Software Tools and Efficient Algorithms for the Feature Detection, Feature Tracking, Event Localization, and Visualization of Large Sets of Atmospheric Data

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität Mainz

vorgelegt von
Sebastian Limbach
geboren am 26. September 1981 in Mainz

Mainz, den 18.06.2013

Berichterstatter:

Tag der mündlichen Prüfung:

19. August 2013

D77 – Mainzer Dissertation

Abstract

Data sets describing the state of the earth's atmosphere are of great importance in the atmospheric sciences. Over the last decades, the quality and sheer amount of the available data increased significantly, resulting in a rising demand for new tools capable of handling and analysing these large, multidimensional sets of atmospheric data. The interdisciplinary work presented in this thesis covers the development and the application of practical software tools and efficient algorithms from the field of computer science, aiming at the goal of enabling atmospheric scientists to analyse and to gain new insights from these large data sets. For this purpose, our tools combine novel techniques with well-established methods from different areas such as scientific visualization and data segmentation. In this thesis, three practical tools are presented. Two of these tools are software systems (Insight and IWAL) for different types of processing and interactive visualization of data, the third tool is an efficient algorithm for data segmentation implemented as part of Insight.

Insight is a toolkit for the interactive, three-dimensional visualization and processing of large sets of atmospheric data, originally developed as a testing environment for the novel segmentation algorithm. It provides a dynamic system for combining at runtime data from different sources, a variety of different data processing algorithms, and several visualization techniques. Its modular architecture and flexible scripting support led to additional applications of the software, from which two examples are presented: the usage of Insight as a WMS (web map service) server, and the automatic production of a sequence of images for the visualization of cyclone simulations.

The core application of Insight is the provision of the novel segmentation algorithm for the efficient detection and tracking of 3D features in large sets of atmospheric data, as well as for the precise localization of the occurring genesis, lysis, merging and splitting events. Data segmentation usually leads to a significant reduction of the size of the considered data. This enables a practical visualization of the data, statistical analyses of the features and their events, and the manual or automatic detection of interesting situations for subsequent detailed investigation. The concepts of the novel algorithm, its technical realization, and several extensions for avoiding under- and over-segmentation are discussed. As example applications, this thesis covers the setup and the results of the segmentation of upper-tropospheric jet streams and cyclones as full 3D objects.

Finally, IWAL is presented, which is a web application for providing an easy interactive access to meteorological data visualizations, primarily aimed at students. As a web application, the needs to retrieve all input data sets and to install and handle complex visualization tools on a local machine are avoided. The main challenge in the provision of customizable visualizations to large numbers of simultaneous users was to find an acceptable trade-off between the available visualization options and the performance of the application. Besides the implementational details, benchmarks and the results of a user survey are presented.

Zusammenfassung

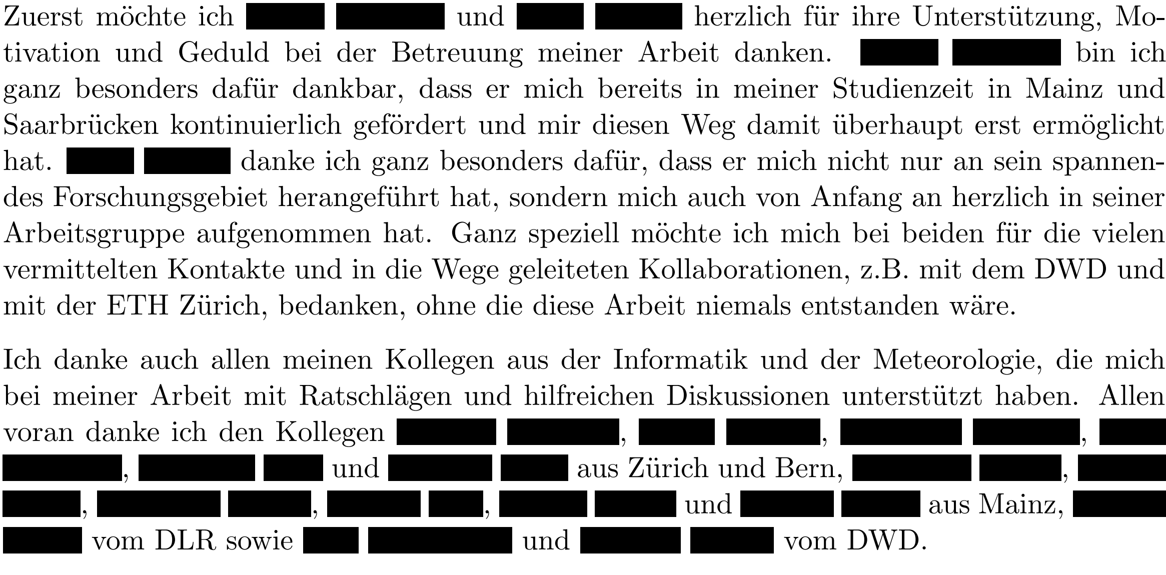
In den Atmosphärenwissenschaften spielen Datensätze, die den Zustand der Atmosphäre beschreiben, eine wichtige Rolle. Im Verlauf der letzten Jahrzehnte stieg die Qualität, aber auch die Anzahl verfügbarer Daten in diesem Bereich deutlich an, was zu einem wachsenden Bedarf an geeigneten Werkzeugen zur Verarbeitung und Analyse umfangreicher, mehrdimensionaler Datensätze führte. Die vorliegende interdisziplinäre Arbeit beschreibt die Entwicklung und Anwendung effizienter Algorithmen und Softwaretools aus dem Bereich der Informatik mit dem Ziel, Atmosphärenwissenschaftlern die Analyse und das Gewinnen neuer Erkenntnisse aus solchen großen Datensätzen zu ermöglichen. Die vorgestellten Werkzeuge kombinieren etablierte und neuartige Techniken aus den Bereichen der wissenschaftlichen Visualisierung und der Datensegmentierung. Insight und IWAL sind komplexe Softwaretools für die interaktive Visualisierung und die Verarbeitung atmosphärischer Daten, das dritte Werkzeug ist ein in Insight implementierter Algorithmus zur Datensegmentierung.

Insight ist ein Softwaretool für die interaktive 3D-Visualisierung und die Verarbeitung großer atmosphärischer Datensätze mit dem ursprünglichen Ziel, eine Testumgebung für den Segmentierungsalgorithmus zu bieten. Insight erlaubt zur Laufzeit das dynamische Kombinieren von Datenquellen, Algorithmen zur Datenverarbeitung und verschiedener Visualisierungstechniken. Der modulare Aufbau und eine flexible Skriptsteuerung ermöglichen weitere Anwendungen der Software, von welchen zwei Beispiele näher betrachtet werden: die Realisierung eines WMS (Web Map Service) Servers und die automatische Erzeugung einer Animation für die Darstellung simulierter Zyklonendaten.

