizer` class, which gets the solution from class `Solver` and that can tag plots by physical parameters from class `Problem` (the shown code is a simplified version of what is found in `oscillator.py`):

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem = problem
        self.solver = solver

    def visualize(self):
        u, t = self.solver.u, self.solver.t   # short forms
        # tag all plots with numerical and physical input values:
        title = 'solver=%s, dt=%g, m=%g' % \
                (self.solver.method, self.solver.dt, self.problem.m)
        plot(t, u[:,0], 'r-',
             legend='%s, dt=%g' % \
             (self.solver.method, self.solver.dt),
             title='Plot of u: ' + title)
        if self.problem.u_exact is not None:
            hold('on')
            plot(t, self.problem.u_exact(t), 'b-',
                 legend='exact solution')
        show()
        hardcopy('tmp_u.eps')
```

A possible `main` routine goes as follows:

```
def main():
    problem = Problem()
    problem.initialize()
    solver = Solver()
    solver.initialize()
    visualizer = Visualizer(problem, solver)

    solver.solve(problem)
    visualizer.visualize()
```

Defining a physical problem (m, $f(\dot{u})$, $s(u)$, $F(t)$, $u(0)$, $u'(0)$) or specifying a solver (T, Δt, method) is supposed to be done on the command line in the code above, using the command-line options specified in the `initialize` functions. Here is an example of what we can run with this simulator, found in the file `oscillator.py`:

Terminal

```
python oscillator.py --method RungeKutta4 \
       --friction "LinearFriction(1/(6*pi), 0.1)" \
       --external "SineForce(0, 1, 0.5)" --dt "pi/80" \
       --T "40*pi" --m 10
```

2.6.3 The Menu Tree

The above mentioned `oscillator` module had to get all its input data from the command line. We shall now address how we can extend the `oscillator` and `functions` modules, without touching their source codes, so that all input parameters can be specified on a menu. The input will then be easier to understand because we avoid requiring constructions like

```
LinearFriction(mu=0.1, R=0.5)
```

Instead we let the user pick a linear friction model and set the two parameters to the constructor, μ and R, separately.

In the present application, it is natural to group input data into a menu with submenus. The physical and numerical parameters are two obvious candidates for submenus. Also, if we want to control what kind of plots the class `Visualizer` should make, we can have a third submenu for this, although we postpone this possibility right now.

The submenu for physical parameters should contain m and the force model. Since each force model involves several physical parameters, it is natural to include new submenus for the different types of forces, and for each of these, submenus for the different types of force models. To summarize, the menu is organized as a tree, where the indentation for each line below illustrates the submenu level:

Physical parameters:
  m
  friction force $f(\dot{u})$:

- model
- linear:
  - μ
  - R
- quadratic:
  - C_D
  - ρ
  - A

spring force $s(u)$:
- k
- model

external force $F(t)$:
- model
- sine:
  - F_0
  - A
  - ω
- bump:
  - F_0
  - t_1
  - t_2
- random1:
  - F_0
  - A
  - B
- random2:
  - F_0
  - A
  - μ
  - σ

Solver parameters:
- T
- Δt
- method

Note that several parameters have the same name, e.g., F_0. However, the paths in the menu are different: Physical parameters/external force $F(t)$/bump/F_0 versus Physical parameters/external force $F(t)$/sine/F_0. Sometimes the name of a parameter is unique, and sometimes one needs (at least a part of) the path to uniquely identify the parameter, as previously mentioned.

2.6.4 Creating a Menu

We shall now realize the menu tree from the previous section by means of the DataPool package. The implementation should be as non-intrusive as possible so we avoid modifying the original code. That is, we shall add all the menu functionality *outside* the module files `oscillator.py` and `function.py` displayed above.

The technique consists in making a module `functions_menu`, where we de-

fine menu items and load data given by the user into data structures from the `functions` module. Consider the class

```
class QuadraticFriction:
    def __init__(self, C_D, rho, A):
        self.C_D, self.rho, self.A = C_D, rho, A

    def __call__(self, dudt):
        return 0.5*self.C_D*self.rho*self.A*abs(dudt)*dudt
```

This class needs three parameters to be initialized, and we want these to be specified by the user through a menu. In the `functions_menu` module we list the parameters by name, description, and default values and attach this list as a static attribute to the class `QuadraticFriction`:

```
from functions import QuadraticFriction

QuadraticFriction.__data_items__ = [
    ("C_D", "drag coefficient", 0.2),
    ("rho", "density of air", 1.2),
    ("A", "cross section area, normal to the flow", pi)]
```

There is no strict need to add the list to the class, the list could well be a stand-alone global list, but many programmers will find it natural to let it be a part of an extension of the `QuadraticFriction` class in the extended module `functions_menu`.

For the other classes in the `functions` module, we make similar lists. A function `friction_menu` can now add friction models and their parameters to the menu:

```
def friction_menu(submenu):
    menu.submenu(submenu)
    menu.add('model', help='zero, linear, quadratic',
             valuelist=('zero', 'linear', 'quadratic')
              widget='optionlist')
    menu.submenu(submenu + '/linear')
    menu.add(LinearFriction.__data_items__)
    menu.submenu(submenu + '/quadratic')
    menu.add(QuadraticFriction.__data_items__)
```

Similar functions are created for the spring and external force models, called `spring_menu` and `external_menu`. (Of course, we could drop the list and manually call `menu.add` for all parameters in each relevant class inside the `friction_menu` function, however there will be more manual work and longer code.)

Some of the data items will be the name of a class, for instance a force model class. Based on this name, we need to create the right instance and then extract more information from the menu to initialize the instance. All these actions are carried out in *factory functions*. The factory function for the friction model reads

```
def friction_factory(submenu):
    menu.submenu(submenu)
    model = menu.get('model')

    # get arguments for initializing the friction model:
    if model == 'linear':
        menu.submenu(submenu + '/' + model)
        mu, R = menu.get(['mu', 'R'])
        obj = LinearFriction(mu, R)
    elif model == 'quadratic':
        menu.submenu(submenu + '/' + model)
        C_D, rho, A = menu.get(['C_D', 'rho', 'A'])
        obj = QuadraticFriction(C_D, rho, A)
    elif model == 'zero':
        obj = Zero()
    else:
        raise ValueError, 'friction model name "%s" invalid' % model
    return obj
```

Similar factory functions, `spring_factory` and `external_factory`, are made for the two other force models.

With the `*_menu` and `*_factory` functions in the `functions_menu` module, we can easily populate a menu created in the `Problem` class with (a lot of) parameters for force models.

The menu made by the `Problem` class must also contain some other parameters. Typically, we would make a method like the following in class `Problem`:

```
def define(self, submenu='/physics'):
    # first define local parameters related to the physics:
    menu.submenu(submenu)
    menu.add('m', 1.0, help='mass')
    menu.add('initial u', 1.0, help="u(0) initial condition")
    menu.add('initial du_dt', 1.0, help="u'(0) initial condition")
    menu.add('u exact', None, str2type=str,
             help="analytical solution u(t)")
    menu.add('du_dt exact', None, str2type=str,
             help="analytical solution u'(t)")

    # module functions_menu allow flexible choice of force
    # models through submenus:
    friction_menu(submenu+'/friction')
    spring_menu(submenu+'/spring')
    external_menu(submenu+'/external')
```

Observe how easily we call external functionality in the `functions_menu` module to make other submenus. That is, the menu tree built in class `Problem` is recursively defined.

The same idea applies to retrieving data from the menu. After the user is prompted (graphics displayed, for example) and input data provided, a method in class `Problem` must load data into attributes in that class:

```
def scan(self, submenu='/physics'):
    menu.submenu(submenu)
    self.m = menu.get('m')
    self.initial_u = menu.get('initial u')
    self.u_exact,menu.get('u exact')
```

```
    if self.u_exact is not None:
        self.u_exact = StringFunction(self.u_exact,
                                      independent_variable='t',
                                      globals=globals())
    ...

    # these will be instances from the functions module:
    self.friction = friction_factory(submenu + '/friction')
    self.spring = spring_factory(submenu + '/spring')
    self.external = external_factory(submenu + '/external')
```

Again, the initialization of data is performed recursively. The `StringFunction` tools from the module `scitools.StringFunction` turns a string with a mathematical formula into a callable Python function (as if the string expression had been hardcoded in the function).

The `define` and `scan` methods are defined as ordinary functions in the module `oscillator_menu`. Then we simply attach these functions as methods in the `Problem` class. This can be done in the `oscillator_menu` module (using a well-known recipe from the Python Cookbook [19]):

```
from scitools.misc import func_to_method
func_to_method(define_menu, Problem)
func_to_method(scan_menu, Problem)
```

Similar define and scan methods are made for the `Solver` and `Visualizer` classes as well, and we refer to the `oscillator_menu.py` file for details.

A new `main` method is needed, where we must define the menus, start the interface, and call all the scan operations, before the classes are ready for computations:

```
def main():
    problem = Problem()
    problem.define()
    solver = Solver()
    solver.define()
    viz = Visualizer(problem, solver)
    viz.define()

    # force a web-based GUI:
    menu.prompt('web')

    # read input data:
    problem.scan()
    solver.scan()
    viz.scan()

    # simulate and visualize:
    solver.solve(problem)
    viz.visualize()
```

2.6.5 Examples on Using the Interfaces

The `oscillator_menu.py` program can now be run as a substitute for the original `oscillator.py` program, the difference being that the former has a menu for providing input.

Command Line. Let us first exemplify the command line interface.

Terminal

```
python oscillator_menu.py --/physics/m 2 \
       --/numerics/T 20*pi --/physics/friction/model linear \
       --/physics/friction/linear/mu 0.2/(6*pi) \
       --/physics/external/model sine --/physics/external/sine/omega 4
```

File. A file interface requires commands in a file, for example:

```
submenu /physics/
m = 2
submenu /numerics/
T = 20*pi
submenu /physics/friction/
model = linear
submenu /physics/friction/linear/
mu = 0.2/(6*pi)
submenu /physics/external/
model = sine
submenu /physics/external/sine/
omega = 4
```

We could also use the relative paths when moving between submenus:

```
submenu /physics/
m = 2
submenu /numerics/
T = 20*pi
submenu ../physics/friction/
model = linear
submenu linear/
mu = 0.2/(6*pi)
submenu /physics/external/
model = sine
submenu sine/
omega = 4
```

Web-Based GUI. The DPW user manual in chapter 3 will cover this since it uses the Oscillator simulation throughout the manual.

Required Software for This Example. The following files make up the software necessary for doing the run above:

```
functions.py
oscillator.py
functions_menu.py
oscillator_menu.py
```

In addition, one needs SciTools version 0.51. To see plots on the screen it is necessary to have, the Gnuplot program and the `Gnuplot.py` module file installed.

Chapter 3

Using DataPool Web: User Manual

3.1 Introduction

This chapter covers all the aspects of the DPW regarding to concepts, details and available functions that are provided through this system. The manual first presents the structure and concepts of the web module and then goes more in-depth on details on the different types of use in a real software simulation. The starting point for the manual is the Oscillator simulation, and all the belonging screenshots are from a real walkthrough and test of this simulation. For newcomers it is recommended to read the manual from start to end, but of course possible and also encourage, to use it as quick reference when needed.

3.2 Menu system

DPW is a web-based menu system designed to present the internal tree structure of the DataPool module in the most usable and effective way according to user interaction. An active session of DataPool could contain huge amounts of data which could be hard to present for the user in an effective and highly lucid manner. The goal of DPW is to provide a solution to this challenge. DPW focuses on user interaction, practical functions and visual communication in order to make the process of using the DataPool module easy and time saving in extensive and large computer simulations with a lot of parameters. Visual elements and functions are of real value when implemented with care. DPW is designed with this in mind, using the visual communication and usable functions in balance, not affecting the real user experience for the end-users with distracting and unnecessary elements.

The design and layout are based upon so called web 2.0 design elements which have dominated a lot of web services and internet sites the last few years. The main purpose for this approach is to keep the visual first impression fresh, clean and minimalistic, but also to obtain a modern environment when working with complex simulations. In most cases computer scientists are on daily basis dealing with terminal windows, hard coding and file dumping, and DPW

is created in this manner because scientists deserve to have both inspiring environments and efficiency in their work. The choice of colors and layouts are chosen to make the system distinct and easily recognizable as a certain system among the users, but also in different fields within computer science.

3.3 Fundamentals

3.3.1 Structure

The DPW menu structure is identical to the concept of an ordinary directory structure. *Submenus* (instances of `MenuItem`) are visualized with orange color, while the *parameters/data items* (instances of `DataItem`) are presented in light blue color. All belonging submenu children of a parent or parameters (data items), are indented one level to the right, making it similar like a directory structure. This is the most natural way to visualize an internal tree structure of parameters. On startup of a new session the root submenu is first shown in the upper-left corner:

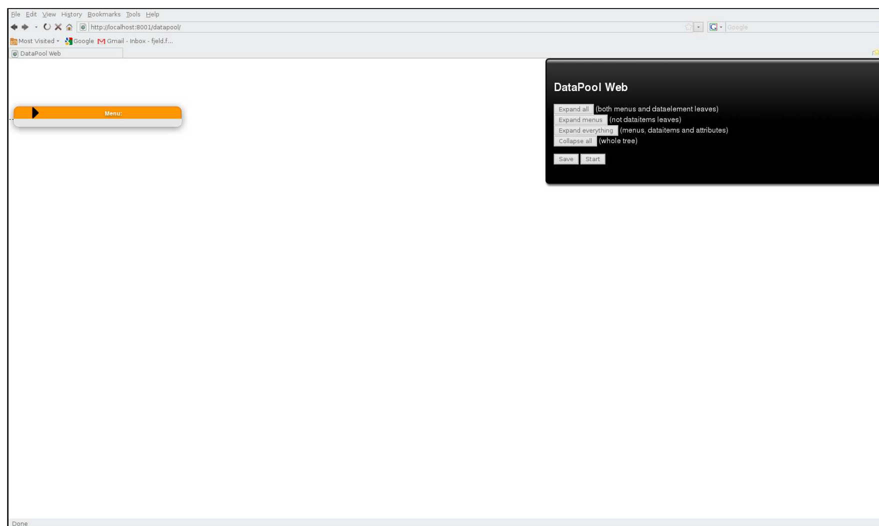

Figure 3.1: Initial start-up of DPW

This is the starting point and makes it possible to do a manual navigation through clicking, or take use of the administration panel for fast manipulation of the menu structure.

3.3.2 Administration Panel

The interface is equipped with an administration panel which is statically placed in the upper right corner. This panel is always available and visual *menu elements* (either data items or menu items) will be overlapped by it if the interface is handling a large and deep menu tree structure. The main idea of a static admin panel is to avoid tedious scrolling to either the top or bottom for doing common tasks on the menu tree.

3.3.3 Menu Items (submenus)

Submenus are colored orange and are used for grouping parameters with similarities to certain criterias, but can also contain other sub groups (submenus) to contain further more defined parameters for the simulation. DPW has no constraints on the level of deepness of the internal tree structure that is visualized in the interface. In most cases all submenus will fit to the web browser without need for scrolling horizontally. In theory, DPW will try to show an infinite levels of submenus, and in those cases the web browser will naturally activate horizontal scrolling. Potential errors will most probably occur because of memory overload or crash of the web browser due to other limitations outside of DataPool's scope.

3.3.3.1 Quick-links

Submenus provide a nifty feature for fast and easy navigation and automatic scrolling through the menu structure. This is one of the most time saving features offered by the web interface. *Quick-links* are available in the *Quick-menus* which are present on all submenus. Quick-menus have shortcuts for both data items and other children submenus.

3.3.3.2 Children & Data Items

Submenus have both other children (submenus) and belonging data items on same level. Belonging parameters to a specific submenu will always be listed before other submenu children. This is a choice of design to save time, since in the long run we will be looking more frequent for the parameters.

3.3.4 Data Items

A data item holds information about and visually represents a parameter. This includes the name of the parameter, value and other user defined attributes that are needed for the specific parameter. Data items are colored light blue, and the reason for giving parameters this color and not orange (as for submenus), is the majority of data items over submenus in a menu tree. Hence, light blue as the most frequent color gives a more comfortable visual expression. Orange and light blue can also be stated to be easy to distinguish from each other and clearly works fine together.

3.3.4.1 Header

The header of the data item is holding the light blue color, the name of the parameter and the current assigned value. A question mark is also provided and can be used to get more information about the purpose of the parameter. Absence of a question mark means there are no help information added by the user.

3.3.4.2 Fields & Attributes

The body of the data item element exists of fields for each of the belonging attributes, including the value itself. The number of fields will variate between

the data items in the menu structure depending on the provided data in the DataPool module. All changes of values will be performed here, including all field related error messages. The type of fields used for each value and/or attributes in the web interface depends on the Python data type of the attributes in the internal DataPool menu tree.

3.4 Concepts

3.4.1 Hovering

DPW is implemented with a visual effect when hovering over the menu elements. This reason is to keep focus when the menu tree becomes complex and to assure complete awareness of interaction.

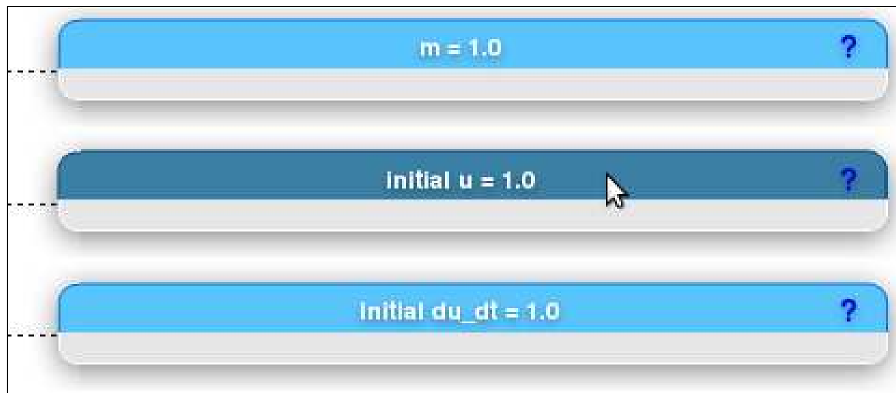

Figure 3.2: Hover effect on data items

Figure 3.3: Hover effect on a folded submenu

3.4.2 Expand & Collapse

The body of data items are initially hidden when first navigating a menu tree, with the reason of saving space and maintaining the focus. A normal approach is to click the hovered head of a data item to expand or collapse the content of the element body. DPW is designed in this way since it usually are only a moderate number of parameters and/or attributes that are up for change during a session, and it would make it distracting if each and all of the data item bodies were expanded.

Figure 3.4: A collapsed data item showing its belonging attributes

Submenus have an expanded/collapsed symbol to show whether or not the submenu is expanded. Since submenus have the feature of always slide down a Quick-menu automatically when hovering, this black direction symbol makes it easy to keep track of the state.

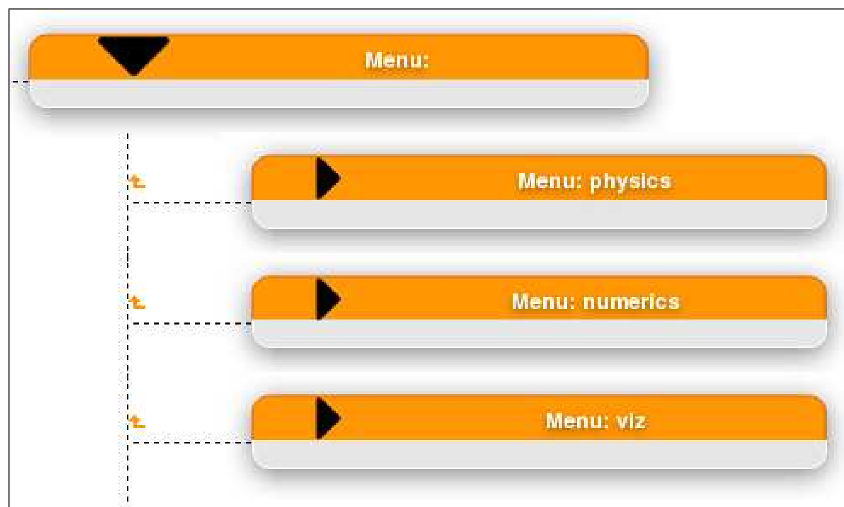

Figure 3.5: An expanded submenu showing its first level of children submenus

3.4.3 Linking & Scrolling

Potentially a DataPool tree structure can contain a significant number of menu elements at each level, leading to a very long menu tree structure in the web interface. This is a case leading to tedious and time consuming scrolling in the web browser for each data item that is up for change. Also number of levels (of submenus) can result in a wide set of menu elements filling the browser window

when expanded. From a user's perspective the simulation to be performed could be well-known, hence the scientist has already obtained pretty good overview of the menu structure. It could be frustrating for experienced users to scroll several pages when only a few and specific values need to be addressed between each run. It is of very high benefits for end users to avoid scrolling and fine-tuning in a complex structure if only few values are going to be changed on a regular basis in typical try-and-fail sessions. For this DPW provides different types of automatic scrolling on the generated menu structure.

3.4.3.1 Menu Item Quick-links

Each submenu will automatically slide down a Quick-menu when hovering over the submenu header. This handy extra menu shows all belonging submenus (menu items) and data items to the owner of the triggered Quick-menu. This feature makes it possible to get a fast hands-on preview of what lay further down on the next menu level without the need for a manual navigation to expand the submenu in order to get this information. In the list, the Quick-links for data items and menu items are presented with corresponding colors to match the right type, making it intuitive to do the appropriate click. A click on Quick-links will automatically start a smooth animated scrolling effect in the web browser to the chosen menu element, either another submenu or data item.

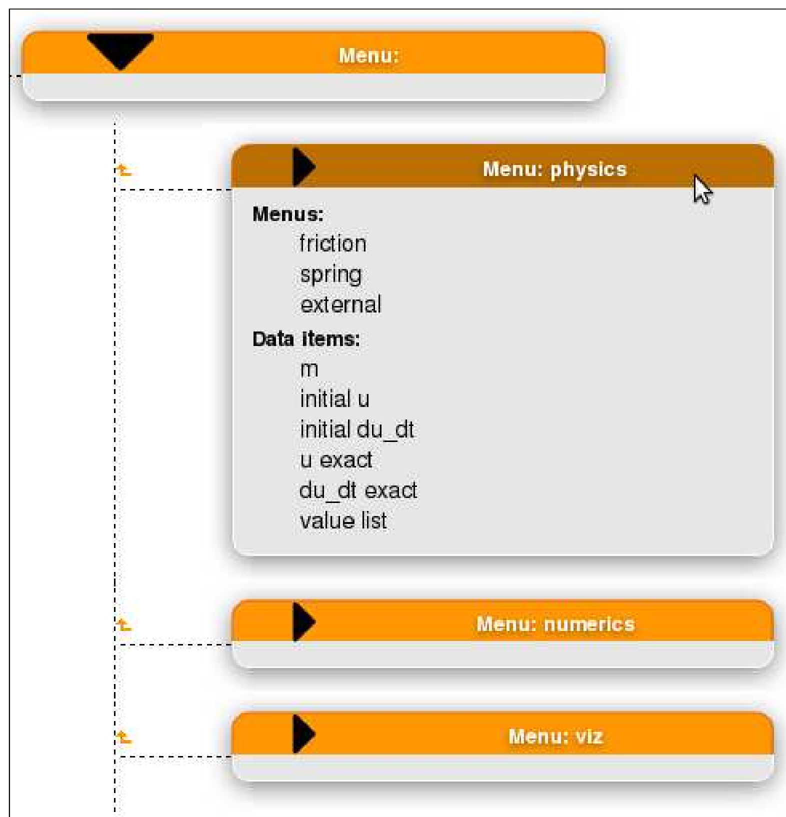

Figure 3.6: Automatically slided Quick-menu

3.4.3.2 Parent Links

A Quick-link will in most cases scroll and move the focus to a menu element placed longer down in the menu structure. In some situation it could be of great help to move the other way, vertically upwards in the interface. The structure can on certain levels be holding a huge set of data items and submenus. In some cases this will make it hard to keep track and remember the parent submenu of the visible menu elements because the parent menu will go out of sight. DPW provides a feature for automatically scrolling up one level to the owner of the menu elements if the user loses track. This is provided for both data items and submenus:

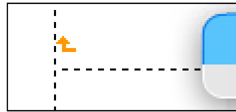

Figure 3.7: Parent link for a data item

3.4.3.3 Administration Panel Links

The administration panel will always be displaying status and error messages. The messages are presented with a fully clickable path to the affected menu elements. This a feature working as the two former linking concepts described, when clicked the interface will automatically animate and scroll to the chosen element. In this way it is easy to go through a list of errors and locate the affected ones to do preferred and needed changes without manual scrolling.

3.4.4 Menu Item Dependency

The DataPool API is equipped with the option to make dependency rules on the internal menu tree. These rules set limitations regarding the allowed numbers of submenus to be chosen and activated at a certain submenu level. This registration through the API is controlled by the scientists and the API provides different attributes in order to cover the most needs. DPW will generate the menu tree and attach the dependency information and denote where rules exist. This symbol indicates the existence of a dependency rule on the children submenus of the owner:

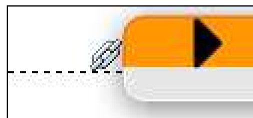

Figure 3.8: Anchor indicates dependency rule on a submenu level

3.4.5 Menu Chooser

The web interface will dedicate and visualize a data item to be the *Menu Chooser* at a submenu level with a dependency rule. A Menu Chooser is used to switch between the available children submenus and the Menu Chooser will show the possible choices according to the rule defined with the DataPool API.

3.4.5.1 Single

The system operates with two different types of dependencies, either single or multiple. The Menu Chooser will provide the correct type of input field in order to choose between the children submenus. If the defined dependency is allowing a maximum of one children submenu to be chosen, a select field is generated in the interface:

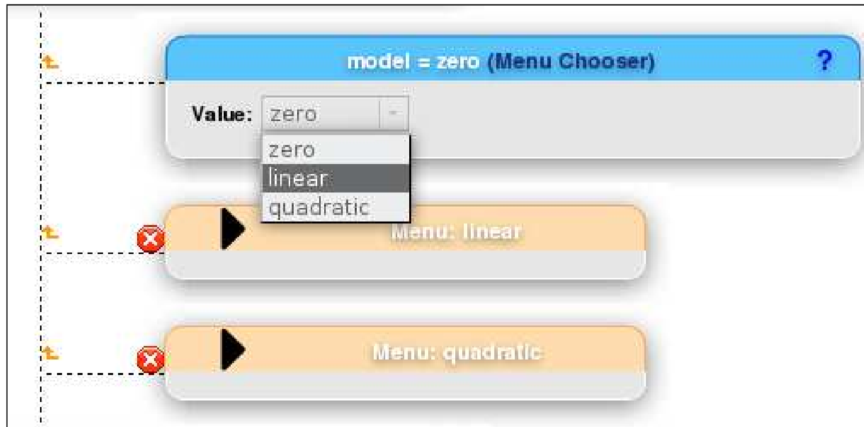

Figure 3.9: A Menu Chooser data item (single dependency)

3.4.5.2 Multiple

If the defined dependency is allowing more than one children submenus to be chosen, a SelectMultiple field is provided:

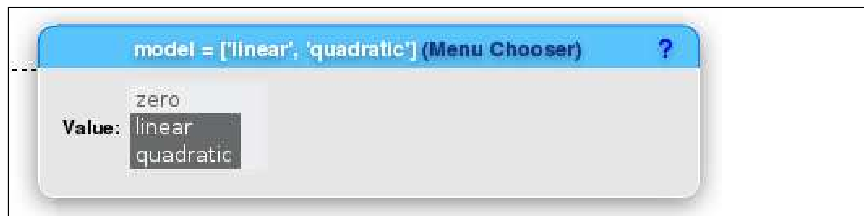

Figure 3.10: A Menu Chooser data item (multiple dependency)

3.4.6 Enabled & Disabled Items

When dependencies are involved in a menu structure there will most likely be several disabled menu elements present. The menu elements are disabled because they are not chosen by the Menu Chooser, meaning they will not be taken into account if the simulation were started. DPW will always visually show disabled menu elements with lighter colors and a red mark. This is a design decision made to abet focus and to assure that the users are changing the values and attributes at the right place. Disabled menu elements are not slidable and the color of submenus and data items will be faded accordingly:

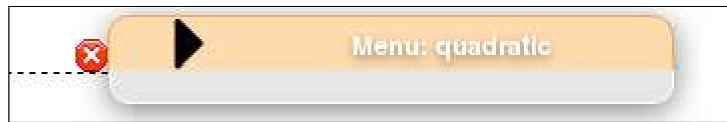

Figure 3.11: A disabled submenu

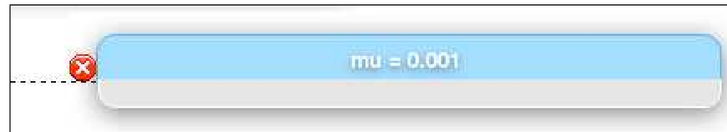

Figure 3.12: A disabled data item

3.5 Usage

3.5.1 Changing Data Item Values

Changing values of data items and belonging attributes are done in the body where the fields are located. How the value types are showed in the interface depends on how the values were assigned in the DataPool API. The concept is that the interface will follow the same syntax as when declaring Python values. This makes the interface consistent, pythonic and easy to relate. Using the same syntax also makes it easy to evaluate the input values in the core and produce eventual error messages. The next section shows how the different types are handled.

3.5.1.1 Numerics

Data items with a float value defined through the API will result in a field type that accepts floats. In contrast to a integer field type, the float field will also automatically convert a provided integer value to a float. In most cases the integer type field will only be used for a special reason and it is natural to allow the float field to accept both integers and floats and do the conversion for the user. Therefore, the integer field will only accept its own type.

Defining a data item with a float value:

```
Menu.add('m', 1.0, unit='kg', help='mass')
```

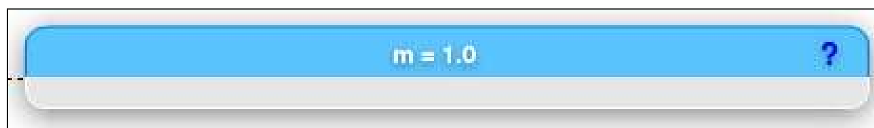

Figure 3.13: A float field

Defining a data item with an integer value:

```
Menu.add('tuple value', value=2)
```

Figure 3.14: An integer field

3.5.1.2 Strings

Fields of string type will allow a combination of characters and numbers treating it like a string, but also numbers alone. In the interface the string type field will accept any input, but will only treat it like a pure string. In the Oscillator simulation the solver method is defined as a string:

```
Menu.add(
        [dict(name='T', value=4*pi, help="stop time for simulation"),
         dict(name='dt', value=0.05, help="time step"),
         dict(name='method', value='RungeKutta4',
         help="ODE solver method (classname ForwardEuler, RungeKutta4)")
         ])
```

Figure 3.15: Representation of a Python string

3.5.1.3 Valuelists

It is possible to assign a python list to the attribute `valuelist`. This will create a drop-down field in the interface and is very convenient if only a special set of values are allowed. When setting the value in the code below it must be one of the values provided in the python list, or an error message will be shown from DataPool in the command line. The interface also know how to include both numbers and strings in the web interface in order to convert and save them with the correct type.

```
Menu.add('value list', 2,  valuelist=['one', 2, 'three'])
```

Figure 3.16: A drop-down from the valuelist attribute

3.5.1.4 Lists (Python)

If the value provided in a data item is a Python list, the interface will show it like a string. However, the field will be dedicated for a list and the result is a field only treating the input as a declaration of a real Python list. When saving, the system will keep track of mixed elements of numericals and strings in the representation of the list declaration and convert it to an ordinary Python list.

```
Menu.add('list value', ['var1','var2', 100, 200])
```

Figure 3.17: Representation of a Python list

3.5.1.5 Tuples (Python)

Data items defined with Python tuples as value can be represented through the web interface. This will result in a special field expecting only a tuple declaration with Python syntax.

```
Menu.add('tuple value', (10, 10))
```

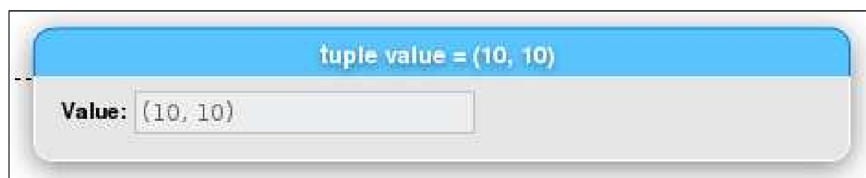

Figure 3.18: Representation of a Python tuple

3.5.1.6 MinMax

It is possible to add a `minmax` attribute when defining a data item in the DataPool API. The `minmax` attribute must be a tuple on the form *(min, max)* and

the interface will show the minmax attribute with its own input field. Usually this value will not be up for change, but a design choice to still show it. The data item with a `minmax` defined will attach a built-in rule towards the assigned value of the data item itself. As shown under, the value of the data item is 2, hence inside the allowed (min,max) interval. Otherwise, an error message will be shown before the web interface starts.

```
Menu.add('minmax interval value', value=2, minmax=(1,5))
```

The user has the option of changing the `minmax` on-the-fly in special cases when the value of the data item still needs to be in another interval. This could be looked upon as an extra security.

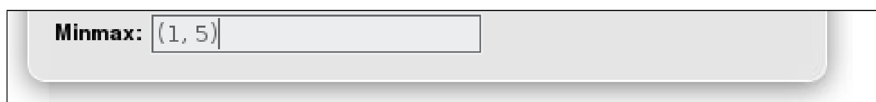

Figure 3.19: Representation of a Python tuple (minmax)

If the value is changed outside of the interval an error message will be displayed:

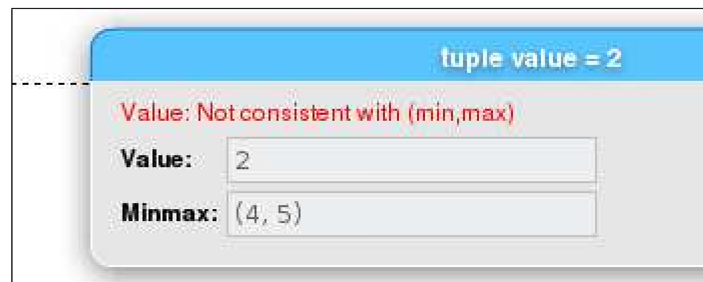

Figure 3.20: Error message using minmax

When defining and changing the `minmax` tuple, the values must be correct according to the valid (min,max) limits or an error will show:

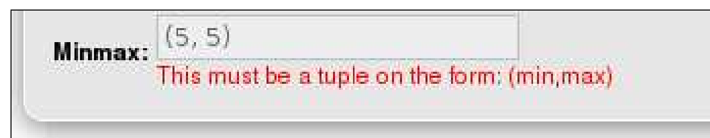

Figure 3.21: Error message when defining minmax

3.5.2 Administration Panel

3.5.2.1 Manipulating Menu Tree

The administration panel offers three different actions that can be performed on the whole menu structure with just one click. As mentioned, the structure

is initially collapsed showing only the root at start-up. From this point the users are free to navigate manually, but dealing with different kinds of menu structures and simulations will create different needs from time to time. In DPW the user can perform these procedures:

Figure 3.22: Administration Panel

Expand all. Pressing this button will expand all submenus and data items in the menu structure. This also includes disabled menu elements due to dependency rules. The function is very helpful when it is desirable to take a deep look into a menu structure with many levels. Especially, the action is convenient when in need for a closer look at parameters of disabled submenus, as instead of using the Menu Chooser in order to first activate the submenu (and branch) and then slide down the data item body before finally looking at the attributes.

Expand menus. This action is almost similar to the former described and the difference is to not show the data item leaves at the very last submenu level. A reason to use this function could be the need for a quick overview of the menu tree only focusing on submenus. This action is similar just excluding the data item leaves at the last level. This is useful when large numbers of data items are present at this last level, avoiding distraction in those situations.

Expand everything. A press of this button will expand all submenus, data items and their belonging bodies. The result is a complete expanded menu tree showing absolutely all fields. The action will also affect disabled menu elements. A typical example of use is simulations that need a lot of adjustments of attributes prior to start, and this makes it convenient to systematically go from top to bottom filling out the fields.

Collapse all. This will collapse the complete menu structure just showing the root, equal to the initial start-up. This useful if the scientist wish to start from scratch at the root and do a manual navigation.

3.5.2.2 Saving Values

During a session of DPW it is possible to save the menu structure state unlimited times before starting the simulation. This is done in the administration panel. Validation and value conversion will be performed on each save. This gives a safe environment for periodical savings when handling a large menu structure that takes long time to plan.

Figure 3.23: Administration Panel - Status - Saved parameters

3.5.2.3 Starting Simulations

The simulation is started in the administration panel and an indicator will show while the simulation is running.

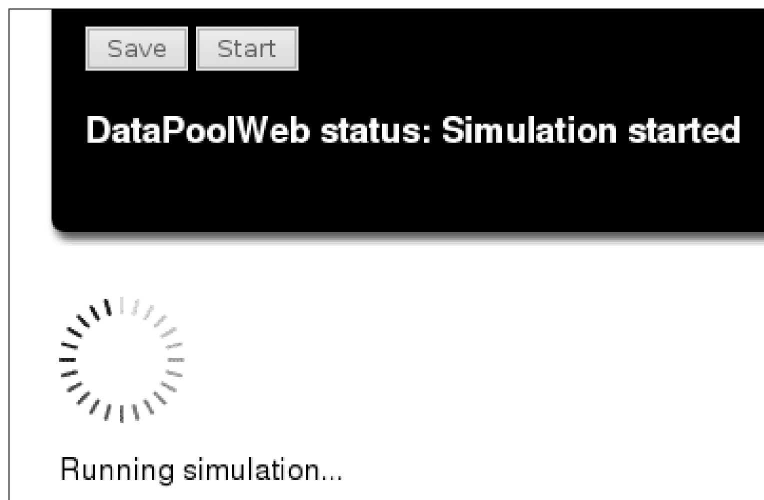

Figure 3.24: Administration Panel - Status - started simulation

The administration panel will also indicate when the simulation is finished:

Figure 3.25: Administration Panel - Status - Simulation done

3.5.3 Choosing Menu Items (submenus)

As mentioned in section 3.4.5, the Menu Chooser will be the place to choose between the available submenus at a level with a dependency rule assigned. In a practical situation, the concept of choosing between submenus with an existing dependency on the parent, is to adjust the web interface for better user interaction. This means DPW will activate/disable submenus and data items and the purpose is to indicate which parts of the menu structure that will be included if the simulation was started. The web interface will be adjusted real-time on each choice with the Menu Chooser. In addition DPW will disable input fields belonging to disabled data items, and restore the state when signaled.

Using the DataPool API it is possible to define the dependency rules in different ways. First of all it is important to be sure we are at the right level for which the rule should be assigned. The *current working menu (cwm)* will be given the dependency rule:

```
# Go back to 'friction' (this menu) to add the dependency
Menu.submenu(submenu)
# Add dependency
Menu.dependency('model', min=0, max=1, none_menu='zero')
```

Three of the parameters are required, and the fourth, none_menu, is semi-required and only needed if min=0, since DataPool need to know about a fall back value when none of the children submenus are chosen.

- chooser – the name of the data item which is used as Menu Chooser. This must be an existing data item belonging to the current *cwm*.
- min – minimum number of menu items that must be chosen.
- max – maximum number of menu items that can be chosen.
- none_menu – must be set if the dependency allows none of the children submenus to be chosen (min=0). This is the value that is used when none submenus are chosen, typically 'zero'. This parameter is semi-required.

We can as explained in section 3.4.5, define two types of dependencies which results in different user interactions in the web interface. This is decided by the parameters provided when calling:
Menu.dependency('menu_chooser', <other args>).

This results in a dependency rule that allows maximum one children submenu to be chosen, but not compulsory to chose any at all:

```
Menu.dependency('model', min=0, max=1, none_menu='zero')
```

This results in a dependency rule that demands one and only one children submenu to be chosen:

```
Menu.dependency('model', min=1, max=1)
```

This results in a dependency rule where it is possible to choose maximum two children submenus, but still not compulsory to chose any:

```
Menu.dependency('model', min=0, max=2, none_menu='zero')
```

This results in a dependency rule that demands minimum one children submenu to be chosen with a maximum of two:

```
Menu.dependency('model', min=1, max=2)
```

This results in a dependency rule that demands exactly two submenus to be chosen:

```
Menu.dependency('model', min=2, max=2)
```

Taking the Oscillator simulation as a starting point and the earlier shown creation of the dependency rule for the submenu `'friction'`, the interface will show the attached rule with the *anchor symbol* next to the menu element header. The interface will also append more information in the Quick-menu to the submenu if a rule is created. The Quick-menu will show the dependency specification, and denote which of the data items that is set to be the Menu Chooser. This makes it easy to get full control over the possible choices of actions regarding the children submenus.

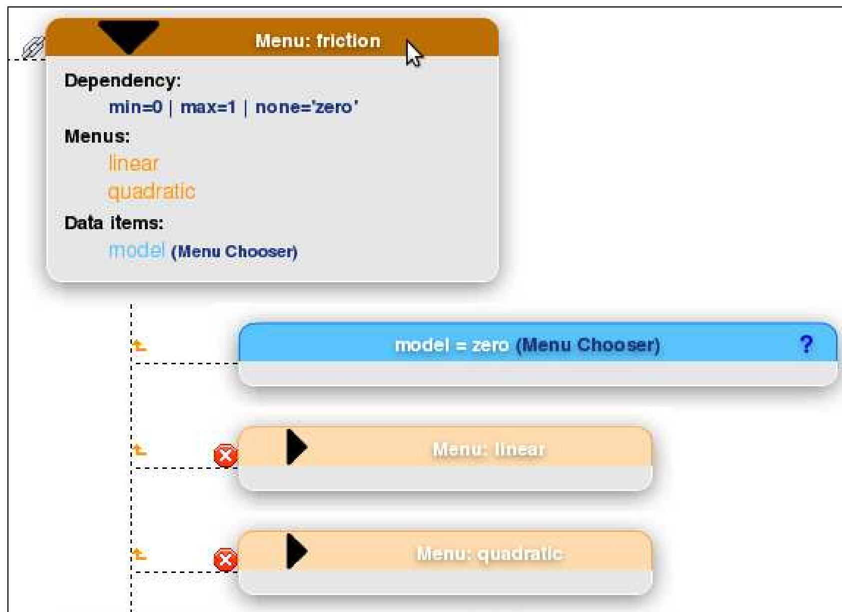

Figure 3.26: Menu item Quick-menu dependency specification

The initial state of the children submenus belonging to the parent `'Friction'` shows that both of them are disabled because the Menu Chooser holds the value `'zero'`. This value indicates that none of the submenus are chosen. Disabled submenus are not possible to expand in order to handle other menu elements beneath the level. However, DPW is designed to still make it possible to hover the submenu header to slide down the Quick-menu for a glimpse of the underlying elements. The scientist should still be able to use this future, no matter if the submenu is disabled for expanding/collapsing, to find out what the submenu has to offer. This choice of design is at the same time clearly showing the user the state of the menu structure while preserving the user mobility. therefore, two different concepts with the goal of increasing the efficiency, should never counteract each other.

It is important to point out the possibility to expand the whole menu structure if this is needed regardless of disabled submenus. This is accomplished by using the administration panel in the former explained way. However the fields will be disabled and read-only, preventing user input:

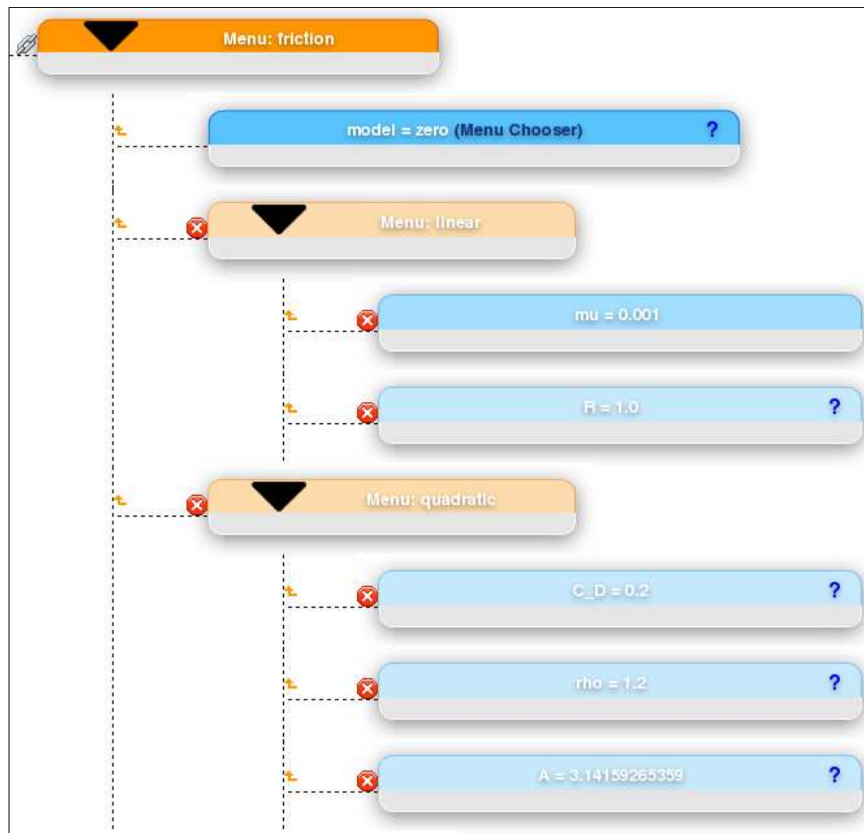

Figure 3.27: Dependency: Submenus not chosen, but expanded

Calling the `Menu.dependency('model', min=0, max=1, none_menu='zero')` will result in a Menu Chooser that is presented with a select drop down list, assuring the choice of only one submenu:

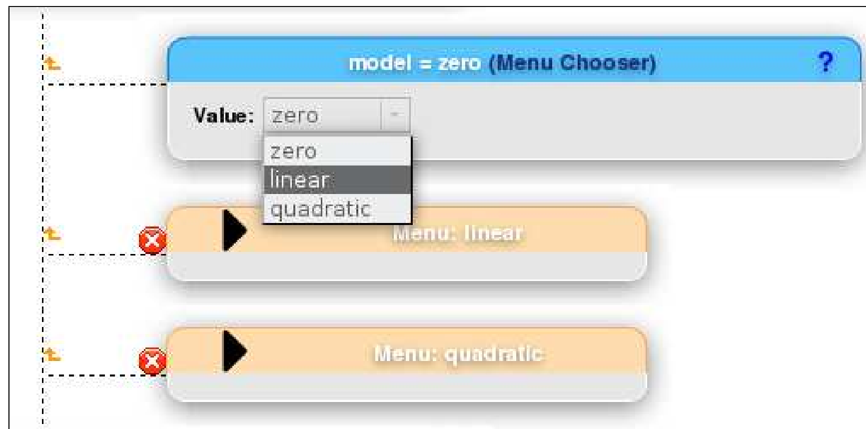

Figure 3.28: Single dependency: Choosing submenu

The choice will immediately activate the chosen submenu which are reflected with restoring it to the normal color and indicated with a `'chosen'` mark. All menu elements at the lower levels will also be activated:

Figure 3.29: Single dependency: Submenu chosen

The other type of dependency is multiple and visualized by MultipleSelect field allowing more than one value to be chosen. Here is a hypothetical example with both submenus selected after calling the DataPool API with these parameters: `Menu.dependency('model', min=1, max=2)`:

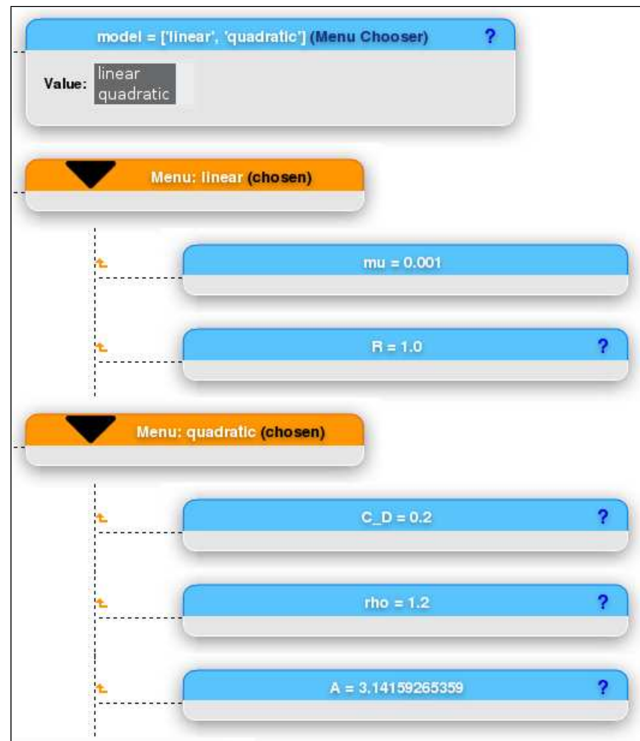

Figure 3.30: Multiple dependency: Both submenus chosen

When dealing with a multiple dependency the menu structure must be saved in the administration panel in order to be updated with the values from the Menu Chooser. Here is only one submenu selected from the list of choices:

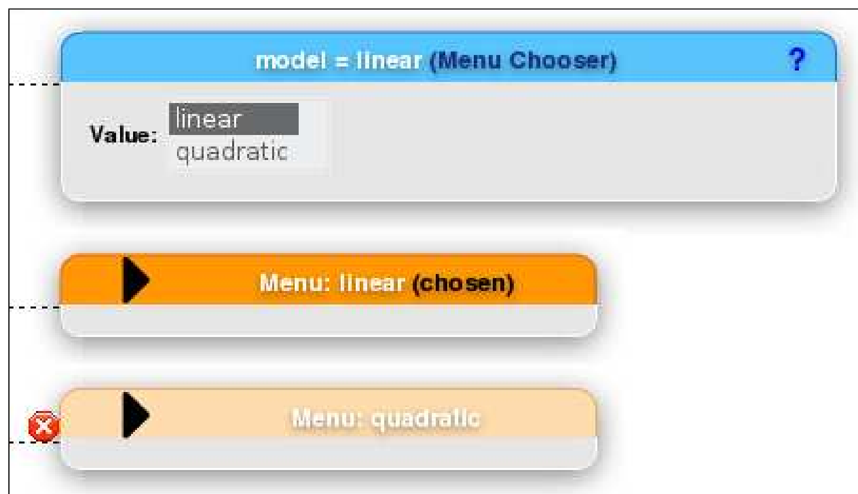

Figure 3.31: Multiple dependency: One submenu chosen

A SelectMultiple field introduces the possibility of choosing a combination

which is not allowed compared to the single select field. The most obvious one in this example is breaking the rule of choosing below the defined minimum limit. Through the API, a following call:

```
Menu.dependency(model, min=2, max=2)
```

will result in a convenient error message according to the specification of the dependency rule:

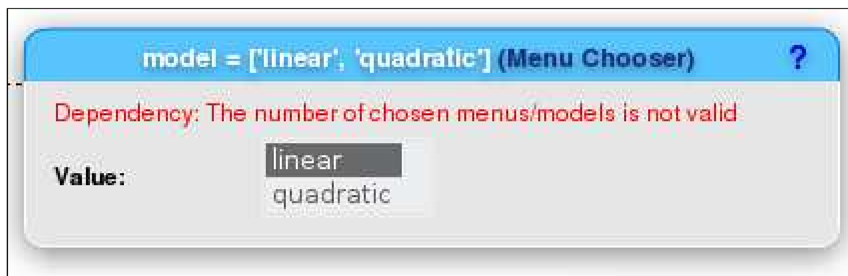

Figure 3.32: Multiple dependency error: Limits

The following call, defining a multiple dependency with the possibility for a none value:

```
Menu.dependency('model', min=0, max=2, none_menu='zero')
```

will result in an explaining error message when making a wrong combination with the `none_menu` valued `'zero'`:

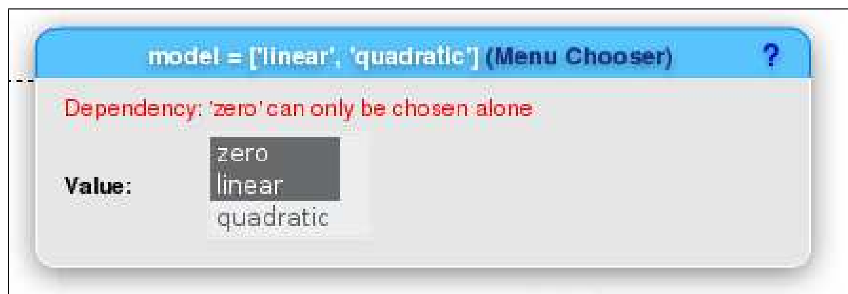

Figure 3.33: Multiple dependency error: None value

3.5.4 Value Unit Conversion

The DPW is equipped with an automatic value unit conversion. Handling units are a source for errors in scientific computing. Especially manual calculation as a tool for conversion demands a high degree of control and accuracy by the scientist. Therefore, the web interface will be adjusted if the user have provided a unit when defining data items through the DataPool API. In the Oscillator simulation, the `Physics` parameter `'m'` could be defined in this manner, providing the parameter `'unit'`:

```
Menu.add('m', 1.0, unit='kg', help='mass')
```

Creating a data item with a unit will be read-only in the web interface, and will be labeled *Unit* in the body of the data item. The field for inputing a unit type for the value of the data item is labeled *Value Unit*. The functionality becomes an optional built-in automatic conversion. On every load of the web interface the Value Unit and Unit field will naturally contain the same unit when no changes are made:

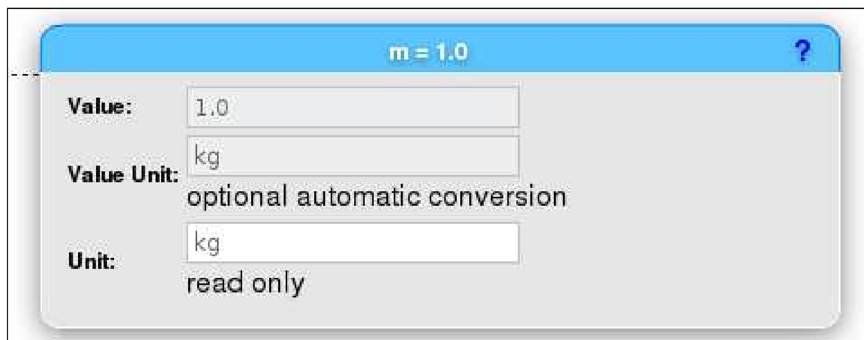

Figure 3.34: Data item with defined unit attribute

In the last figure the value `m` could be read like `1.0 kg`. If we were in need to change the unit of the input value to `g` instead of `kg`, this could easily be done like this:

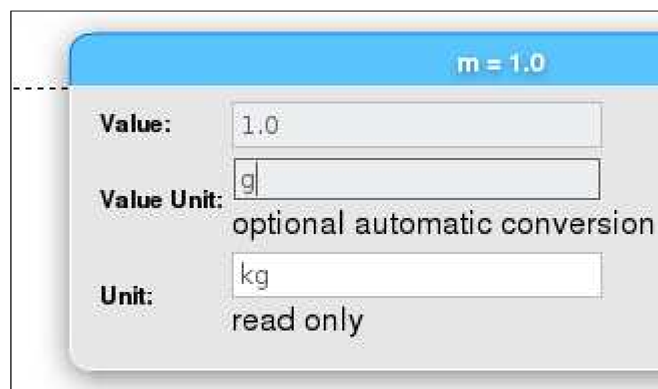

Figure 3.35: Changing unit of the input value

Further it is possible to change both the new value and the preferred unit for the new value at the same time:

Figure 3.36: Changing both data item value and the unit of the input

The automatic conversion feature will convert `5 g` into the previously defined `unit` (read-only) of the data item which is `kg`, hence the value will be converted accordingly. The automatic conversion supports all quantities in the module `Scientific.Physics.PhysicalQuantities`.

Figure 3.37: Data item value converted

3.5.5 Balloon Help

The `help` parameter can be specified when creating data items:

```
Menu.add('initial u', 1.0, help="u(0) initial condition")
```

This is an optional but a very helpful description of the data item parameter and the purpose of it. The web interface will provide a question mark in the data item header if the help information is added by the user in the API. To view the help text, hovering over the mark will show the information in a new box floating at the right side. Choosing this hover approach in DPW is to avoid the amounts of information presented through the web interface, but still be easy to obtain when needed.

Figure 3.38: Balloon help

3.6 Input Error Handling

3.6.1 Python Values

DPW will on each saving do validation and return possible error messages. The errors will be appended to each of the corresponding input fields. The system uses a comprehensive set of technicalities in order to give the user appropriate messages. The web interface is designed to stop all possible errors at an early stage to save time in the long run. The interface is designed to be generic and easy maintainable for the user, but the underlying goal is to never allow a simulation to start if the input are not totally valid. Under are some examples on how the system will react upon different wrong input:

Figure 3.39: Input error: Float

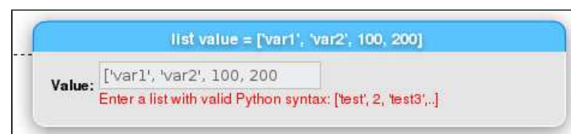

Figure 3.40: Input error: List

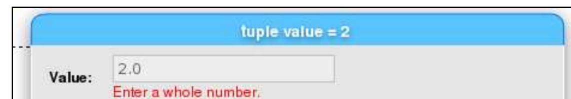

Figure 3.41: Input error: Integer

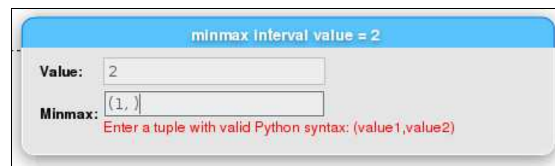

Figure 3.42: Input error: Tuple

3.6.2 Wrong Input

The administration panel will always show errors that needs to be fixed before the simulation can be started. The panel will show a list over the existing errors and take use of the helpful aspect of being placed statically. The user can easily go through each and all of the errors in the list to handle them in a systematic

way if wanted. The panel will provide the full path to the affected data items that are concerned with input errors. The path is constructed to be clickable on all levels, making it very useful for fast navigation using the automatic scrolling feature when the menu structure is large. A centralized static overview with clickable paths with animation scrolling to the affected area is a feature which saves a lot of time. The affected attribute of the data item will also be denoted in the panel with the belonging error messages. Also, these messages are stated locally under the input field in the data item body.

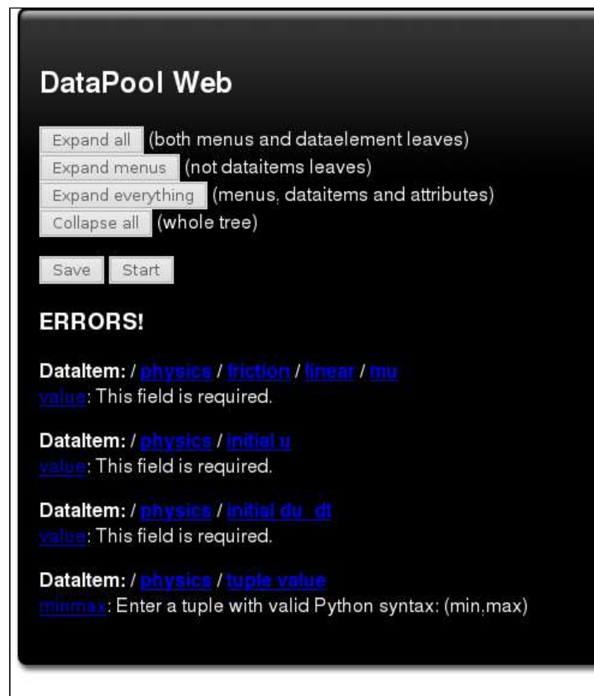

Figure 3.43: Administration panel error links

Here are some of the error messages caught in the same view which also can be found in the error listings in the administration panel in the last figure:

Figure 3.44: Input field errors

3.7 Output

DPW provides functionality to output results from the actual simulations. The normal approach and outcome from computer simulations are dumping of results to files and/or plots. When this web interface can provide results integrated in the same environment from which the simulation were started, the whole system can be compared to be more like a problem solving environment (PSE).

3.7.1 Plot Visualization

The web interface offers a dynamic real-time visualization of plot results from simulations. The scientist activates this feature through the creation of a sub-menu with a special reserved keyword:

```
# Show Visualization in DataPool Web
Menu.submenu('/ShowPlots')
# Define image types to show in web
Menu.add('image_types', ['png'])
Menu.add('source', os.getcwd())
```

- `'/ShowPlots'` – This is the required name of the submenu in order to activate the DPW to show plot results from a simulation.
- `image_types` – This is a list over the file formats the DPW should look for, and all files of the provided formats will be shown in the web interface.
- `source` – This is the location where the web interface should fetch the plot files. In this example we are using the *cwm*, but this is made flexible to suit different needs.

The creation of the submenu to visualize results is done as the last step before sending the quit signal `Menu.quit()` in the running simulation code. After the factory/scan methods are finished with collecting the values from the web interface, the scientists will probably first call the `solver` methods in order to start the simulation internally. As the last step involving the DataPool module, the `Menu.quit()` method must be called in order to send signal for change of state, and to do necessary internal adjustments before the main method in the calling simulation script is finished.

When the simulation is finished the interface will be updated instantly with the plot results underneath the administration panel. The results from the simulation can be scrolled through, depending on the amount of produced results.

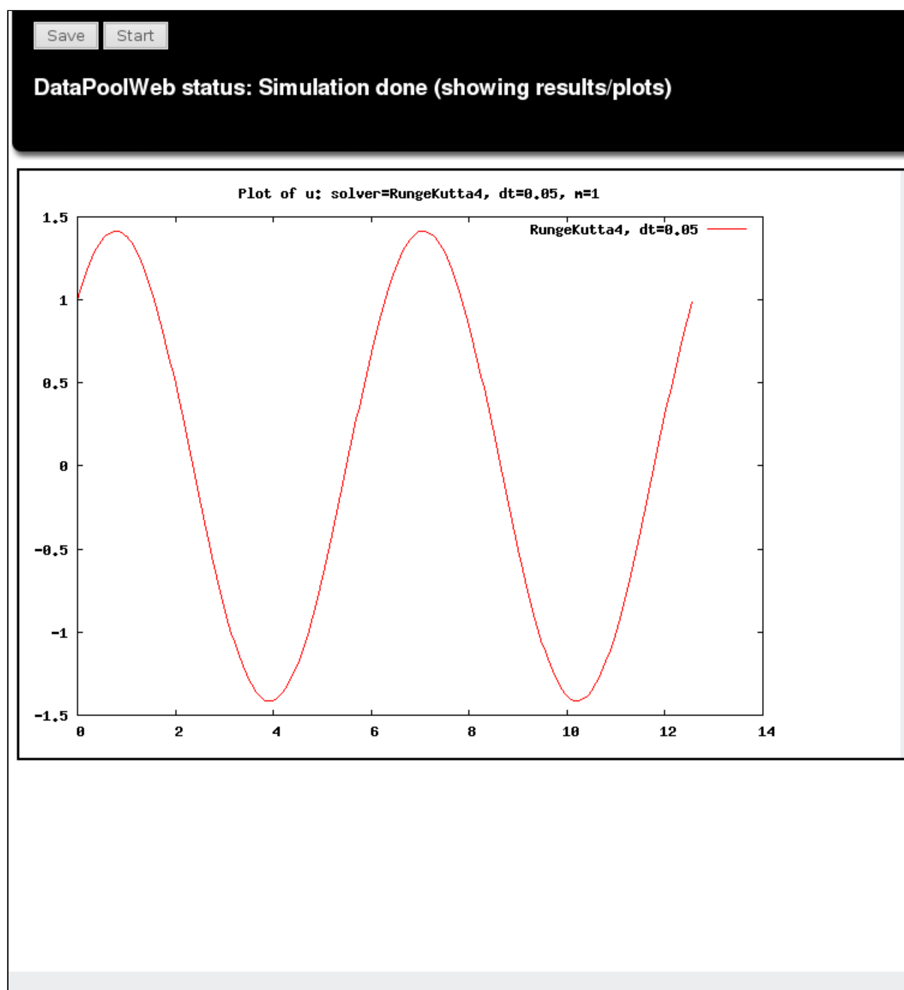

Figure 3.45: Simulation plot results

3.7.2 Simulation data

In addition to plot results the web interface can show data results also. Using another simulation called `ball1_import`, a version of the original `football1.py`

script, we are measuring different drag and gravity forces on a kicked football. This is a simulation not involving any plot results, and ordinary results from this simulation in the web interface can be accomplished by defining a special reserved '/Results' submenu:

```
# Create the needed 'Results' submenu
menu.submenu('/Results')
menu.add('hard kick drag force', hard_kick_drag, str2type=float)
menu.add('soft kick drag force', soft_kick_drag, str2type=float)
menu.add('gravity force', gravity_force, str2type=float)
```

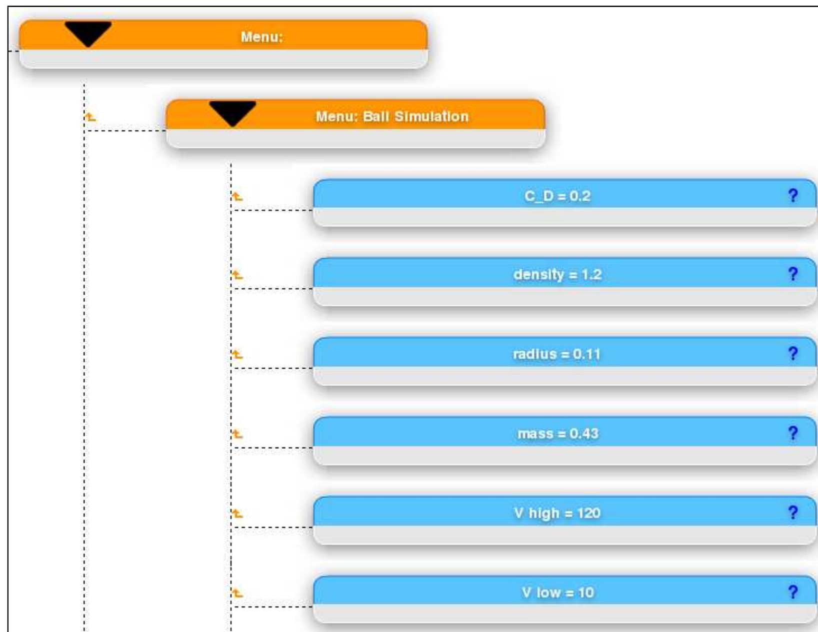

Figure 3.46: Ball simulation in DPW

The web interface will be updated with the preferred results upon a successful simulation:

Figure 3.47: Simulation data results

This is the whole and extended code for running the `ball1_import.py` simulation and almost similar to defining plot results:

```
"""
Compare the drag and the gravity forces on a body moving through air.
In particular, compare the forces for a hard and a soft kick of a football.

This version applies the DataPool module and defines data items and
retrieves values with bulk calls involving multiple items at a time.
The definition of the data items are put in a separate file.
"""

def drag(C_D, rho, A, V):
    return 0.5*C_D*rho*A*V**2

def gravity(m):
    g = 9.81  # m*s**(-2)
    return m*g

import DataPool
menu = DataPool.menu
from ball1_dataitems import data_items
menu.submenu('Ball Simulation')
menu.add(data_items)
```

```
# Prompt for web ui
menu.start_ui(ui='web', port='8001')

names = [item[0] for item in data_items]
C_D, rho, a, m, V_hi, V_lo = menu.get(names)

from math import pi
A = pi*a**2          # cross section area normal to movement, m^2
V_hi = V_hi/3.6      # velocity in m/s
V_lo = V_lo/3.6      # velocity in m/s

# "simulate":
hard_kick_drag = drag(C_D, rho, A, V_hi)
soft_kick_drag = drag(C_D, rho, A, V_lo)
gravity_force = gravity(m)

# Create the needed 'Results' submenu
menu.submenu('/Results')
menu.add('hard kick drag force', hard_kick_drag, str2type=float)
menu.add('soft kick drag force', soft_kick_drag, str2type=float)
menu.add('gravity force', gravity_force, str2type=float)

# Always send quit signal
menu.quit()
```

3.7.3 Combined

DPW is designed to handle both plot and ordinary data results integrated in the web interface. The users are free to define both `'/Results'` and `'/ShowPlots'` submenus in the same simulation to initiate both types of results. Beneath is a hypothetical example based on the Oscillator simulation which implements both functionalities. The results data (`'/Results'`) showed here are only for example and proof-of-concept and would not be of any real use in this simulation, but it shows the possibilities:

```
def main():
    problem = Problem()
    problem.define_menu()
    solver = Solver()
    solver.define_menu()
    viz = Visualizer(problem, solver)
    viz.define_menu()

    print dir(problem)
    print dir(solver)
    print dir(viz)
    import pprint

    # Promt for UI
    Menu.start_ui(ui='web', port='8001')

    # Read input data:
    problem.scan_menu()
    solver.scan_menu()
    viz.scan_menu()

    # Simulate and visualize:
    solver.solve(problem)
    viz.visualize()

    # Show Visualization in DataPool Web
```

```
    Menu.submenu('/ShowPlots')
    # Define image types to show in web
    Menu.add('image_types', ['png'])
    Menu.add('source', os.getcwd())

    # Show results in DataPool Web
    Menu.submenu('/Results')
    Menu.add('T', self.T)
    Menu.add('dt', self.dt)
    Menu.add('method', self.method)