Ein Hauptmerkmal von Insight ist die Bereitstellung des neuartigen Segmentierungsalgorithmus für die effiziente Erkennung und zeitliche Verfolgung interessanter 3D-Merkmale in großen Datensätzen und für die genaue Lokalisierung der auftretenden Genesis-, Lysis-, Verschmelzungs- und Auftrennungs-Ereignisse. Die Segmentierung führt in der Regel zu einer drastischen Reduktion der relevanten Datenmenge und ermöglicht so eine zweckmäßige Visualisierung, statistische Analysen und die manuelle und automatische Erkennung interessanter Situationen in den Daten, die sich für weitere Untersuchungen eignen. Das Konzept und die Implementierung des Algorithmus sowie Erweiterungen zur Vermeidung von Über- und Untersegmentierungen werden beschrieben und zwei Beispielanwendungen zur Segmentierung von Jet-Streams und Zyklonen als vollständige 3D-Objekte werden präsentiert.

Abschließend wird IWAL vorgestellt, eine in erster Linie für Studenten entwickelte Webanwendung, die eine einfache, interaktive Visualisierung atmosphärischer Daten ermöglicht, ohne dass sich der Anwender um deren Beschaffung oder die Installation und Verwendung komplexer Visualisierungstools kümmern muss. Um einer großen Anzahl gleichzeitiger Nutzer anpassbare Visualisierungen bereitstellen zu können, musste ein Kompromiss zwischen den verfügbaren Darstellungsoptionen und der Performance der Software gefunden werden. Die Details der Implementierung, abschließende Benchmarks und die Ergebnisse einer Nutzerbefragung werden vorgestellt.

Danksagung



Zuerst möchte ich ███ ███ und ███ ███ herzlich für ihre Unterstützung, Motivation und Geduld bei der Betreuung meiner Arbeit danken. ███ ███ bin ich ganz besonders dafür dankbar, dass er mich bereits in meiner Studienzeit in Mainz und Saarbrücken kontinuierlich gefördert und mir diesen Weg damit überhaupt erst ermöglicht hat. ███ ███ danke ich ganz besonders dafür, dass er mich nicht nur an sein spannendes Forschungsgebiet herangeführt hat, sondern mich auch von Anfang an herzlich in seiner Arbeitsgruppe aufgenommen hat. Ganz speziell möchte ich mich bei beiden für die vielen vermittelten Kontakte und in die Wege geleiteten Kollaborationen, z.B. mit dem DWD und mit der ETH Zürich, bedanken, ohne die diese Arbeit niemals entstanden wäre.

Ich danke auch allen meinen Kollegen aus der Informatik und der Meteorologie, die mich bei meiner Arbeit mit Ratschlägen und hilfreichen Diskussionen unterstützt haben. Allen voran danke ich den Kollegen ███ ███, ███ ███, ███ ███, ███ ███, ███ ███ und ███ ███ aus Zürich und Bern, ███ ███, ███ ███, ███ ███, ███ ███, ███ ███ und ███ ███ aus Mainz, ███ ███ vom DLR sowie ███ ███ und ███ ███ vom DWD.

Ganz herzlich möchte ich mich auch bei allen anderen Mitarbeitern am Institut für Informatik, am IPA in Mainz und am IACETH in Zürich für die großartige Arbeitsatmosphäre bedanken, die die letzten Jahre zu einem ganz besonderen Abschnitt meines Lebens gemacht haben.

Last but not least bedanke ich mich bei meiner Frau ███, bei meinen Eltern und bei meiner ganzen Familie. Ihr habt mich während der letzten Jahre geduldig unterstützt und mir Rückhalt gegeben.

The Insight project has been funded by the *Center for Computational Sciences in Mainz*. `http://www.csm.uni-mainz.de`.

The IWAL project was financed by *Innovedum*, ETH Zurich. `http://www.innovedum.ethz.ch`.

Project websites:
`http://insight.zdv.uni-mainz.de/trac`
`http://insight.zdv.uni-mainz.de/iwal/trac`
`http://iwal.ethz.ch`

Contents

1. Introduction **13**

2. Insight **17**
2.1. Introduction . . . 17
2.1.1. Motivation . . . 19
2.1.2. Features . . . 21
2.1.3. Examples of use . . . 23
2.1.4. Insight's predecessor . . . 24
2.1.5. Outline . . . 25
2.2. Data processing concept . . . 27
2.2.1. Introduction . . . 27
2.2.2. Data dimensions . . . 29
2.2.3. Data variables . . . 30
2.2.4. Variable and dimension requests . . . 31
2.2.5. Data entities . . . 31
2.2.6. Data compositions . . . 33
2.3. Data visualization . . . 33
2.3.1. Transformation types . . . 34
2.3.2. 3D data on a (deformed) regular grid . . . 36
2.3.3. Subsets of 3D data on a (deformed) regular grid . . . 36
2.3.4. Trajectories . . . 39
2.4. Software architecture . . . 39
2.4.1. Overview . . . 40
2.4.2. MVC concept . . . 40
2.4.3. Third-party components . . . 43
2.5. Realization of the data processing concept . . . 45
2.5.1. Data processing models overview . . . 46
2.5.2. The DataDimension class . . . 46
2.5.3. The DataVariable class . . . 50
2.5.4. The DataGrid and PosProvider classes . . . 51
2.5.5. The DataEntity class . . . 52
2.5.6. The DataComposition class . . . 57
2.5.7. Data processing controllers, factories, and views . . . 58
2.6. NetCDF file format handling . . . 61
2.6.1. Description of the netCDF format . . . 61
2.6.2. NetCDF import . . . 61
2.6.3. NetCDF export . . . 65
2.7. Scripting support . . . 66
2.7.1. The user's perspective . . . 67
2.7.2. Technical realization . . . 77

2.8. Realization of the serialization 82
2.8.1. Serialization using XML files 83
2.8.2. Serialization using Python scripts 83
2.9. Example use case: Visualization of cyclone simulations 85
2.9.1. Introduction 85
2.9.2. Input data 85
2.9.3. Realization 87
2.10. Outlook 90

3. Feature segmentation, event localization **93**
3.1. Introduction 94
3.1.1. Identification and tracking of atmospheric flow features 94
3.1.2. Conceptional view on feature identification and tracking 95
3.2. Foundations of the algorithm 97
3.2.1. Input 97
3.2.2. Output 98
3.2.3. Feature detection predicates 101
3.3. Segmentation algorithm 103
3.3.1. Feature detection 103
3.3.2. Feature tracking 104
3.3.3. Event localization 105
3.3.4. Efficiency 107
3.4. A climatology of upper-tropospheric jet streams and their events 108
3.4.1. A Rossby wave breaking event over the North Atlantic 108
3.4.2. Frequency of jets, jet merging, and jet splitting 110
3.4.3. Lifetime and stability of jet segments 112
3.5. Extensions of the algorithm 113
3.5.1. Double thresholding 114
3.5.2. Extended sub-segment detection 117
3.5.3. Feature dilation 119
3.6. Identification and tracking of cyclones 119
3.6.1. 3D segmentation criterion 120
3.6.2. Improved sub-segment assignment 121
3.6.3. Results 122
3.7. Outlook 126
3.8. Conclusions 127

4. Interactive Weather Analysis Laboratory (IWAL) **129**
4.1. Introduction 129
4.1.1. Outline 130
4.1.2. Features 131
4.1.3. Related work 133
4.2. Technical building blocks 134
4.2.1. Django and South 135
4.2.2. PostgreSQL 135
4.2.3. PyNGL and PyNIO 136
4.2.4. Apache and mod_wsgi 136
4.2.5. HTML5, JavaScript, and JQuery 137

4.2.6. JSON 137
4.2.7. WMS 138
4.3. Server architecture 138
4.3.1. Django basics 139
4.3.2. The `wms` application 139
4.3.3. The `wmsclient` application 147
4.3.4. Database design 149
4.4. Client architecture 153
4.4.1. User, course, and case study management 154
4.4.2. WMS client basics 155
4.4.3. WMS client models 157
4.4.4. WMS client views 161
4.4.5. WMS client controllers 163
4.4.6. Creation and management of plot data 165
4.4.7. Serialization of program states 167
4.5. Results 169
4.5.1. Performance and Benchmarks 170
4.5.2. User survey 175
4.6. Outlook 180

5. Outlook and Conclusion **183**
5.1. Achieved goals 183
5.2. Outlook 184
5.3. Conclusion 186

A. Installing and launching of Insight **187**
A.1. Installing Insight 187
A.2. Launching of Insight 189

B. Reference of the data entities of Insight **191**
B.1. Data sources 192
B.2. Data processors 196
B.3. Visual representations 210
B.4. Other entities 218

C. Reference of the Insight script interface **219**
C.1. The `insight_core` module 219
C.2. The `insight` package 227

D. Example script for a simple Insight-WMS-server **237**

E. Full script for the idealized cyclones case study **241**

F. Maintenance of IWAL **247**
F.1. Deployment of IWAL 247
F.2. Example database migration 249

1. Introduction

Since the advent of the first major computer systems, one important application of these systems was the processing of large meteorological data sets. Numerical weather prediction was already performed by the first electronic general-purpose computer, *ENIAC*, since the 1950s (Charney et al., 1950). Along with new developments in the field of electronic data processing, further technological progress led to rapidly growing numbers of available meteorological data. For example, *Tiros 1*, the first fully operational weather satellite and pioneer of a whole generation of satellites, was launched on 01 April 1960 (cf. Fritz, 1964). Further improvements in the areas of data acquisition and analysis, as well as the exponential growth of available computational power and memory, fueled this development over the last decades. An effective handling, understanding, and analysis of such large amounts of data is only possible with the support of software tools and efficient underlying algorithms.

One important interdisciplinary approach for the handling and for the extraction of new insights out of large, multidimensional data sets is *scientific visualization*. This relatively new discipline (Johnson, 2004, describes it as being "launched" in 1987) combines techniques from many different fields, including computer graphics and image processing, and even user interface studies (McCormick et al., 1987). The impact of scientific visualization was described by McCormick et al. (1987) as follows:

> *Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to* observe *their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science.*

An important task of the (manual and automatic) processing of large data sets, not limited to the field of scientific visualization, is the reduction of the data to the parts of interest, with respect to the particular application. *Data segmentation* is a common method in different scientific areas, for example in image processing and computer vision (Zucker, 1976; Nikhil and Sankar, 1993; Jain et al., 1995; Davies, 2005), as well as in flow visualization (cf. Post et al., 2003), for achieving such a reduction of the data. In the context of atmospheric data, segmentation enables a detection and the tracking over time of distinct instances of interesting atmospheric phenomena. The present work covers the development of appropriate software tools and algorithms for the effective handling of large sets of atmospheric data, with a focus on the development of a novel data segmentation method.

In this introductory chapter, an overview of the topics and objectives of the present work is given. This thesis consists of three major building blocks. Two of the main chapters deal with the design, implementation and example applications of two software tools for the processing and visualization of atmospheric data. One of them is *Insight*, a desktop application for data processing, segmentation, and visualization; the other is *IWAL*, a web application primarily aimed at students, providing the easy and interactive creation of a

wide range of meteorological plots. The third building block is a novel algorithm for the detection and tracking of interesting features in large sets of atmospheric data, together with a localization of the occurring genesis, lysis, merging, and splitting events.

In Chapter 2, the Insight software project is presented. Insight is the successor of the visualization software Vis3d (Marto, 2008). It was primarily intended to provide a testing environment for our new segmentation algorithm. Due to its modularity, adaptability, and ease of use (compared to many existing tools for the two- and three-dimensional visualization of atmospheric data), the software was additionally used by different scientists and students for their own work and studies. The chapter on Insight starts with a motivation, a list of Insight's features, and a short discussion of similar software tools. Furthermore, examples for the application of Insight, mostly in the context of the work of Master and PhD students from different institutions, are presented. The main part of the chapter covers the concept and realization of the software, with a focus on the realization of Insight's flexible data processing pipeline. Insight supports script control through an embedded Python interpreter and a custom Python module. The details of this scripting support of Insight are presented and an example for the application of a Python script for the creation of a rudimentary Insight-based WMS server is given. The chapter closes with the presentation of the application of Insight in the context of the work of Schemm et al. (2012). Schemm used Insight for the visualization of his simulations of the formation and intensification of cyclones under dry and moist conditions.

As mentioned above, Insight serves as the framework for the implementation of a novel feature detection and tracking algorithm. This algorithm is presented in detail in Chapter 3. The chapter contains a modified version of our article published in the Geoscientific Model Development (GMD) journal (Limbach et al., 2012). The core article was expanded by additional sections describing the recent extensions and additional applications of the algorithm. Originally motivated by an approach from Siegesmund (2006) for the segmentation of two-dimensional ozone holes and their development over time, we combined ideas from different fields, for example from image segmentation and flow visualization, for the first formulation of our algorithm. The algorithm is capable of performing an efficient segmentation and tracking of three-dimensional objects and their development over time. One of the new aspects of the algorithm is that the full available spatiotemporal information is used for the feature detection and tracking. Additionally, the algorithm is capable of localizing the occurring genesis, lysis, splitting, and merging events on a per-grid-point basis. For an efficient realization of these new features, appropriate algorithmic concepts involving the usage of efficient data structures had to be developed. The chapter starts with an introduction of the new method and a comparison to existing approaches. Afterwards, the idea behind the algorithm and its realization are described in detail, and the usefulness and efficiency is demonstrated by means of a case study involving the computation of a two-years climatology of jet streams. We subsequently applied the algorithm to further atmospheric phenomena, in form of joint works with fellow students and meteorologists from the University of Mainz and the ETH Zurich. In this context, we extended and refined the algorithm and its implementation in order to avoid common problems such as over- and under-segmentation. These extensions are presented, followed by the presentation of first results of a survey on cyclones performed in collaboration with Heini Wernli and Lukas Papritz (ETH Zurich). The chapter ends by giving an outlook on possible future developments and a summary of the achieved goals.