    # Simulation done signal
    Menu.quit()

    print Menu.get_tree(show_items=True)
```

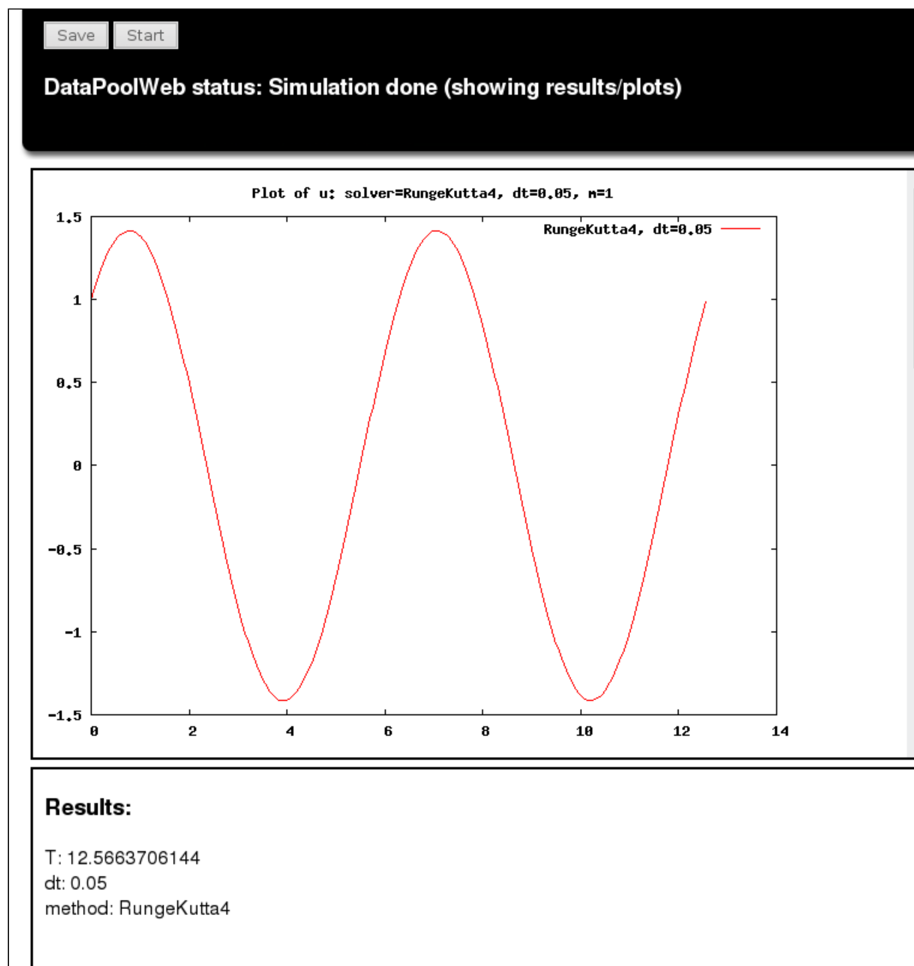

Figure 3.48: Combined simulation results

Chapter 4

Web frameworks

If we go back to the old days of the Internet, the Web 1.0 era, web pages mostly consisted of static pages or in best case generated pages, but they were still static. The term Web 1.0 is actually named after the heavily used term of today, Web 2.0, not the opposite. During this era web developers designed simple web pages that only returned plain HTML to the end-user for each server request. Regarding generated pages at this time, CGI (Common Gateway Interface) was the main actor, a protocol for web servers that made it possible to run external programs that generated the requested web page [21, p. 25]. After the CGI generation of the content, the pages were redirected back to the end-users browser in the same way as non-generated static web pages. How the server requests were handled, regarding CGI or not, were solely based on the server configuration. Because of this configuration, the server could redirect the different requests (URL) to corresponding CGI-scripts for specific sub pages. A *CGI program* or script could be implemented in any programming language, but usually Perl or Python were chosen.

If we were to make rich web applications (RIA) today with everything the task concerns, several technologies are involved behind the scenes. Those technologies are actually the foundation used in different compounds in so called Web 2.0 applications; JavaScript, Python/Perl/PHP/Ruby, CGI, HTML and HTTP [21, p. 17]. It is of course possible to make spectacular and advanced web application while using the technologies separate from the start, but this will no doubt make things very time consuming and laborious as the size of the application increases. Briefly, the HTML-code is the structure of the web page and belonging CSS specifies the layout and the visual style of the page. JavaScript are integrated code in the HTML-code on the client-side controlling page interaction and communication towards the server. The CGI-script (or a Python-script) receives requests from the JavaScript-code or the browser, and answers with generated HTML/XHTML/XML/JSON. HTTP is a simple text-based network protocol on the top of the TCP/IP stack, which defines requests or responses to/from the user. With use of the independent technologies together in their pure form in development of web applications of a certain size, the developers will often experience a stall in the development process very quickly and makes further progress hard. The use of these technologies together illustrate all the server- and client-side components involved in the process of

producing dynamic content.

4.1 The role of Python

Experienced users of Python from earlier days have noticed that Python as a programming language has expanded it's user mass, especially in the web business, from just being a web page generator, to becoming a server-side software for control of web pages [21, p. 17]. I want to point out that Python always has been, and is, most used for scientific computations like simulations and visualizations, but also for typical desktop applications, 3D engines and diverse forms of scripting. The reason of why Python has gained a solid share within web development is mainly because Pythons is fast, high-level, completely object oriented and interpreted. For a web developer this means coding and testing can be done in several small rounds along the road, and very valuable under iterative development [21, p. 12]. This results in a great starting point for experimental development that encourage an agile strategy, which often is a necessity for the creation of creative and useful web applications. The explanation is quite simple, the applications are changing fast and the small core teams usually consist of developers working close together. A language that offers compact and dynamic code is a huge benefit in this context of web development, in addition to the possibility of self-modifying code under runtime [21, p. 13]. One of the main advantages of Python is the massive selection of extension modules made by the enormous Python community. With this solid collection of functionality and add-ons, advanced problems on the server-side are also highly manageable if the web application is dependent off some specific underlying functionality or some other services. Independent programs can actually be run on the same server hosting the web application, and in these cases both the logic of the main application and the needed add-on programs, which make up the total system, are written in Python.

4.2 First Approach

The first try to ease the pain for the many web developers was PHP (Hypertext Preprocessor), a server side scripting language that generated the HTML in an easy way [21, p. 47]. PHP became enormously widespread and also the most popular language used for the web. Today PHP is still used in a large scale in all types of projects. Web frameworks for also exists for PHP, but the obvious trend of older PHP developers turning their back to this language is also the reason for the decreasing user mass. The observed pattern among developers is soon as they get experience with the web frameworks for Python, or Ruby however, they tend to not fall back to PHP. The clear point of using a web framework is primarily to save time writing the same code like handling database interaction and URL mapping, but also to avoid all types of duplications of the same type of code written several places. Without the use of a web framework the developer actually has to reinvent the wheel every time an application is started from scratch when it comes to settings, configuration and other general code, which has to be present to even make the web application work in the

first place [21, p. 48]. It is considered hard, impossible and a waste of time to hard-code extensive solutions in this manner. One of the significant obstacles is the fact of tangled code, and a little change dealing with one page can result in several other changes that has to be done around in the system. Further, this results in tangled code with a tight coupling because of lack of a higher level of abstraction. Typical boilerplate code can be separated from the rest of the system and considerably reduced with a web framework, and the achievement of cross-browser compatibility is a also assured.

4.3 Inter- and intra crosscutting

In a journal [12] by Kojarski and Lorenz, the problem of pure CGI-scripting in large projects are discussed in great detail. They have a clear vision of the problems: We distinguish between intra-crosscutting that results in code tangling and inter-crosscutting that results in code scattering [12, p. 53]. Intra-crosscutting occurs by code tangling within individual pages, and inter-crosscutting happens because of code scattering across all the pages of the web application [12, p. 54]. The intra-crosscutting does only map problems within one isolated page, in a way that functionality, presentation and control are tangled together on the page. Intra-crosscutting is a shortcoming of dynamic pages by design, and depends only upon the specific functionality, presentation, and control code within the affected page [12, p. 54]. The more severe inter-crosscutting, deals with the total underlying structure of the web application regardless of existence of tangled functionality- or presentation code, that often leads to multiple changes of several belonging pages of the web application. Inter-crosscutting affects most of (sometimes all) the application pages, both dynamic and static. [12, p. 56].

As mentioned earlier, the intra-crosscutting results in strong dependence between the different types of content on the web pages. Groups of developers have to cooperate closely if their gonna accomplish to maintain the code on the specific pages in this coincidence. The clear division is drawn between web designers and web programmers, which will be especially evident during redesign where the other divisions have to handle documents with code exceeding their ability and area of expertize [12, p. 56]. In this context the web designers normally handles all types of HTML/CSS and some JavaScript in few cases, and the web programmers maintain and develop all types of template-language, scripting-language, server-side, settings and configuration and even most of the JavaScript code.

On the other hand the inter-crosscutting affects both static pages and dynamic pages, compared to the intra-crosscutting which only concerns dynamic pages. This involves the structure of the web application as a whole, and is tied to the underlying resource structure of the application: The structure concern is a rigid skeleton thats cross-cuts multiple application files coupling the web application to a particular underlying resource structure [3, p. 56]. The inter-crosscutting problem can be compared to the sync state towards the underlying resource's structure of the application. This can typically be hard-coded name of variables, parameters and relations to the database. A clear example is the

need for a developer to go over all affected locations in the code and edit them manually, and usually includes most of the pages that integrated such information in their context [3, p. 57]. This can also go under the terms of unweaving the application, before weaving it together when changing the existing structure of the web application [3, p. 57]. However, there is a demand to maintain the sync and the consistency with the structure, and the these issues involved result in a high development and maintenance cost.

The MVC pattern of Django is a clever example of attacking this problem, but only the intra-crosscutting problem. Inter-crosscutting is also getting under control with Django, but that is solved due to the underlying structure of Django as a web framework itself, not the resulting MVC pattern of the web application made with Django. Kojarski and Lorenz are stating that: Despite its obvious severity, no comprehensive solution for the inter-crosscutting problem exists to date [12, p. 57]. This thesis will investigate and present the aspects of this web framework for Python, and even using it in highly advanced tasks in computational science. As mentioned under the specifying the thesis about the importance of making the DataPool system as generic as possible, we can avoid the common way for advanced developers making very specific ad-hoc systems which comes short in being generic: Having no institutional support, advanced developers sometimes build their own ad-hoc tools to tackle specific cases [12, p. 57].

4.4 Introducing Django

Django [27] is the web framework for Python that has received most attention. The framework also uses the model-view-controller (MVC) pattern for the underlying architecture for applications. The MVC pattern was originally developed and implemented with the language Smalltalk, and gives a clean structure with separation of presentation-, control- and business logic [21, p. 48]. It is very common for all desktop applications to use a sort of MVC in their architecture, and by using this in the web framework as well, gives a solid convention and a steady foundation from the start. Django was originally a project of the World Online team, a web team maintaining several newspapers on the web such as the Lawrence Journal-World. It was created from scratch with one clear goal, a system for faster production of newspapers on the web. The Web Framework for perfectionists with deadlines, is the mantra of Django [27]. Compared with the TurboGears web framework, Django is not built upon already existing components, but only on self produced parts [21, p. 71]. Django was developed based on the need for frequent updates of the websites with new articles and blog entries. In other words, a CMS (Content Management System) with high capabilities of producing scalable and dynamic newspapers on the web with a lot of data. Django is dynamic in all senses and follows principles like DRY (*Don't Repeat Yourself*), MVC and KISS (*Keep it Simple, Stupid*), in addition to the framework architecture being built upon modules for easy expansion of functionality from the large community that also has taken over the future development of the framework. By following this combination of principles the amount of redundant code are taken to a minimum, like the specification of database models which is intuitive in Django. The framework is built upon different components:

Templates

Django templates contains HTML and control logic in form of a template language specially developed for Django. This language provides contexts of variables and control logic which makes the dynamic generation of the markup and HTML content on the web pages.

Views

This is Python functions with the purpose of gathering data from the database, and also carry out calculations or processing of data before the resulting data is sent for further handling by Django templates.

URL-mapping

This consists of the important urls.py files, which provides proper handling and redirecting of the incoming URL-requests to the right view functions (see over). The real power behind this module is the regular expression (regex) implementation which manage the matching of URL-patterns and fetches the the desired data from the URL-string and attaches this to the right view function.

Model

This is the Django ORM (Object-relational mapping) providing automatic mapping between database models and model-objects. Worth mentioning is the abstraction of ordinary SQL queries. With the built-in ORM, the writing of SQL queries can be completely avoided and the task of analyzing long and complex SQL queries is replaced with an intuitive and effective database API. Django is also offering the opportunity to write SQL queries if this is needed.

4.5 Loosely coupled Django

Django's philosophy of being a loosely coupled web framework makes the components of the framework independent from each other. Web designers and HTML-coders should avoid to concentrate about other code in the system because this can be disturbing and a big source of errors. Isolated components makes it easier to learn the different parts of the framework. Practically this structure arranges a simple way to effectively switch parts in the system without huge rewrites of the remaining parts of the system. Django's focus on not doing too much magical tricks behind the lines, is what makes Django a good web framework. As mentioned, the code is compact, but in no way resulting in a bad overview of how the information flow is related in Django.

4.6 TurboGears

TurboGears [8] is the web framework for Python that relates to the more typical Web 2.0 term. Compared to Django, this framework were not built from scratch

using its own written components, but consists on these existing independent technologies: SQLObject [24], CherryPy [6], Kid [11] and MochiKit [18]. TurboGears can be said to be the answer to Ruby's well-known web framework Rails. As if TurboGears is Python's real competitor against Ruby On Rails and have a strong and integrated support for Ajax, Django on the other side can be stated to be more a generic web framework. TurboGears vision is to be an out of the box engine towards making database driven web applications focusing on the Ajax-power, in a typical Web 2.0 spirit. Django on the other hand originates from the need of a CMS. In the journal Python for Scientific Gateways Development by Heiland et al [1, p. 1], they point out that:

> However, what are known as Enterprise Frameworks has dominated much of that landscape for the past several years. In this paper, we offer some alternatives that, we believe, are lighter weights in the terms if their development and deployment costs yet empower scientific communities more than ever.

This journal is one of very few which mentions these two web frameworks for Python in a context of computational science [1, p. 3], and also describes the new trends among web development in this area. Actually, in computational science, TurboGears have support for some pretty complex widgets for visualization like data grids, 2-D plots and different multiple-value selection [1, p. 3]. This journal also presents the need of allowing users to easily define input parameters and offer remote execution of simulations. Providing a gateway component for parameter sweeps could greatly improve a researcher's productivity [1, p. 5].

However, this master thesis gives priority to Django because of the need for a generic approach for the DataPool and the web interfaces developed for this system. At this time, DataPool is not using databases that is an assumption in TurboGears. Regarding the use of databases in scientific computing, this can be considered low of several reasons. The different concerns are pointed out in the journal [1, p. 39]. In any case of sudden need for a database, this can be handled by Django as mentioned. Implementation of technologies like Ajax is used in the development with Django in the thesis, even though it is not supported out of the box. The loose coupling of Django gives us the opportunity to integrate a preferred JavaScript library for the creation of Ajax-features, and is one of the main reasons for using Django, even if we could have used the same MochiKit JavaScript-library TurboGears have integrated.

4.7 Choice of technologies

To choose the right set of technologies for use in a web-based interaction, depends on many factors that needs to be weighted. Several journals are discussing this and a common pattern are clearly showing among the authors. Flash, Java, JSP, JavaScript, HTML, CSS and other plug-ins are mentioned concerning technologies for web interaction. Bethel et al [2, p. 4] presents AMR WebSheet that is completely implemented in JavaScript and HTML, which is an interface providing a display of the visualization parameter space. Holmberg et al [7] have

a solid focus on using JavaScript as the base for all types of technologies for web interaction and what JS can provide [7, pp. 5 6]. They also argument for DOM integration [7, p. 6] with JavaScript and CSS, and the wide browser support. Hamann et al [9] also touch the aspects of using Flash, Java or plug-ins [9, p. 42], but are concluding that a platform-independent solution is the main priority. One of the journals also discuss the use of SVG for the interaction, but are suggested only to be used if its also intended for the visualization of simulation, because of lack of native support [7, p. 5]. The dynamic integration of new content into active web page, the Ajax technology are discussed and used by several authors [1, 2, 7, 22, 9]. In this technology, JavaScript is the central actor [7, p. 6], and is showing consistency with the authors use of JavaScript as base for the interaction [9, p. 44]. Heiland et al [1] also points out like Langtangen [14], the possibility of editing the layout for end-user: Imagine, during the prototyping phase, that end-users (scientists) make incremental changes to the gateway input and functionality, evolving it to be optimal for their needs [1, p. 6].

Holmberg et al [7] discuss several technologies for visualization of results from simulations, in addition to Gnuplot images, GIF-animations and reports. Technologies like SVG, SMIL, Java3D, VRML and X3D are presented and discussed in the context of choosing the right technology to obtain the highest compatibility. The central of this journal is the importance of a more generic system with the possibilities of adding the different technologies when needed. They are concluding with a general consensus that there is no tool or technology that can be best suited for all types of visualization [7, p. 8]. With DataPool being highly generic, the ability of choosing the appropriate visualization technology can be one opportunity for the system.

4.8 Summary

A developer should only need to write the parts of the web application that makes it unique and compelling, and not the fundamental and general code, which all web applications must have to be running and actually work. However, not meaning the applications will be simpler or have reduced usefulness, the reason are wholly to write minimal of code to get up and running from scratch, and of course be able to do fast changes on the way for the complete system with no hard effort and problems. With other words, totally overview and control, keeping the developers concentration at the right place and where it should be, to create interesting and compelling web applications with outstanding usability.

Chapter 5

Technicalities

5.1 DPW Contents

DPW is located in the `DataPool.gui.webui_django` package. The contents of the `webui_django` have the identical structure like any normal Django project. Django comes with an utility that automatically sets up the conventional directory structure for projects and applications. The frameworks operates with two different concepts which are important to be aware of. A Django *project* (like `webui_django`) is a collection of configuration and *applications* for a particular web site. In this context a Django application is a web application used in the web site, like a blog system which is a part of the web site. Therefore, a Django project can contain multiple Django applications, and an application can be present in multiple projects.

The first level contains the mandatory files for the Django project:

- `manage.py` – A utility (command line) for interaction with the project. In this context mainly used for starting and handling the Django server.
- `settings.py` – All global Django settings for the project.
- `urls.py` – The top level URL pattern handler and declarations for the Django project.

Inside the `webui_django` project we also find the module `datapoolweb`, the application that makes up the DPW. As mentioned, at this level several applications can exist, but in this implementation we are only using one Django application for everything.

Our `datapoolweb` application contains the mandatory files:

- `models.py` – Contains the data model definition which gives a database-abstraction API used for interaction towards a database for the application. DPW is not using a database, leaving this file empty.

- `views.py` – The file contains all the functions that handles the web requests (from `urls.py`) and returns a Web response. The server-side code are located here.

`datapoolweb` contains two other packages that makes up the total DPW system:

- `media` – The package holds all CSS, images, JavaScript and visualization files for the web interface. This package is the only part of the project that are accessible directly outside of the Django application.
- `templates` – Contains all template files used by the server-side code in order to render certain HTML code.

The correlation is presented under:

Figure 5.1: The Django project structure

5.2 Implementation

DPW is implemented using a different set of technologies for code and markup. According to the MVC pattern they are used with hard focus on keeping them separated.

5.2.1 Python

All server-side code, configuration and settings are written in Python.

5.2.2 Django Templates

It is especially the Django template engine with its belonging template language we are using for producing the HTML read by the web browser. The system

have different template files for the various data models represented in the web interface. The Django server-side code will fetch the templates stored in the `templates` directory, avoiding hardcoding them into the view code. The templates could been located in the view code, but this demands a lot of string interpolation in Python, and it would be hard to get a satisfying and full overview over the context to be rendered. The solution is to construct the context for the different data models in the view code, before shipping them of to the Django template engine for easy rendering using the template language. The template language offers a very usable set of logic constructs on how the context should be rendered. Django performs this without destroying the MVC pattern.

5.2.3 HTML

We are using only one base template, `datapool.html`, which contains the typical HTML header, doctype, meta headers and script inclusions. This template defines the basic semantic markup for the whole web interface and it only includes Django template language for the generation of the global error list located in the administration panel. In addition, the template only includes one template language statement for the inclusion of the whole menu tree(!). Behind this single expression there is a comprehensive procedure on the server-side and in the template engine, which will be covered later. This approach keeps the base template to an absolute minimum. Actually, it is approximately the shortest possible. The base template is the only one that are rendered to the end-user in the web browser. All other templates mentioned in the previous section, will be rendered and used as a step in the server-side code, *before* the main rendering of the web interface using the base template.

5.2.4 CSS

The semantic HTML code is also fully separated from the CSS code, which defines the visual presentation. The CSS code consists of 470 lines of code and is located in `datapoolweb.media.css`. The style sheet here takes care of the concepts of the hovering effects, the visual elements of the DPW like rounded corners etc, fonts, placing and so on. However, the complete HTML generated by the Django template engine, were designed in a fashion to be fully operational and visualized correct without the CSS by abiding design principles. It should be pointed out that the DPW will always be using the style sheet.

5.2.5 JavaScript

The web interface is using the powerful, but lightweight, jQuery JavaScript library [10] to realize the different effects and interaction. This code for DPW is written from scratch, consists of 250 lines and is located in `media.js`. Ajax handling and DOM manipulation are implemented using this library.

5.2.6 Design Elements

The design elements used in the web interface are modified and/or extended versions of free design techniques and elements. All composition, layout, presentation and colors are special and distinct for DPW.

5.3 System Structure

We will now cover the system structure and the architecture of the whole system. Due to several distinct parts that need to cooperate and work together in correlation, it demanded a special solution. First we introduce the parts of the total system behind DPW:

Simulation code. The simulation code to be run will import the DataPool module and start using the API. The code have the responsibility of calling the methods for prompting for desired UI, and finally ending the session.

DataPool (menu tree structure). The internal DataPool menu tree structure that will be built from the simulation code through using the API.

DataPool API. The API (`MenuAPI.py`) on top of the DataPool core internals and works as the service towards the scientists taking this in use from the simulation code.

State handling. `\.datapool` is a special folder located at the local machine on the path defined in the `settings.py`. The folder have responsibility for holding a set of .dat-files that keeps track of different states in the total system.

Django server. The Django development server which is practically initiated by calling the function `start_ui()` using the API from the simulation code. The internals of the API will start the server locally.

Server-side view code. All server-side code and functions are located in the `views.py` in the application of our Django project. This module have a special method `index` which has the responsibility for receiving all incoming requests from the web browser.

Web interaction (front end). This is the user's web browser running locally on the Django development server. The browser will use the Ajax technology in order to signal the rest of the system when to start the simulation. In addition, the Ajax will handle the dynamic update of returned results from the server-side view code.

Because DPW is using the Django server locally on the machine, we instantly get the issue of not being able to communicate with the active internal menu structure tree in DataPool from the view code in the Django project. The user of the API will call the `start_ui()` method when ready to prompt for the web interface. This method, located in `MenuAPI.py`, will start the server and open the system default web browser. Unfortunately, but naturally, the server will be started in a new process and context, when a new command line window is created especially for the server (using `manage.py`). The actual standard command for starting the web browser is:

Terminal

```
> python manage.py runserver <port>
```

From the API internals, the Django server is started like this:

```
os.system('gnome-terminal --execute python %s runserver %s &'
% (django_server_path, port))
```

Another option tried was starting the server in the same command window as the simulation. Considered a semi-success the server started, but interfered with the on going handling by the API internals, making it very unstable. Researching heavily on this challenge and several approaches, the conclusion became to welcome the solution of using serializing/de-serializing of the internal menu tree structure in Python.

Surprisingly this solution turned out to be the best one in order to prove the DPW system to be powerful enough to accomplish the job without touching the active menu tree session of DataPool, using no synchronous *Python* communication. We were able to dump the internal menu tree structure using the cPickle module to file, not affecting the active internal tree at all and leaving it on pause/hold. Further, work on a copy of the identical menu structure in a new workspace in the server-side code, successfully making the web interface.

However, the solution required the introduction of state handling in a new environment which was chosen to be based on *state files*. The files are contained in the `.datapool` folder defined in the location by `TREE_LOCATION` in the Django project settings file. They consist each of a single `'0'` or `'1'`, meaning on/off. In this situation several new functions were written in the API, functions for setting/clearing the different states. The state files are not part of the DataPool internal data structure, but they were chosen to still be placed in the API. The Django view code should not be responsible for creating, open and reading files, and therefore dedicated the API. Another good reason, the API is also dependent of calling internal functions operating on the state files when called directly from the simulation code, strengthening the argument to be a consistent location for them. In the next section we will go deeper on how this is implemented.

5.4 Design Philosophy

The web interface was realized using the Django framework following a certain set of guidelines. These guidelines were defined prior to the development process to emphasize the importance of planning the implementation. One of the main goals of this thesis were to test and explore the framework regarding to functionality, conventions, design philosophy and type of approaches. This gives a natural environment and reason to involve the most important aspects of the framework and use those together in the same project. In this way we can highlight all the unique feature of the framework, try to use them in cooperation in the same project in the means of creating a proof-of-concept. We involve the

central structures, approaches and functionality of Django in one space while focusing on well-known general design principles. The result should be a compact, effective and smart code that proofs the power of the framework. Under are the development guidelines for implementing the DPW using the Django framework:

Use the most aspects of Django. Include all the important and central functionality provided by the Django framework. The solution should show what the framework has to offer.

Use the aspects to the fullest. The implemented aspects from Django used in our solution should reflect correct and expected use of them. The solution uses those aspects to the fullest showing their potential. However, the implementation should maintain the integrity while using the features of Django, using them for only what they were meant. Shortcuts in the implementation should be highly avoided.

Override Django source code. The code should reflect complexity that goes beyond the out-of-box functionality. We will try to subclass parts of the Django source code in order to preserve compact code and long term solutions. This will integrate special needed functionality close to the rest of the Django principles, obtaining a seamless and consistent solution.

Maintain extensibility. Avoid hardcoding when possible and make the way for more functionality in the future.

Minimal use of code. The implementation should only contain crucial and necessary code, keeping redundant code to a minimum. Most code in the implementation should be obvious, and reflect a balanced optimization retaining code readability.

Isolate the web interface from DataPool. The DataPool module should be isolated from the web module in highest possible degree. In theory, DPW should not be aware of the underlaying DataPool module. In this way providing the MVC pattern for both systems.

5.5 Core

This chapter will explain the essential data models and belonging functions which are used together in order to fulfill the behavior of the system. Function arguments are simplified or let out, and we will only cover the main responsibilities for the different parts.

It is important to point out one central design architectural concept in the core that everything are based upon. As explained in section 5.3, we will be working on an identical copy of the menu tree structure in the view code on the server side. Hence, we are free to alter and modify the internal DataPool menu tree in the view code in the best benefit for the efficiency. The algorithm for

traversing the tree recursively will be the same every time we are performing something on the whole tree. It is very convenient preserving this structure and the belonging algorithm to maintain consistency and pattern throughout the implementation. Another aspect which makes this approach effective, are the Django forms created for each parameter (data item). The forms are instances of a subclassed Django form class, holding their set of fields for the respective attributes of each data item. The system needs those forms to be able to do validation and render them in the web interface a number of times during a session. In the end of a DPW session, the new values from the user input also needs to be fetched and converted back to the matching data item object in the menu tree.

This makes it convenient to keep the menu tree structure and objects, and only extend the tree by making a data item wrapper object for each data item. The wrapper object will contain the real data item object, the Django form object and a prefix id in order to distinguish the forms when updating from post data. The wrapper can contain all related information for each data item. The approach keeps everything tidy and gives complete control. In the end, the extended tree is cleaned when all operations on the menu tree are done, resulting in both updates values for the data items and the tree set back to normal before serializing the tree again.

5.5.1 Django

Data model related in the Django application:

- `DataItemWrapper` – The class that operates as the wrapper for the data item on server-side. Contains the id, data item and the form.
- `LinkedTextInput(forms.TextInput)` – A subclass of the Django `forms.TextInput`. Similar to the super class, but altered to append an anchor tag with a name attribute around the input field. This link are used for linking in the animated scrolling effect to find the right one.
- `DataTupleField(RegexField)` – A subclass of the Django RegexField. Defines a new custom field type used for a representation of Python tuple input in the web interface. The class provides a custom regular expression for Python tuple syntax validation and conversion.
- `DataListField(RegexField)` – A subclass of the Django RegexField. Defines a new custom field type used for a representation of Python list input in the web interface. The class provides a custom regular expression for Python list syntax validation and conversion.
- `DataItemForm(forms.Form)` – A subclass of Django forms.Form and represents the form belonging to all data items in the menu tree structure. The form is wrapped by being put in the wrapper object during a web session.

- `DataItemForm.__init__(self, data_item, parent, *args, **kwargs)` – The constructor of the DataItemForm class. It contains all the logic for creating the different form instances for data items.
- `_create_field(self, show_attr, attr_value)` – A function called from the constructor of the DataItemForm class. Responsibility for making the appropriate field types for the data item attributes in the form instance.

Control handling in the Django application:

- `index(request)` – The main function of `views.py` and gets all incoming HTTP request from the web browser.
- `_traverse_tree_controller(action, node, data=None)` – A central function working as a controller and typically called from the `index` function. It controls all calls to other functions performing actions on the whole menu tree. The function also provides a clean namespace convention for calling tree procedures from the `index` function.
- `mi` – The essential pointer to `DataPool.menu`, the DataPool API.

5.5.2 DataPool API

State-, server- and result handling in the `MenuAPI.py`:

- `start_ui()` – API function called from a simulation code in order to start the preferred user interface. Responsible for starting the Django server.
- `quit()` – API function called at the end of a simulation code. A cleanup and signaling function that cleans up old results and plots. Also responsible for checking for the reserved keywords `'/Results'` and `'/ShowPlots'` in the menu tree for triggering and producing new results. Calls the quit signal `set_quit_signal()`.
- `set_quit_signal()` –API function called from `quit()`. Setting the state file `quit_status.dat` in `TREE_LOCATION`.
- `reset_quit_status()` – API function called from `start_ui()`. Resets the state file `quit_status.dat` in `TREE_LOCATION` to indicate normal state and simulation code could be running again.
- `clear_old_results()` – API function called from `quit()`. Removes the old results file `results.dat` in `TREE_LOCATION`.
- `remove_old_plots()` –API function called from `quit()`. Removing old plot files from `visualization` in `MEDIA_ROOT`.
- `check_for_plots()` – API function called from the server-side view code. Checking after plot files in `visualization` in `MEDIA_ROOT`. Returns a list of names of present plot files.

- `check_for_results()` – API function called from the server-side view code. Checking for results in `results.dat`. Converts present results to a Python list which are returned.
- `status_web_done()` – API function called from the server-side view code. Setting the state file `status.dat` for signaling the 'start' action from the web interface.
- `wait_for_signal_web()` – API function called from `start_ui()`. Continuously checking for an alteration of the state file `status.dat` that indicates the 'start' signal sent from the web interface, and than gives the control back to the caller (`start_ui()`).
- `wait_for_quit_signal()` – API function called from the server-side view code. API function continuously checking for an alteration of the state file `quit_status.dat` which indicates the simulation code is done, and than gives the control back to the caller (server side).

Settings:

- `DATAPOOL_ROOT = '/path/to/DataPool/root'` – Location of DataPool module on system (forward slashes on Unix).
- `TREE_LOCATION = '/path/to/.datapool/'` – Work directory `.datapool` for the DataPool module (forward slashes on Unix).
- `DEFAULT_WEB_URL = 'http://localhost:8000/datapool/'` – Default web URL for DPW.
- `DEFAULT_PORT = '8000'` – Default web URL port.

5.6 Program Flow

In this section we will cover the internal program flow of our implementation of DPW and argument for the system design. The thesis will show the total conjunction, involving the different modules of the system and explain the different flows in detail and why they were made like this. Describing specific source code will not be included here.

5.6.1 Approach

The design philosophy regarding the distinction between DataPool and DPW is central for the chosen program flows described. Regardless of the web interface working on an identical copy in its own space, we continue to follow this angle. In practice, meaning the Django implementation avoids the use of DataPool core functions. The implementation of DPW does not make any calls to the DataPool core functions directly. Calls touching the core, comes from the calls made from the called API functions. However, no core functions called, but the view code takes use of direct contact in the core, tough only for fetching the root menu pointer: `mi.menu.root`

In addition we got the different API calls described in the previous section, not working on the menu tree structure at all, but the state files. This results in complete separation of the systems. Under is the description of the involving functions.

Use of API functions which works on the menu tree structure:

- `mi.set()` – Offers interface for updating the value of a DataItem.
- `mi.get_fullpath()` – Get the path to a `MenuItem` (submenu).
- `mi.write(<file>, 'bin-pkl')` – Offers interface for saving data structure to binary file.
- `mi.load_from_bin('param_tree.pkl', path=settings.TREE_LOCATION)` – Offers interface for loading data structure from binary pickle file.

Use of other API functions that only works on state or result files:

- `mi.status_web_done()`
- `mi.wait_for_quit_signal()`
- `mi.check_for_plots()`
- `mi.check_for_results()`

5.6.2 Initial Browser View

All numbers refers to steps in Figure 5.2.
The first and initial view of the DPW in the web browser will be a result of a HTTP GET towards the server. In short, behind the scenes this process consists of first fetching the binary pickle file to load the menu tree structure internally in the server-side code to get it up running (3). Further, the process involves traversing the whole tree while creating Django forms for all data items (4). As mentioned in previous sections, this process swaps out the elements in the `items` of `MenuItem` with `DataItemWrapper` objects for each `DataItem` (7). The data item object itself is put in this wrapper object together with the newly created form object. The last thing added in the wrapper is a id number. At last, the final part in this process will traverse the menu tree once more transforming it into a nested Python list (8). Rendering the final HTML code for all data items (10) and menu items (11) are also accomplished in this process. This nested list represents the menu tree structure, holding rendered HTML strings as elements instead of menu element objects. Finally, the returned list will be sent for rendering in the base template in the web browser, which ends this request (12).

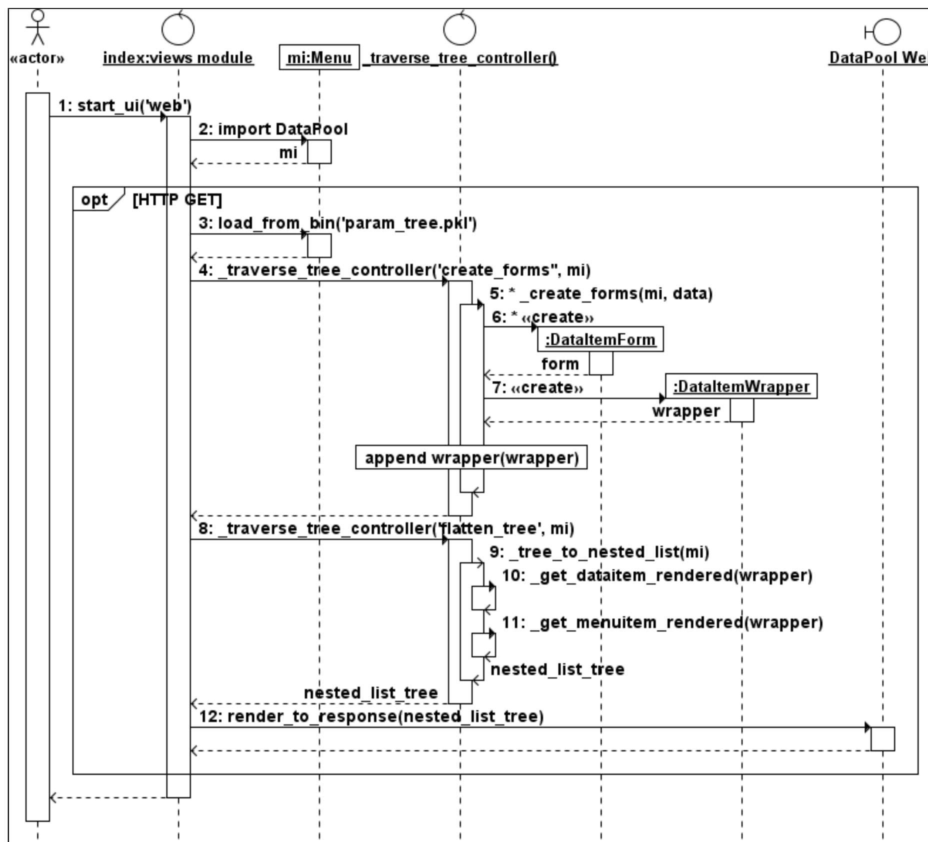

Figure 5.2: Program flow - Initial Browser View

5.6.3 Browser Interaction

This section explains the underlying actions when the user presses either the 'save' or 'start' button in the web interface. The result of the click is an Ajax call making a HTTP POST to the Django server. There is a lot going on here and therefore split up in portions. The POST data can either be valid or invalid, resulting in to different operation blocks. If the data are valid, the *validated* block in the diagram is performed. If the data contains issues, the *errors* block is performed.

1. Step - Validated

All numbers refers to steps in Figure 5.3.
The server side code will recognize the incoming request as POST. Before validating the posted data, the system will update the already existing forms in the menu tree with new data and the wrappers accordingly (3). Further, all forms in the menu tree will be validated with the validation routine (6). Validated data are safe to feed the data items in the tree, and is done by the `_update_tree_data()` function (9). The menu tree now contains the new values from the POST, but we need to clean (11) the tree before saving it to a

binary file (13). The cleaning removes the wrappers and resets the extended tree back to normal state ready for saving. As the last step, before either starting the simulation or saving the values and returning to the web interface, we prepare the system by extending the tree with the forms (14) and flattening (16) it again.

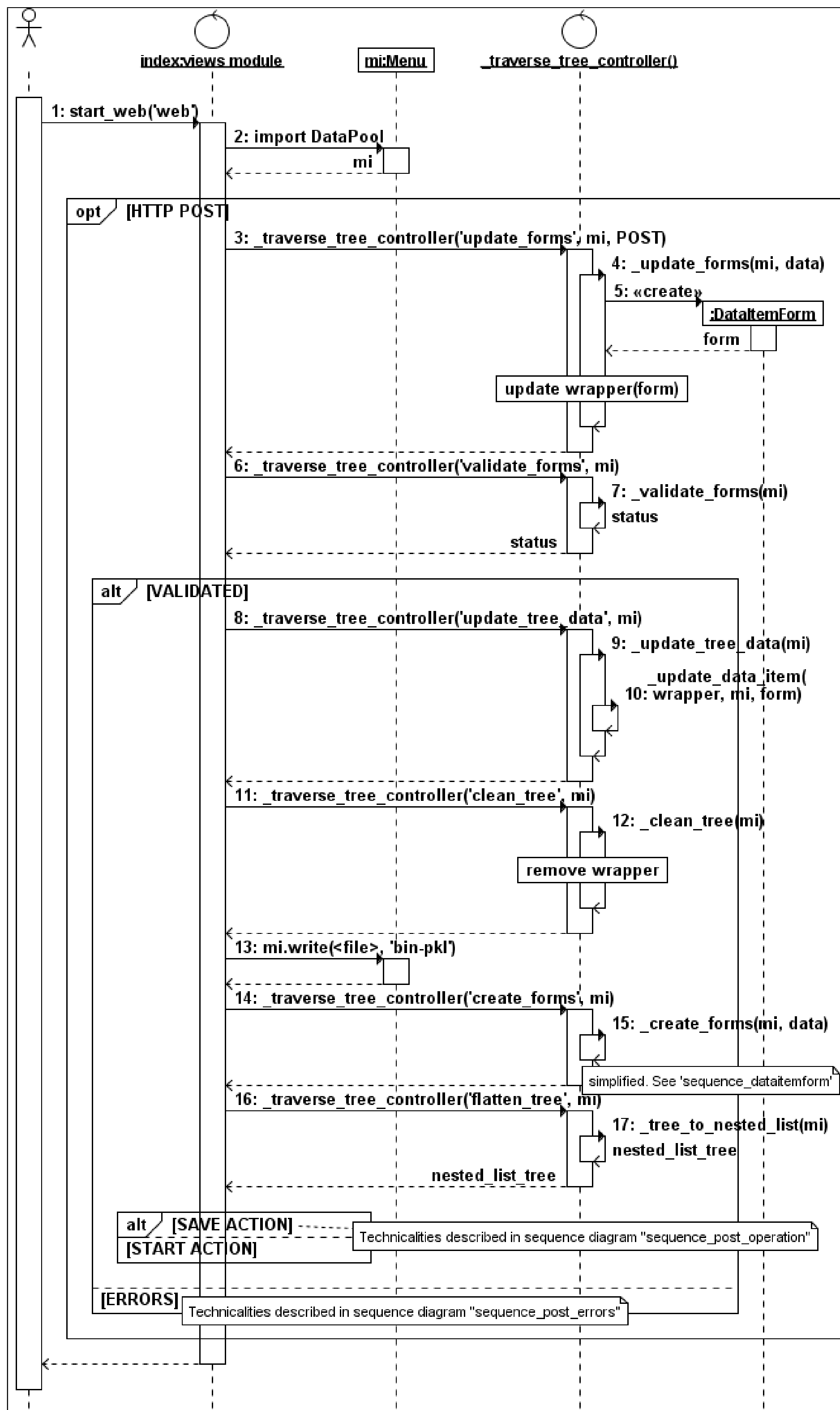

Figure 5.3: Program flow - Browser Interaction - Validated

2. Step - Operations

All numbers refers to steps in Figure 5.4.
If the post was only for saving the menu tree in an ongoing web session, the system will render the menu tree back to the web browser ending the request (3).

In the case of a simulation start, the server-side code will signaling the API and rest of the system the simulation code can continue its work (4). Then, the server-side goes into a rest, giving the simulation code time to finish (5). The resting will last until the server-side gets a signal back again. When the simulation code signals complete finish, the process goes on with checking for possible plots (6) and results (7) to be included in the final Ajax call to the web browser for visualization (8). This ends the request.

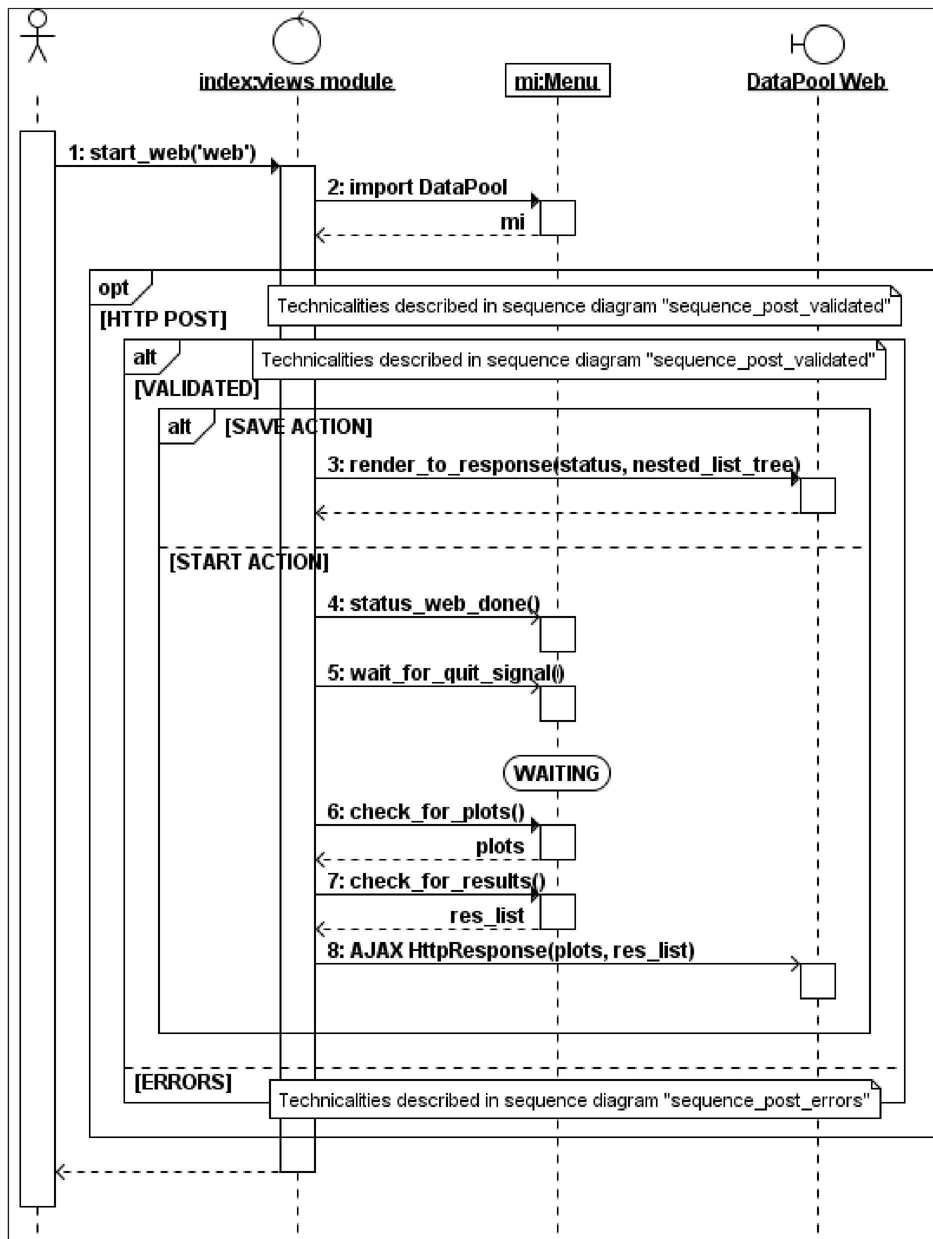

Figure 5.4: Program flow - Browser Interaction - Operations

Errors

All numbers refers to steps in Figure 5.5.
If the post data turned out to be invalid, the system will flatten the menu tree to a rendered nested list (3), create a global error list (4) for the administration panel in the web interface and then rendering it in the web browser (7).

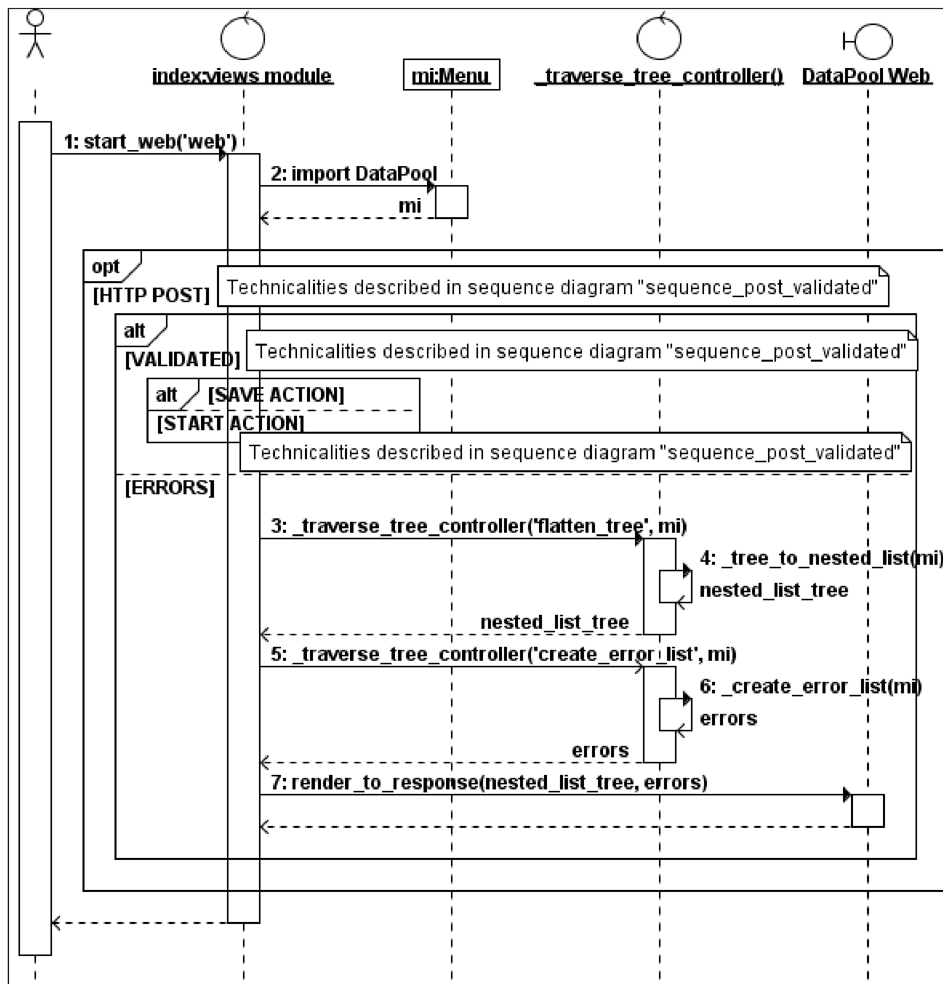

Figure 5.5: Program flow - Browser Interaction - Errors

5.6.4 System State Handling

All numbers refers to steps in Figure 5.6.
This subsection goes deep into the important state handling system involving the signals and the state files. New aspects introduced here is the role of the *.datapool storage* holding state and result files. The simulation start operation described earlier is also presented here with its lifeline put into the total context (10, 11, 16, 17). The sequence-state diagram shows the API initially saves the menu tree structure to binary (3) so the server-side can load this when started (4, 6). Then the API goes into waiting-state, letting the web interface get the lead (7). The API will continue and load the new menu tree (12), dumped to binary by the web again (10), when the server-side sends the `status_web_done()` signal (11).

The API will finish its `start_ui()` and let the simulation code perform the remaining steps before ending the code by sending the cleanup signal `quit()`

(13). The server-side will pick up this signal, return from the waiting-state (16), fetch generated plots and results (17), made by the cleanup signal, before finally making an Ajax call back to the web browser ending the whole process (18).

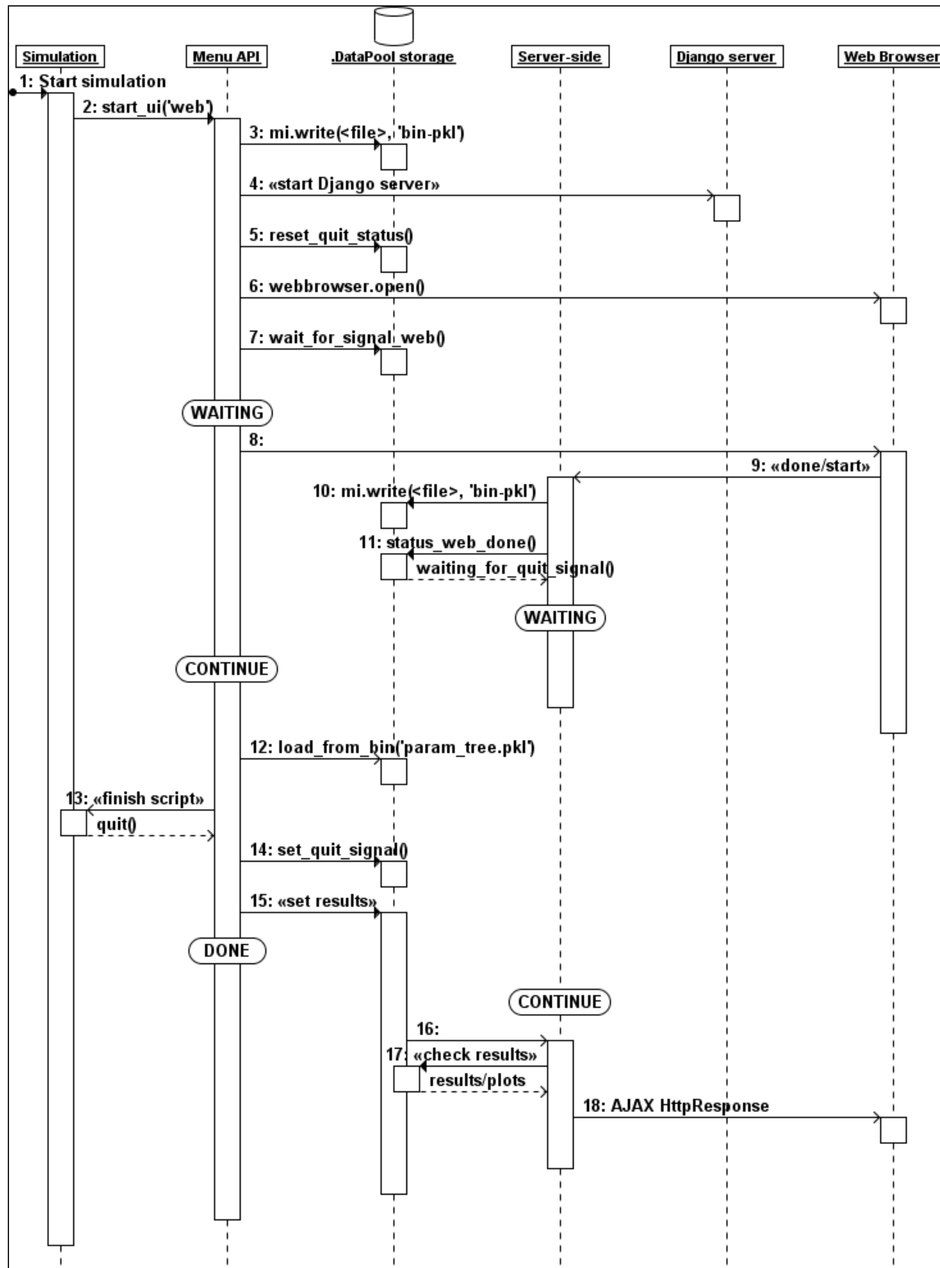

Figure 5.6: Program flow - System State Handling

5.6.5 Form Creation

All numbers refers to steps in Figure 5.7.
Now we go deeper into the very central internal process of creating the Django forms for all data items in the menu tree. This involves the longest functions measured in lines of code. The constructor of the form class (1) steer the process, while the class function `_create_field()` analyze the attributes of the data item and creates and sets up the correct fields for the corresponding form. A highly similar process is also performed when the system runs the form updating function.

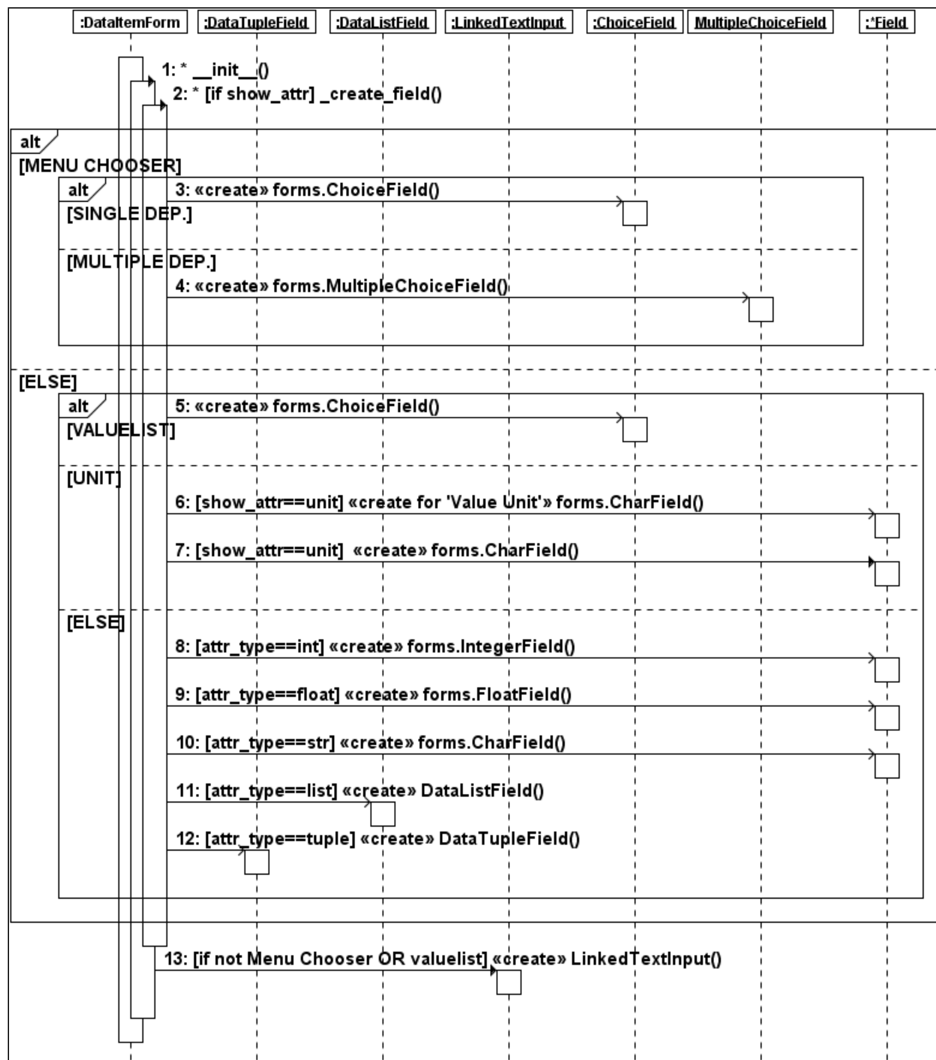

Figure 5.7: Program flow - Form Creation

5.6.6 Form Validation

All numbers refers to steps in Figure 5.8.
Django has a validation system following a certain convention. The normal procedure for doing data validation on a Django form is calling the form class method `is_valid()`. This starts the whole validation process which are divided into 3 actions. The term clean, is to validate data and send it along to the next step. The sequence:

1. Field type cleaning. This is the first part of the cleaning process. All fields are cleaned in a generic way for a field type. This is typical validation to check if the input type is correct, like float, int, string etc.

2. Field-specific cleaning. This gets the cleaned data from the field type cleaning. This step performs cleaning specific for the attribute. Hence, aspects not related to the type of field. If the first validation step succeeded and sent a valid string to this process, this step could be the place for checking special required patterns in a string or similar. It is important to state that this step will not be done on fields that did not pass the first step, but step 2 will always do validation for the cleaned ones from step 1 without breaking the sequence.

3. Form cleaning This final process starts when step 1 and 2 are completed on all the fields in the form. This step involves optional validation logic on all previous cleaned fields. This is the place to do dependency validation that spans several fields. This step is always performed.

In our implementation of DPW we have several fields to validate for each data item depending on the variating number of attributes in the form (2-11). Two field-specific clean functions (step 2) are also defined for the `value` attribute and the `minmax` attribute, respectively: `clean_value()` (12) and `clean_minmax()` (13). The form validation will as mentioned always be performed (14). In the implementation of DPW this results in the following sequence flow when form validation is performed on the whole menu tree:

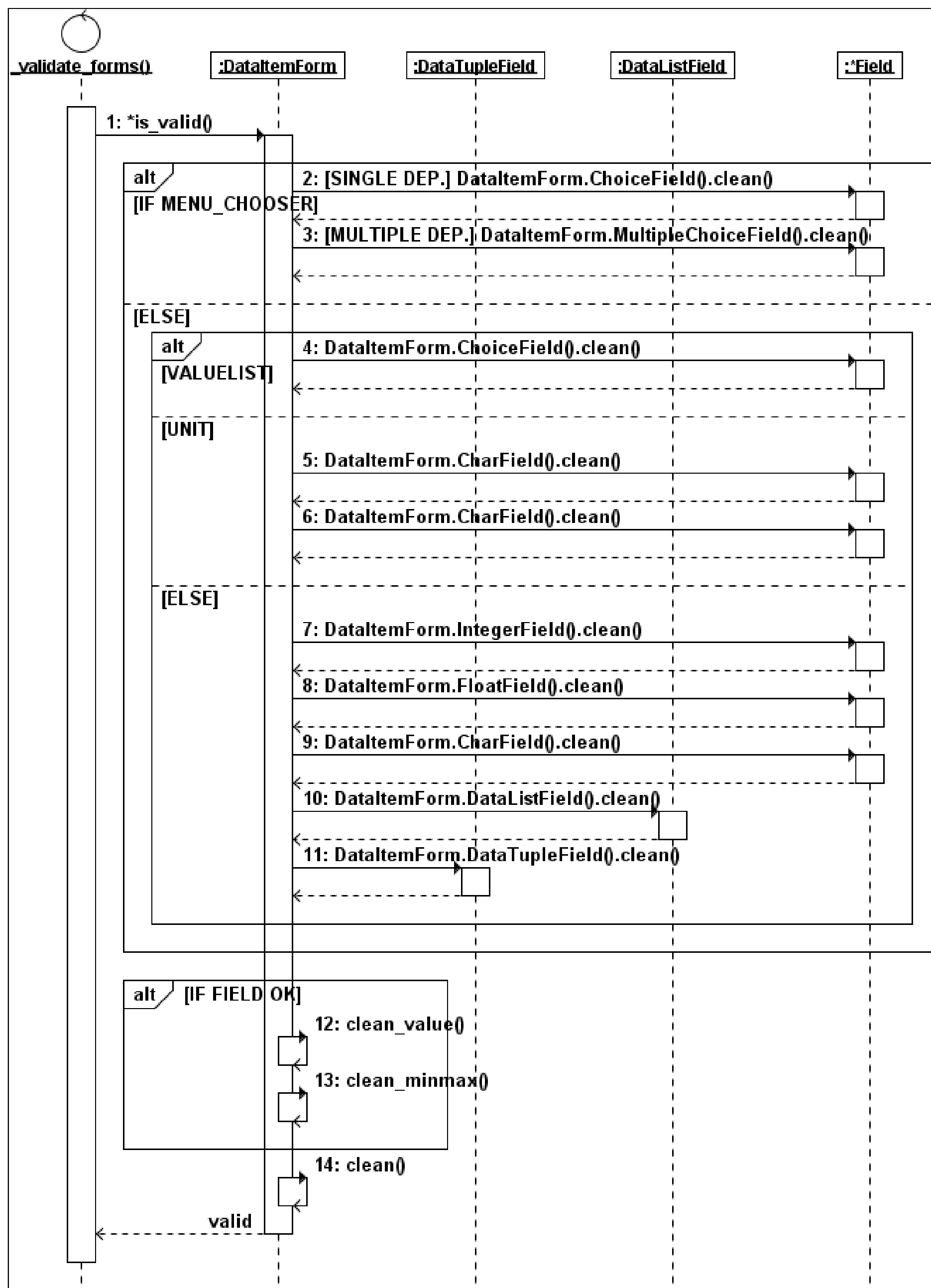

Figure 5.8: Program flow - Form Validation

5.6.7 Menu Tree Rendering

All numbers refers to steps in Figure 5.9.

We will now cover the last step in the server side view code before everything are rendered in the web browser. This process is the most transforming one in the whole system, and is a direct solution tailored to the built-in Django template

filter called `unordered_list`. This template filter provided by Django offers automatic generation of an unordered list. It takes a self-nested python list as input and automatically transforms it into a HTML unordered list without opening and closing the `<ul>` tags. In our base template, `template.html`, the filter is used on the menu tree variable sent from the view code with one single statement. This template code block is only needed in the template:

```
{{ data|unordered_list }}
```

A nested Python list of this structure:

```
['Items', ['Menu1', ['Data_item_1', 'Data_item_2'], 'Menu2']]
```

Would result in the following HTML using the template filter through the Django template engine:

```
<li>Items
<ul>
        <li>Menu1
        <ul>
                <li>Data_item_1</li>
                <li>Data_item_2</li>
        </ul>
        </li>
        <li>Menu2</li>
</ul>
</li>
```

Taking use of this powerful template filter, the implemented solution is tailored to take advantage of this feature. Following the usual recursive algorithm for traversing the menu tree, we build up a nested list structure for all the menu element objects in the tree (2). During the recursion, the form objects belonging to menu items and data items are rendered (3, 8) into HTML and appended to the nested list. The render methods are using the templates stored in the `templates` directory, and rendered ad-hoc (7, 12)) as the nested list is constructed. The procedure will for each menu item and data item define the context, by gathering the necessary data (4, 5, 6, 9, 10, 11), before rendering it. Finally the view code will call `render_to_response`, which is the built-in Django function that ends the session by sending the data to the web browser (13). In this last step, we take use of the convenient template filter in the base template, keeping the template exceptionally clean (14).

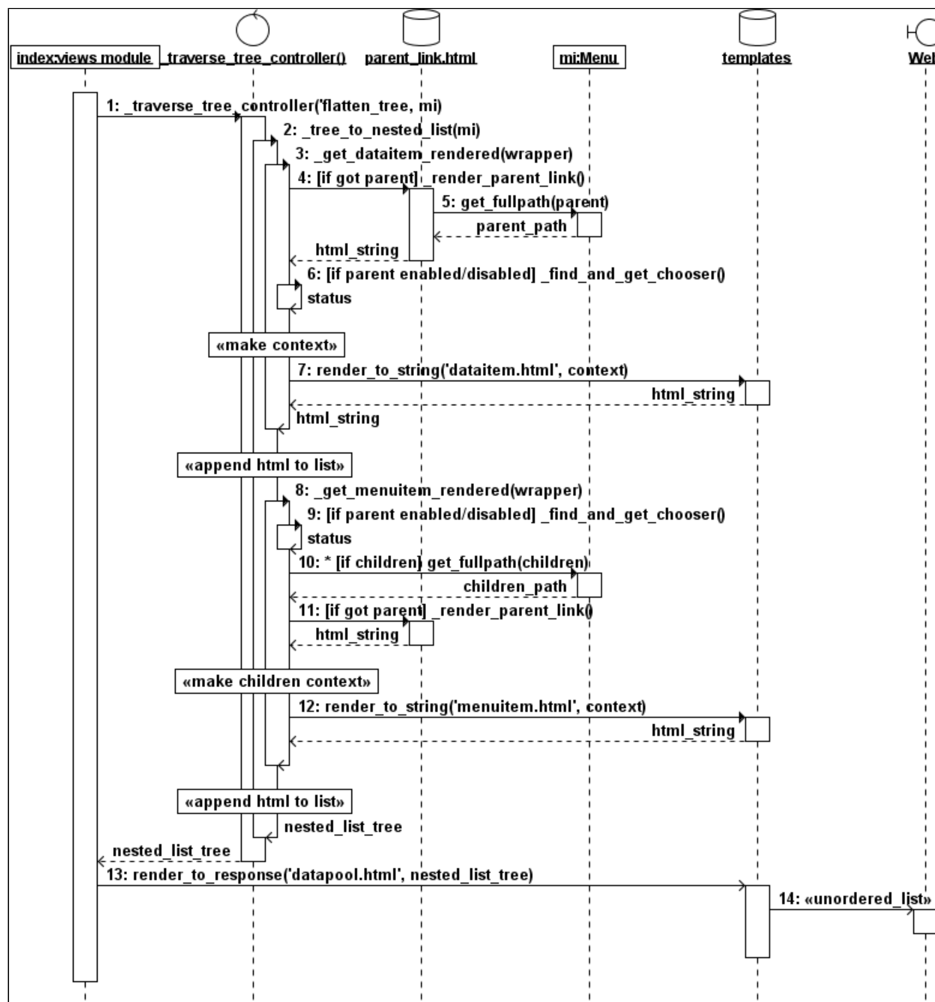

Figure 5.9: Program flow - Menu Tree Rendering

5.6.8 Interaction: JQuery

Four JavaScript files makes up the DPW implementation and have the responsibility for web interface interaction:

- `anchor.jquery.js` – Special code for animation scroll linking. A modified and tailored version of this script for our implementation.
- `datapoolweb.js` – The main DPW source code written from scratch handling all aspects of interaction.
- `jquery.livequery.js` – Special code for real-time update of DOM [4] tree when doing manipulation with the jQuery library (plug-in).
- `jquery-1.2.6.min.js` - The source code of the jQuery library.

The JavaScript handles different functionality in the web interface:

- Initiation of the tree state, and show the first submenu.
- Initiation of the tree state when errors are present.
- Show data item bodies with errors slided.
- Set expanded effect on all submenus in the path to data items with errors.
- The toggle sliding effect on data item bodies.
- Animation scroll effect (anchor linking).
- Ajax call for simulation start.
- Submitting the forms.
- Menu Chooser handling.
- Global tree functions.
- Expanding/collapsing submenus (and the collapse-symbol).
- Sliding submenu Quick-menus.

5.6.9 Alternatives

In the early start of the thesis a JavaScript library named EXT JavaScript [16] was investigated. A highly out-of-box widget based library with a solid appearance. However the library was too comprehensive in this context and gave a an unacceptable overhead to the rest. The functionality needed for DPW turned out to be hard to implement using this library. The main reason could be stated to be strong coupling to the predefined and lengthy built in objects. Emphasizing that EXT JavaScript is a powerful and outstanding library, but not optimal for our needed solutions.

For user input validation, the LiveValidation JavaScript library [17] for client side form field validation could be satisfying. It provides a large toolbox and platform for defining validation rules in JavaScript. This is very convenient for stopping errors before submitting forms, and will probably reduce the server load. Anyway, it was avoided in the end since it demands heavy embedding of dynamic JavaScript defined on server-side to be of any use in our implementation, resulting in even more work. The most obvious reason is to take use of the powerful Django validation, which is one of the main aspects of this thesis.

5.7 Variations in DataPool

The implementation of DPW comes with a slightly different modification compared to the original DataPool source code.

5.7.1 MenuAPI

`MenuAPI.py` file has several functions needed especially for DPW. However, this is according to the design principles of DataPool, the API should be exchangeable to fit different needs. All other functions in this file is similar to the original DataPool source code.

5.7.2 Settings

The settings files inclusion differs from the one in the DataPool source code.

Chapter 6

Alternative Framework: TurboGears

6.1 Intro

The contents described in this section concerns the TurboGears 1.0 version. During this thesis the framework released a 2.0 version in parallel. The TurboGears project exists today of two different versions, where the 1.0 version is still maintained for old users, and the 2.0 is fronted as the future road for the framework. The main difference between the editions are changes regarding the underlying modules of TurboGears. The concept of TurboGears which makes up its architecture, is the collection of the independent technologies and how they are related and used together. As of version 1.0 the involved constructs of the framework are:

SQLObject. A Object Relational Manager providing the object interface to the database with a Python-object-based query language. *SQLAlchemy [5] will be used as default in TurboGears $\geq$ 1.1.*

CherryPy. Object-oriented HTTP web application framework written in Python. Handles the data flow and HTTP requests in the web application. *TurboGears $\geq$ 1.1 will actually be using the third-largest Python web framework Pylons [30] using it as a middleware for back-end logic.*

Kid. A template engine where all templates are valid XHTML/XML. This gives the opportunity to open the template as simple XHTML files to check the design. The template language use a technique to embed Python code snippets inside the markup tags, like XML. *Genshi [29] is the successor to Kid in TurboGears $\geq$ 1.1.*

MochiKit. MochiKit is the preferred (optional) and default JavaScript library for TurboGears. The library is written to make development in JavaScript more Pythonic. In TurboGears it is mainly used for Ajax handling and widgets developed for the web framework.

6.2 Approach

Before trying to implement DPW using this framework the decided approach were to carry through this using the API function `walk()`. In this way making a proof-of-concept for developing alternative web interfaces using this generator function provided by the DataPool.

The chosen method for form creation was copied from the implementation made with Django. The inclusion and fetching of the binary file in order to make the internal menu tree was also done.

The TurboGears framework is different in many ways when it comes to the handling of the MVC pattern of an application. The most obvious differences from Django regards the URL handling and request mapping to the internal view functions. All view code is located in the `controllers.py` file. The file contains the different controller classes for the application. The required class are the `Root` class:

```
class Root(controllers.RootController):

    @expose(template='datapoolwebturbo.templates.index')
    def index(self, tg_errors=None):

        for f in static_tableform.fields:
            print f.render()

        return dict(forms=forms, submit="Save")



    @expose()
    @validate(form=static_tableform)
    @error_handler(index)
    def save(self, **data):
        """Handle submission from the form and save."""

        redirect(url('/'))
```

The framework has a strong concept of using Python decorators for different types of actions. The `@expose` decorator with a template path provided defines which template to use for the specific view function. The mapping of incoming URL requests to the right view functions is magically done behind the scenes. The framework will automatically find the function name at runtime, meaning a request for *http://localhost:8080/index*, will result in calling the `index` function of the root controller class. There are plenty of different validators that can be appended to the functions, making the connections to other functionality in the system, and validation and error handling are just examples (`@validate` and `@error_handler`).

6.3 Challenges

After getting an overview over the basic functions, concepts and approaches of the framework, the usual concepts were put into practice. However this turned

out to be a highly cumbersome task from the start. It seemed hard to copy the same approaches and concepts from the Django implementation using the typical and standard user patterns of the TurboGears framework. The decision were made to take a look at the source code of the framework in the hope of manage to subclass and tailor it for our need. This drove the process a little bit further and generated a form from the test data, though being very buggy.

Thoroughly study of the source code and long sessions of experiments lead to a solution using three different widgets in a type of connection. The implementation using the three widget in a connection are shown under sequential:

1. `widgets.TableForm`:

```
forms = _traverse_tree()
static_tableform = widgets.TableForm(fields=_create_static_form(forms),
                                     name='form')
```

2. `FormFieldsContainer`:

```
def _traverse_tree():
    prefix = 0
    forms = []

    # Create field widgets
    for item in mi.walk(mi.menu.root):
        for dataitem in item[2]:
            prefix += 1
            form_fields = widgets.FormFieldsContainer(
                            fields=_create_fields(dataitem, prefix))

            forms.append(form_fields)
    # Set validator
    _set_required_validator(forms, DataItem._required_attr)

    return forms
```

3. `widgets.TextField`:

```
def _create_fields(data_item, prefix):
    fields = []
    # Traverse the defined attributes to show and create fields
    for show_attr in data_item.attr_show_order:
        # 'attributes' dict where attribute belong
        if data_item.attributes.has_key(show_attr):
            # Only show attributes with a value set (not None)
            if data_item.attributes[show_attr] != None:
                fields.append(widgets.TextField(
                    name='form_'+show_attr+'_'+str(prefix),
                    label=show_attr,
                    default=data_item.attributes[show_attr]))

        # 'user_def' dict where attribute belong
        elif data_item.user_def_attr.has_key(show_attr):
            # Only show attributes with a value set (not None)
            if data_item.user_def_attr[show_attr] != None:
                fields.append(widgets.TextField(
```

```
                    name='form_'+show_attr+'_'+str(prefix),
                    label=show_attr,
                    default=data_item.attributes[show_attr]))

    return fields
```

Here is the outcome of the form considered semi-functional and not complete:

Figure 6.1: Form generated with TurboGears.

6.4 Conclusion

The TurboGears approach of attacking the same concepts of the Django implementation resulted in challenges. The result was neither satisfying regarding all the struggling behind it, proving the obviousness of TurboGears not being optimal for our needs. The framework speaks of itself of being an out-of-the-box framework focusing on rapid web application and the creation of Ajax power. We were able to make other small applications which were very easy to set up (not DataPool related), proving TurboGears on the other side to has succeeded in being production ready.

The conclusion is clear, TurboGears not being constructed to be flexible enough for our demands and not meant for this kind of use. Django is in addition to its application MVC, also designed to be easily extended in the means of subclassing. This is also reflected through Django's documentation,

which is not the case for TurboGears, which arguments further for the other characteristics for the latter. In the case of realizing web applications in the web 2.0 spirit with easy shortcuts for Ajax technicalities, TurboGears will stand for itself. The last point to be stated, are the higher level of *magic* in the TurboGears framework reflected through its different conventions. Django gives the user much more power in these situation. The magic behind the scenes in TurboGears hide details of the application, and this could be an argument to be more appropriate for developers who accept the conventions as long it serves their needs of creating typical web 2.0 applications. However, as mentioned in section 4.6 there has actually been made some widgets with the roots in Scientific Computing by using TurboGears, but this demands further research.

Chapter 7

Conclusion

This section summarizes the purpose of this thesis and presents our main contributions. Also, a possible future road for development is also proposed and described.

7.1 Summary

The motivation for this master thesis was to find a way of increasing the efficiency and usability when performing computational simulations, by extending and equipping the GUI module of the package DataPool with a highly visual, easy-to-use and powerful user interface. DataPool is a configurable Python package and tool for managing and controlling input data in simulation programs, by building up a menu tree structure for the involved parameters and attributes. The angling of a possible solution was set on realizing this with the use of Python web frameworks, since the package is used for simulation code written in Python. The thesis were to investigate the feasibility of obtaining a satisfying solution abiding loose coupling, extendability and being generic in a framework-based implementation. The starting point of evaluated Python web frameworks was Django, which also become the main focus and aim for the process. Other framework for Python like TurboGears was also up for evaluation and compared and reflected with the former. The thesis made an in-depth investigation and evaluation of the Django framework in order to follow a set of implementation guidelines in best possible manner.

7.2 Contributions

We have through a process with two thesis with a certain overlap been able to realize the creation of an user interface built on top and tailored for the DataPool package as an extension. The result is the user interface DPW (DataPool Web). DPW is a web-based menu system designed to present the internal tree structure defined through the DataPool package in the most usable and effective way according to user interaction. It focuses on user interaction, practical functions and visual communication in order to make the use of DataPool package easy and time saving in extensive and large computer simulations with a lot of parameters. The interface has the ability to present large amounts of data in

an effective and lucid manner for the user.

DPW was created with the help of the Django framework, and in this thesis has introduced the present situation of Python web frameworks and highlighted the two most important of them. We have ratified the increasing use of Python as a language in the web business shown though the arising trend of web frameworks. The importance of utilizing and focusing on web frameworks outside of the normal web-scope have been stated, and also proved to be of high benefits. The Django framework have clearly a remedy for inter- and intra crosscutting in web applications. In the context of using tools from the world of web 2.0 in the scientific computing domain, we decided to base the design and layout upon web 2.0 design elements to equip our solution with a modern and inspiring environment when working with complex simulation.

The creation of concepts of visual effects in order to increase efficiency have been done in DPW, in addition to introducing a visual function for controlling defined dependencies in the menu tree structure, built-in value unit conversion, input error handling, handling of Python type representations. The feature providing support for visualization of results in form of plots and numeric result, is making the DPW characteristics more a problem solving environment.

Furthermore, during our research and implementation some challenges were met along the road. The integration of a solution for a system state handling had to be made in order to make the communication between the different modules complete. The main reason for this challenge has the roots in the Django web server running in its own environment and Python context. This aspect will be up for further discussion in the section 7.3, covering possible future enhancements. We fulfilled all the implementation guidelines defined prior to the implementation, and the final technical implementation proved to be utilizing the core feature of Django to the fullest. Last, but not least, we succeeded in creating DPW without touching the core of the DataPool package. We also managed to keep the use of the DataPool API to a minimum. We can conclude the outcome of the thesis is an useful scientific web interface and a proof-of-concept regarding to both the use of Django and how it was implemented. Our final result is an interface implemented with a non-intrusive approach towards the DataPool package. On the other hand, our solution gives the users a a non-intrusive approach towards the simulation code utilizing DataPool. The simulation code is not aware of the DataPool package, and the latter is not aware of the DPW. Three systems built independently, but functioning perfectly together, using the exchangeable API as the glue.

Finally, as an alternative we evaluated the second most known Python web framework TurboGears. The usual concepts of the framework were put into practice, and turned out to be a highly non-productive task from the start. We were able to implement a simple prototype in the end, but the framework was quickly considered not being optimal for our needs. We have proven in this thesis that a framework stating to be equal and an out-of-the-box framework focusing on rapid web application and the creation of Ajax power, is not flexible enough and meant for our type of implementation.

7.3 Future Work

7.3.1 Visualization of Results

Future possibilities regarding DPW functionality could be extending its result visualization. We could introduce new API functions with more complex features for defining different simulation output. One suggestion is more interactive plots or matrices. A matrix could be holding different result quantities calculated on different series of moments in time. Those matrices could be operated hovering over them showing the different values generated from the simulation. Also, the plots could operated hovering over regions of the graph presenting the different values.

7.3.2 Web Service

DataPool is run locally with the web browser for the web interface, and the simulation systems involved are located together with DataPool on the same hardware. A future aspect of the DataPool system could be expanding the services towards a more distributed service system. This can be done connecting DataPool to existing web services with SOAP and WSDL [25, p. 94]. In this way DataPool could support a collection of SOAP web services in addition to the built-in technology it offers, without worrying about programming implementation of the services [7, p. 94]. This kind of SOA support could be easily plugged in and tailored for users with special needs. In this way, as Puppin and Tonelotto stating: Scientists are not anymore bound to a single machine (or a single cluster of machines) and a single source of data.... [28, p. 1].

In the coming era with cloud computing, the obvious enhancement of DataPool and DPW is to transform them into web services. Moving everything over to a stand-alone server offering its own DataPool web service. The system could handle multiple users in time sharing several simulation programs in collaboration. The web service could offer an intuitive and highly interactive in-line editor for the hosted simulation programs. Further, the solution could provide drag-and-drop functionality for defining the menu tree structure. Finally, simulation results could be embedded into a comprehensive visualization engine in the same workspace. With other words, a PSE web service, offering the DataPool functionality without the need for local configuration and installation.

7.3.3 Integration with the FEniCS Project

DataPool/DPW could also be tailored and integrated for the FEniCS Project [26]. FEniCS is a free software for automated solution of differential equations, involving tools for working with computational meshes, finite element variational formulations of PDEs, ODE solvers and linear algebra. Since FEniCS is based on a collection of interoperable components/projects, DataPool could be a part of this software. DOLFIN mentioned in section 1.1, is the C++/Python interface of FEniCS, providing a consistent PSE (Problem Solving Environment) for ordinary and partial differential equations and DataPool could be tailored and function as a complement.

Bibliography

[1] *Python for scientific gateways development.* Grid Computing Environments (GCE), 2007.

[2] Wes Bethel, Cristina Siegrist, John Shalf, Praveenkumar Shetty, T.J. Jankun-Kelly, Oliver Kreylos, and Kwan-Liu Ma. Visportal: Deploying grid-enabled visualization tools through a web-portal interface. In *Proceedings of 3rd Annual Workshop on Advanced Collaborative Environments*, June 2003.

[3] deal.II Official Website. `http://www.dealii.org/`.

[4] W3C Document Object Model (DOM). `http://www.w3.org/DOM/`.

[5] SQLAlchemy The Database Toolkit for Python. `http://www.sqlalchemy.org/`.

[6] CherryPy framework. `http://www.cherrypy.org/`.

[7] Nathan Holmberg, Burkhard Wünsche, and Ewan Tempero. A framework for interactive web-based visualization. In *AUIC '06: Proceedings of the 7th Australasian User interface conference*, pages 137–144, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[8] TurboGears Homepage. `http://www.turbogears.org/`.

[9] T. J. Jankun-Kelly, Oliver Kreylos, Kwan-Liu Ma, Bernd Hamann, Kenneth I. Joy, John Shalf, and E. Wes Bethel. Deploying web-based visual exploration tools on the grid. *IEEE Comput. Graph. Appl.*, 23(2):40–50, 2003.

[10] jQuery JavaScript Library. `http://jquery.com/`.

[11] XML-based Templating Kid Pythonic. `http://www.kid-templating.org/`.

[12] Sergei Kojarski and David H. Lorenz. Domain driven web development with webjinn. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 53–65, New York, NY, USA, 2003. ACM.

[13] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming.* Texts in Computational Science and Engineering, vol 1. Springer, second edition, 2003.

[14] H. P. Langtangen. *Python Scripting for Computational Science*. Springer, second edition, 2005.

[15] H. P. Langtangen. *Python Scripting for Computational Science*. Texts in Computational Science and Engineering, vol 3. Springer, third edition, 2009.

[16] EXT JavaScript library. `http://www.extjs.com/`.

[17] LiveValidation JavaScript library. `http://www.livevalidation.com/`.

[18] MochiKit A lightweight Javascript library. `http://mochikit.com/`.

[19] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.

[20] Rustam Mehmandarov. *DataPool: A Tool for Handling Input Data in Simulation Programs*. Master's thesis, University Of Oslo and Simula Research Laboratory, 2009.

[21] D. Moore, R. Budd, and W. Wright. *Professional Python Frameworks: Web 2.0 Programming with Django and TurboGears*. Wrox, 2007.

[22] B. Nron, P. Tuffry, and C. Letondal. Mobyle: a web portal framework for bioinformatics analyses. 2008.

[23] Boost C++ Libraries Program Options. `http://www.boost.org/doc/libs/1_39_0/doc/html/program_options.html`.

[24] SQLObject ORM. `http://www.sqlobject.org/`.

[25] Marlon Pierce and Geoffrey Fox. Making scientific applications as web services. *Computing in Science and Engg.*, 6(1):93–96, 2004.

[26] FEniCS Project. `http://www.fenics.org/wiki/FEniCS_Project`.

[27] The Django Project. `http://www.djangoproject.com/`.

[28] Diego Puppin, Nicola Tonellotto, and Domenico Laforenza. How to run scientific applications over web services. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 29–33, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Genshi Python toolkit for generation of output for the web. `http://genshi.edgewall.org/`.

[30] Pylons Python web framework. `http://pylonshq.com/`.

[31] DOLFIN Official Website. `http://www.fenics.org/wiki/DOLFIN/`.