Chapter 4 covers the planning, development, and the outcome of the IWAL software project. The project was realized over the course of one year, financed by Innovedum[1], a fund at the ETH Zurich for projects with the goals of improving the teaching and establishing new methods of learning. IWAL is a web application, allowing students of meteorology an easy access to a wide range of different types of visualizations of atmospheric data. The development was influenced by our experiences with Insight. Many concepts were taken over, and Insight's visualization capabilities were used in an early stage of the program for the creation of the plots on the server-side. IWAL is based upon a range of open protocols and (web-)technologies. At its core, IWAL implements both a WMS (Web Map Service) server and client for data interchange. The details of both the server and the client implementation are presented in this chapter. In contrast to existing web applications for the visualization of two-dimensional data sets (e.g. Google Maps), the creation and transmission of plots on the basis of three-dimensional data sets poses a special challenge to our implementation. In this chapter, we explicate the difficulties in finding a trade-off between the visualization options offered to the user and the server load. We evaluated the impact of several caching strategies by means of a series of benchmarks. Additionally, we present the results of a user survey that was carried out with the first group of students using IWAL in class. At the end of this chapter, an outlook on possible future enhancements and improvements of the software is given.

At the end of this thesis, Chapter 5 provides a summary of the accomplished tasks and a brief discussion of the general challenges and chances we experienced in the course of this interdisciplinary work. Finally, we provide an outlook on some of the most important future developments and extensions of the presented tools.

[1] cf. `http://www.innovedum.ethz.ch`, last accessed: 19-September-2012

2. Insight - A Visualization and Segmentation Toolkit for Atmospheric Data

2.1. Introduction

Whenever we study a physical system like the atmosphere, the properties and the state of the system are typically described by means of *data*. In the atmospheric sciences, data occurs in many different forms. Commonly, the data represents different parameters of the atmosphere as numerical values associated with fixed points in time and space. Some of this data is obtained through measurements, but since most weather phenomena have a fairly large spatial extension, the available measured data is often too sparse. Therefore, the data has to be processed in such cases, for example by applying data assimilation and interpolation techniques. Other examples of data in the context of atmospheric sciences are sets of idealized data coming from model simulations, forecast data, and secondary data such as statistical data and physical quantities derived from other variables.

Computer-aided *visualization* plays an important role in presenting and understanding scientific data (cf. McCormick et al., 1987). A standard personal computer of today has the capability to process and to visualize data sets in ways that were unthinkable or impractical some decades ago. Back then, atmospheric scientists were restricted to the production of limited sets of manually drawn maps and diagrams, and later to the output of computers with moderate graphical capacities. Figure 2.1 shows a photograph of the two meteorologists Tor Bergeron and Jacob Bjerknes (members of the "Bergen school of meteorology" that was co-founded by Jacob Bjerknes' father Vilhelm Bjerknes, cf. Friedman, 1993) working on the creation of a hand-drawn map in the year 1919. Figure 2.2 shows an example visualizations of idealized cyclones by Bjerknes and Solberg from the same year. An early three-dimensional visualization of a PV (potential vorticity) anomaly from the year 1950 by Ernst Kleinschmidt (Kleinschmidt, 1950) can be found in Fig. 2.3. The drawing also contains further examples of visualizations, namely coastlines and isolines.

Although these hand-drawn visualizations may have their individual advantages, a computer running appropriate software significantly improves the quantity and quality of possible data visualizations. Beyond that, computer aided visualization tools are often capable of interactive manipulations and adaptions of the visualization, allowing for an interactive exploration of multidimensional input data sets. The handling and visualization of time series is also a common task accomplished by these software tools, often realized in form of batch processing of the input data. Data visualization aims at pointing out important and interesting aspects of the visualized data sets to the user. For this, visualization methods often act as a filter on the input data sets.

The rise of the available computational and graphics power over the past decades goes hand in hand with an increase of the sheer amount of available data. Computer algorithms

Figure 2.1.: Tor Bergeron and Jacob Bjerknes working on a hand-drawn meteorological visualization. Bergen, 1919. Image from Shapiro and Grønås (1999).

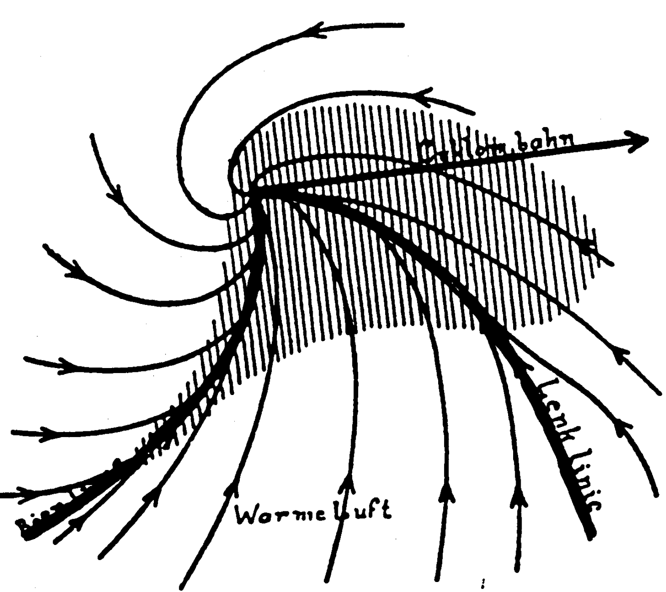


Figure 2.2.: Hand-drawn visualization of an idealized cyclone by Bjerknes and Solberg, 1919 (cf. Bjerknes, 1919).

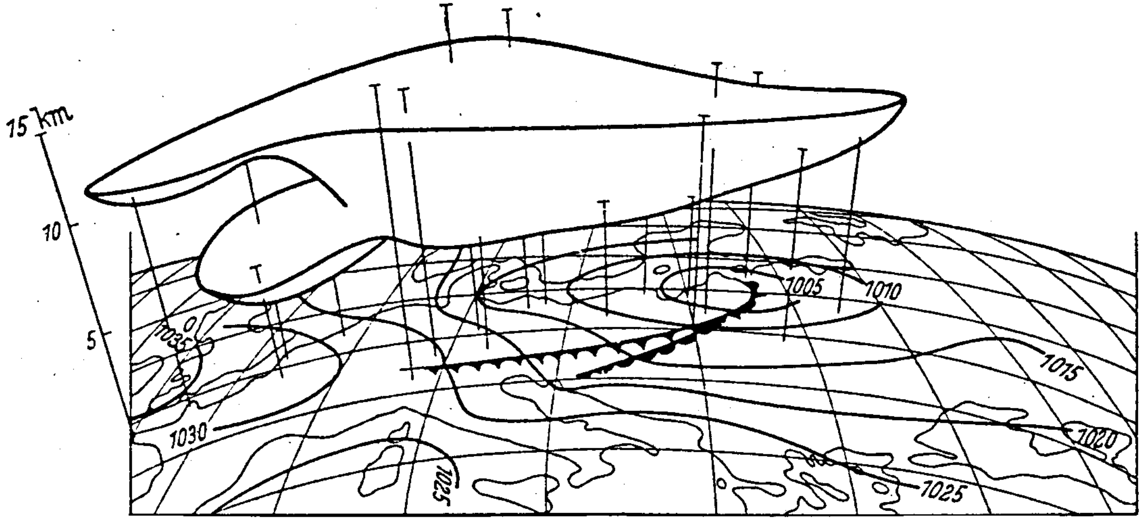


Figure 2.3.: Hand-drawn visualization of an upper-level PV anomaly by Kleinschmidt (1950).

are not only involved during the measurement of atmospheric variables, but also in the data preparation and assimilation processes, as well as in the process of creating new data sets derived from the measured data. Data *segmentation* techniques can be used for the identification of interesting features in large input data sets. Some segmentation techniques are an implicit part of a visualization technique (e.g. the visualization of isosurfaces requires a classification of the data set into areas above and below a given isovalue). Other segmentation techniques are realized as an explicit, stand-alone preprocessing step.

For the analysis, segmentation, and visualization of increasingly large sets of data, new methods, algorithms, and software tools had and have to be developed. An easy-to-use toolkit, which fits the needs and is easily adaptable to many different applications and that allows an interactive exploration of three-dimensional data visualizations helps developing an improved and deeper understanding of data sets. Feature extraction and feature tracking techniques help to derive new and/or reduced data sets which aid the user in the identification of the important regions and events in the original (larger) data sets.

The software presented in this chapter, Insight[1], was developed with exactly these goals in mind. The development of Insight started with the enhancement of an existing data visualization software by adding additional visualization methods and feature extraction and tracking capabilities. Over time, it became a flexible and individually adaptable toolkit for data segmentation and for a whole range of different visualization methods working on large amounts of atmospheric data.

2.1.1. Motivation

The main motivation behind the development of Insight was the need of a software foundation for the new segmentation algorithm (see Chapter 3). In the stage of development and testing, it was very helpful to visualize the input data and the results as three-dimensional

[1] see `http://insight.zdv.uni-mainz.de/trac`, last accessed: 18-June-2013

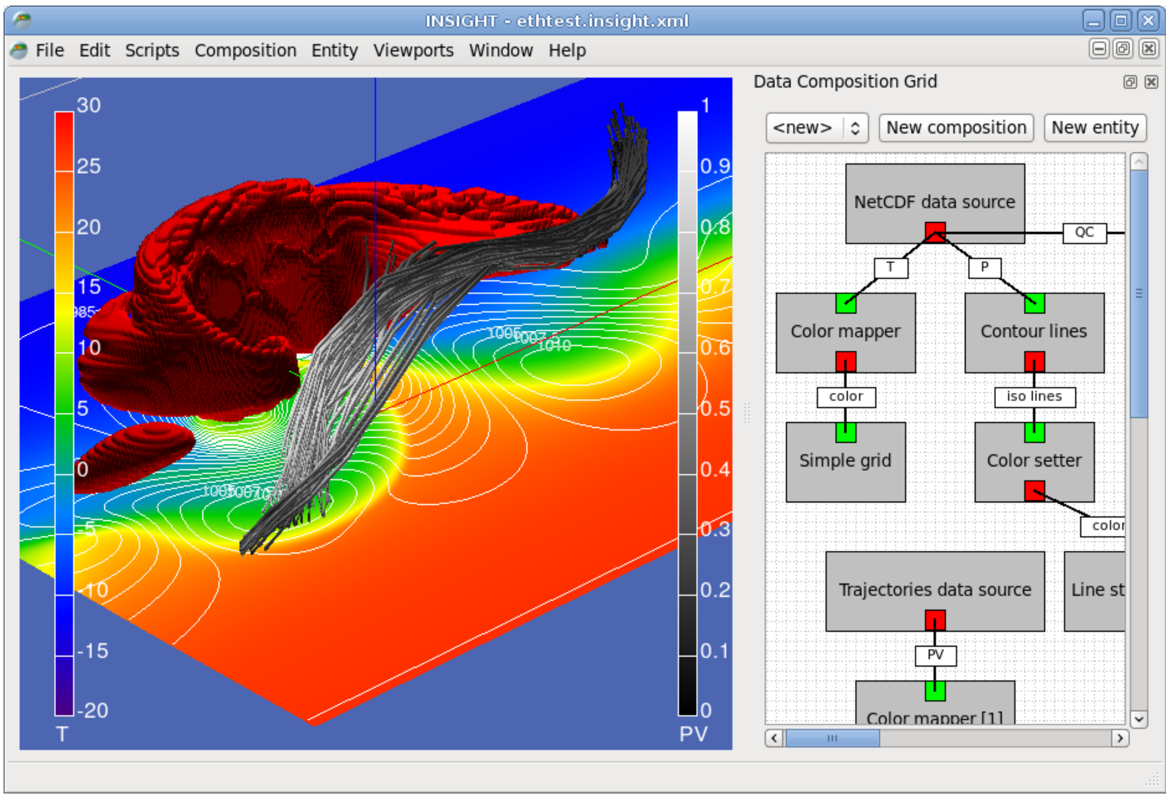


Figure 2.4.: Screenshot of a typical Insight session. The left part of the screen shows several types of 3D visualizations: an isosurface, cylinder curves representing trajectories, contour lines and an axis-oriented cutting plane. The right part shows a schematic visualization of the connected data entities.

objects, with different types of projection and the ability to manipulate the view on the data, and the current time step, at interactive frame rates. Along with the development of the segmentation algorithm, more and more features were added to Insight, such that today it can be applied to many other tasks besides data segmentation. Insight is now a versatile software tool providing an interactive platform for the processing and three-dimensional visualization of atmospheric data. Figure 2.4 shows a screenshot of a typical Insight session. The software evolved from Vis3d, a tool for the visualization of isosurfaces written by Marcus Marto (Marto, 2008), into a complete framework providing a wide range of components for the loading, processing, visualization, and output of large data sets.

Since many basic functionalities of Insight proved to be useful for other meteorological applications as well, it was obvious to extend the basic framework to cover use cases beyond data segmentation. The decision to extend Vis3d as a stand-alone software tool guaranteed a flexible, focused development of the segmentation algorithm. Nevertheless, we also looked into several other software tools for the three-dimensional visualization of scientific data, such as Vis5d[2] (Hibbard and Santek, 1990), ITK[3] (Ibáñez et al., 2005), Processing[4] (Reas and Fry,

[2] `http://www.ssec.wisc.edu/~billh/vis5d.html`, last accessed: 24-October-2012
[3] `http://www.itk.org/`, last accessed: 24-October-2012
[4] `http://processing.org/`, last accessed: 24-October-2012

2007), VisIt[5] (Childs et al., 2005), Vapor[6] (Clyne et al., 2007), OpenDX[7] (Thompson et al., 2001), Unidata's IDV[8] (Murray et al., 2003, 2009), and Kitware's ParaView[9] (Henderson, 2008).

2.1.2. Features

Although the development of Insight is still ongoing and the software could be further extended in various ways, it already offers a wide variety of features. Below, the main features of Insight are introduced.

Processing of atmospheric data

Insight offers a multitude of methods for the processing of data. The methods range from simple unary or binary operations, over more complex and completely customizable operations up to the computation of new data sets based upon given input data (an example are the results of the data segmentation methods provided by Insight). The program offers a schematic visual representation of the data processing pipeline in form of a two-dimensional graph. The program's modularity allows for the combination of the various available data processing algorithms in order to adapt to many individual data processing and visualization tasks. The general concept of data processing in Insight is described in Sect. 2.2. The technical realization of this concept is presented in Sect. 2.5.

3D visualization of atmospheric data

A major aspect of Insight are the different available types of 3D visualizations. For example, Insight provides several techniques for the visualization of sets of piecewise linear curves. These curves can be visualized as either simple colored line-strips of fixed line width, or as fully shaded, cylindrical curves. Possible application for these techniques are trajectories, contour lines, or coastline data. In addition, three-dimensional volume data sets can be visualized by means of isosurfaces, either fixed colored or shaded using another 3D data set as input. Another way of visualizing such data sets are axis-aligned cutting planes. Sets of single samples associated with a three-dimensional grid, for example resulting from the four-dimensional segmentation algorithm, can be visualized by shaded points or sets of small boxes. Of course, such sets of samples can as well be re-transformed into a 3D volume data set and be visualized accordingly. The event graph of a four-dimensional segmentation may also be plotted seamlessly as a three-dimensional construct consisting of spheres and connecting lines. In Insight, the data visualization is separated as much as possible from the loading and processing steps that are applied to the data in advance. The results of the three-dimensional visualization can be exported as an image file in various file formats and arbitrary image resolutions. More information on the different data visualization techniques can be found in Sect. 2.3.

[5] `https://wci.llnl.gov/codes/visit/`, last accessed: 24-October-2012
[6] `https://www.vapor.ucar.edu/`, last accessed: 24-October-2012
[7] `http://www.opendx.org/`, last accessed: 24-October-2012
[8] `http://www.unidata.ucar.edu/software/idv/`, last accessed: 24-October-2012
[9] `http://www.paraview.org/`, last accessed: 24-October-2012

Projections and data transformation

In Insight, the final position of all visualized data on the screen runs through two projection steps. The first step transforms the position from geographic coordinates (longitude, latitude and pressure) into points of $\mathbb{R}^3$. Insight offers different realizations of this transformation step, each of which corresponds to a certain map projection type. Examples are equidistant cylindrical or stereographic projections (centered at either the north or the south pole), as well as a projection from geographic coordinates to points on concentric spheres (where the radius depends on the height associated with the data). The second step transforms the point of $\mathbb{R}^3$ into screen coordinates, taking into account all required transformations of the current camera setup. This setup contains parameters such as camera position, view- and upward direction, and projection type (perspective or orthogonal). The details of the different data transformations of the first transformation step are discussed in Sect. 2.3 as well.

Data import and export

Insight provides functionality for the import of data in different file formats. The main file format supported by Insight for the import of atmospheric data is the netCDF file format (**Net**work **C**ommon **D**ata **F**orm, see Rew and Davis, 1990). Time series or other forms of clustered data which is split into several input files, are handled as well. Additional supported input file formats are the Lagranto (Lagrangian analysis tool, see Wernli and Davies, 1997[10]) file format for trajectory data, as well as an XML-based file format for sets of polygonal line chains. Insight offers mechanisms for the constraining of any input dimension, if the amount of input data or the available region should be reduced. Insight correctly handles input data on grids with hybrid pressure variable as well as data on rotated grids. For cases in which the visualization of the rotated data appears disturbed using the desired projection, Insight offers mechanisms to globally rotate all processed data back. For the general mapping of netCDF input data to geographic positions, a set of XML-based file format descriptions is parsed and matched to any given input file. Results from any computation or data processing in Insight can be exported as an output file in netCDF format, see Sect. 2.6 for details on the import and export of netCDF files. Data can also be exported in any custom binary or text format using the Python scripting interface and its capabilities of accessing any output variable (see Sect. 2.7.1).

Serialization

Insight supports the saving and loading of its own state, the whole data processing setup and the camera setup in form of XML-based save files and alternatively in form of Python script files. Parts of the data processing setup, so called *data compositions*, may be stored and restored separately[11]. This allows the user to individually add often required components, for example a visualization of coastlines, into existing program setups. In Sect. 2.8, the technical realization of both approaches for the serialization are described in detail.

[10] this paper focuses on the underlying method of trajectory calculations

[11] The separate export and import of data compositions is currently only implemented for the serialization through XML files.

Scripting support

All major aspects of Insight are remotely controllable through a set of Python bindings. Insight offers an interactive console for the direct input of single Python commands, but is also capable of executing script files directly from the command line. In addition, simple scripts combining several single commands are used to give users the option to easily execute commonly required control sequences. In Insight, these new commands are called *manipulators* and can be dynamically extended and added to the program. Additionally, Insight provides a data processor which lets the user define the computation of the output variable from multiple input variables by using an arbitrary Python script. The script is compiled at run-time into byte code and executed every time the output variable is requested. The scripting interface of Insight is discussed in Sect. 2.7.

2.1.3. Examples of use

Aside from our own application of Insight as a software tool for the segmentation and visualization of interesting features in atmospheric data, Insight was used from the beginning of its development by other meteorologists, mainly at the Johannes Gutenberg-University of Mainz and at the ETH Zurich, for several different visualization tasks. An example image from Philipp Reutter's PhD thesis (Reutter, 2009) is shown in Fig. 2.5. He used the three-dimensional visualization provided by Insight in order to analyze the results of his numerical simulations of pyro-convective clouds. Andreas Winschall examined the three-dimensional distribution of tagged water vapor in his diploma thesis about the origin of water during heavy precipitation events (Winschall, 2009). Julia Bachmann produced plots of PV-towers for her diploma thesis (Bachmann, 2010) on the forecast accuracy of the structure and tracks of extratropical cyclones (see Fig. 2.6). Wolfgang Langhans produced a series of plots and a video[12] for his studies on COSMO-simulations of heavy precipitation events. An article containing some of these visualizations was published in the ETH Globe magazine (Würsten, 2012). Sebastian Schemm is using Insight for visualizations of his simulations of idealized cyclone formation and of the related warm conveyor belts (Schemm et al., 2012). An example visualization by him is shown in Fig. 2.7. He also utilized Insight's scripting interface for automatically producing a series of plots and a video[13] showing the results of the simulations. The necessary setup for these visualizations are described in form of an example use case at the end of this chapter, in Sect. 2.9. Jana Campa visualized backward trajectories showing the pathway of moisture involved in the development of cyclones as a sequence of three-dimensional plots. One example plot is shown in Fig. 2.8. These visualizations were part of the presentation she gave for defending her PhD thesis (Campa, 2012).

Other examples for the use of Insight focus more on the segmentation algorithm that is described in more details in Chapter 3. Christine Aebi (University of Bern) and Erica Madonna (ETH Zurich) use the segmentation algorithm implemented in Insight for researching the connections of warm conveyor belts and streamers of high potential vorticity. First results can be found in Aebi's master's thesis (Aebi, 2012). Lukas Papritz (ETH Zurich) investigates

[12] `http://www.iac.ethz.ch/people/wolangha/cloud_animation_slow_mr.gif`, last accessed: 24-September-2012

[13] `http://insight.zdv.uni-mainz.de/download/video/Idealized_WCB_small.mov`, last accessed: 18-June-2013

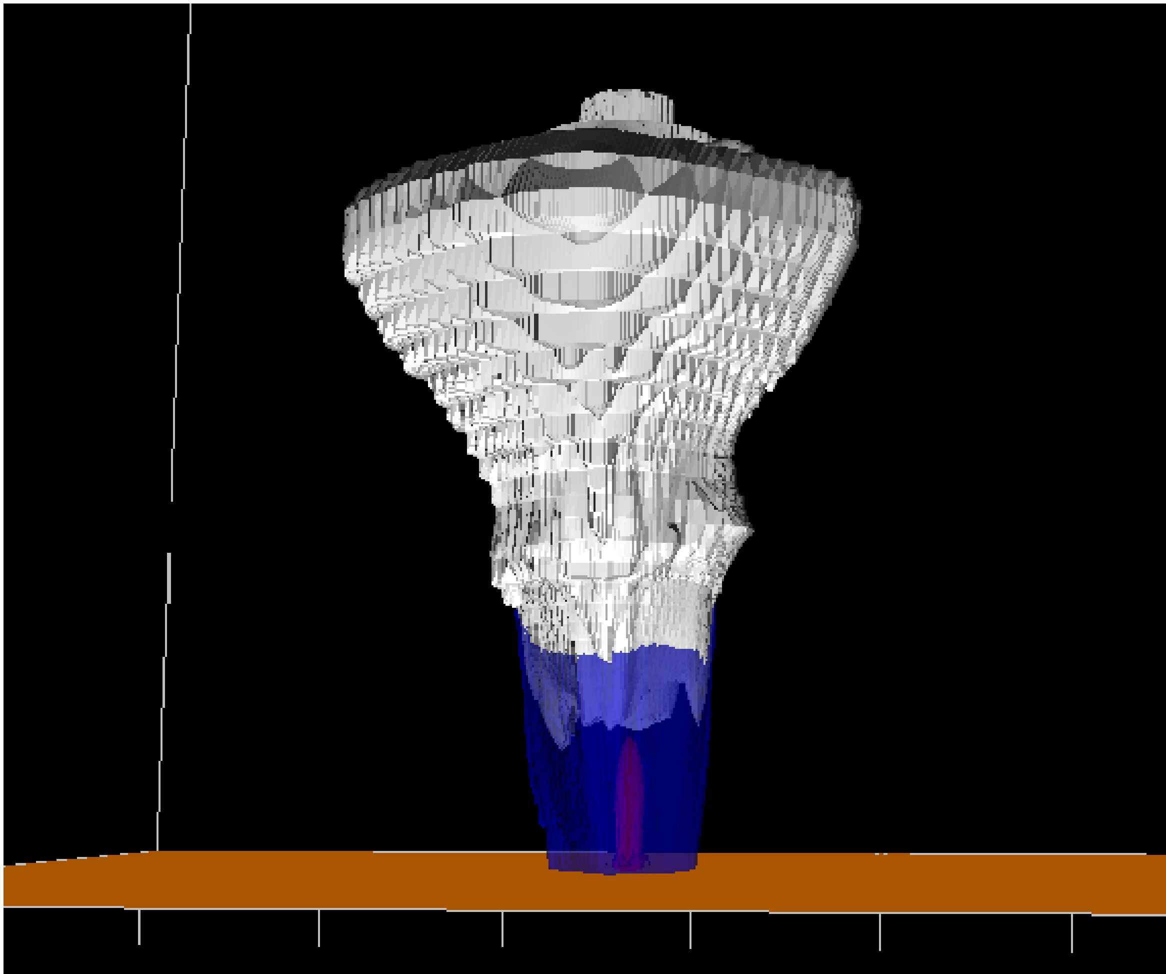


Figure 2.5.: Isosurfaces depicting different components of a pyro-convective cloud: The hydrometeor content without rain (white, 0.1 g/kg), the rain water content (blue, 0.1 g/kg), and the location of the fire (red, 25°C). Image taken from Reutter (2009) with permission from the author.

the three-dimensional structure of cyclones and their development climatologically. Insight is used for the cyclone detection and tracking, as well as for the visualization of some of the results. Details of the application of the segmentation algorithm, as well as some first results of his work are presented in Sect. 3.6.

2.1.4. Insight's predecessor

Insight evolved from Vis3d (Marto, 2008), a software tool for the visualization of three-dimensional isosurfaces developed by Marcus Marto (see Fig. 2.9 for a screenshot). The main features already present in Vis3d were netCDF import, visualization of isosurfaces (including shader-based transparency and cropping of the visible parts) and some basic UI elements (including support of multiple viewports). One basic concept of Vis3d, namely the separation of input data and different visual representations, was taken over in form of Insight's data processing concept using single data entities.

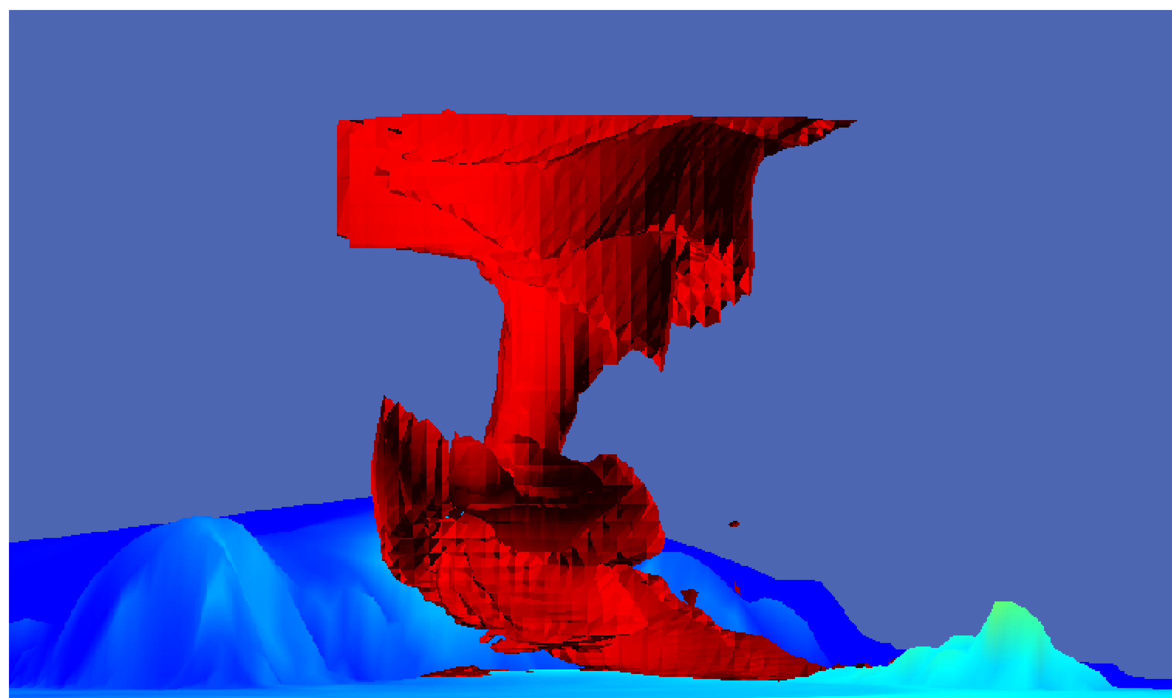


Figure 2.6.: Isosurface of potential vorticity (2 pvu) representing a “PV-tower”, i.e., a fully developed extratropical cyclone. The surface patch at the bottom of the plot indicates potential temperature on a color scale from 270 K (dark blue) up to 300 K (red, not reached in this plot). Image taken from Bachmann (2010) with permission from the author.

Several other software tools inspired the development of Insight indirectly, see the list at the end of Section 2.1.1. Although Insight was never meant to be a direct alternative for any of these existing programs, one goal during the development was to offer an easier and more intuitive user interface compared to those of the existing tools. This was already an important aspect during the development of Vis3d. The main motivation behind the decision to start the development of an own software was, however, the aim for maximum flexibility during the development and implementation of our own algorithms. By sticking to a modular software design, all parts of Insight benefited from improvements made primarily for the implementation of the segmentation algorithm, and vice versa, the segmentation algorithm profited indirectly from improvements made in context of other use cases.

2.1.5. Outline

The upcoming sections of this chapter can be grouped into two main parts and two closing sections. The first main part deals with the data processing and visualization concept of Insight. The second main part describes the integration of Python as a scripting language for Insight. The two closing sections contain the description of an actual application of Insight in form of a case study in which data from simulations of dry and moist idealized cyclones and warm conveyor belts are visualized.

The first two parts follow the general idea of describing the general concepts and ideas first,

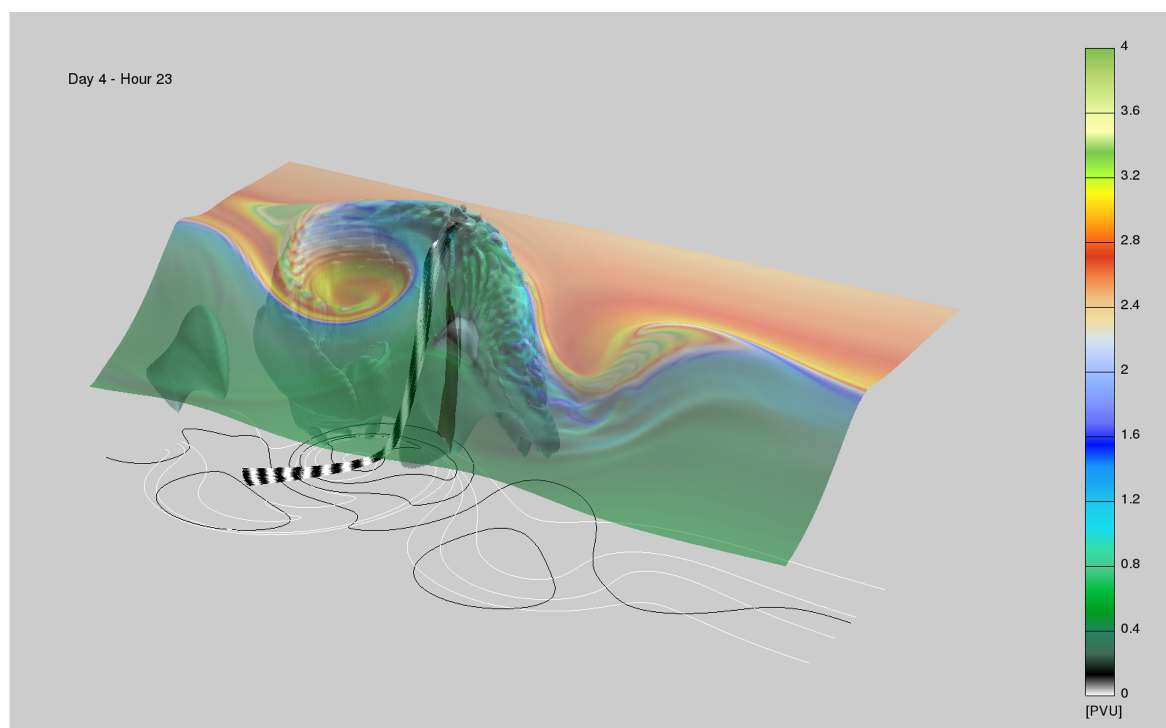


Figure 2.7.: Visualization of a cyclone simulation by Sebastian Schemm (image taken with permission from the author). The red isosurface shows the area of latent heating, the light blue isosurface depicts saturated air. The colored isosurface shows PV on the 320 K isentrope. The black and white trajectories run along the warm conveyor belt. The color switches between black and white for every simulated hour.

before all technical details and aspects of the implementation are discussed. Section 2.2 provides a general overview of the data processing concept of Insight. Basic terms such as "data variable" and "data entity" are introduced. The next section, Sect. 2.3, deals with the three-dimensional visualization techniques provided by Insight. Following these conceptual sections, the next sections cover the implementation and technical details of the core component of Insight, the data processing and visualization pipeline. Section 2.4 introduces the general outline of the software architecture, as well as all third-party libraries which are part of the implementation. In Section 2.5, the implementation of the models, controllers, and views of the data processing concept are discussed in detail. At the end of the first part, the technical details of the handling of netCDF files for data import and export are described in Section 2.6.

The second part consists of Sections 2.7 and 2.8. Section 2.7 deals with the applications of the Python language from a user's perspective, as well as from a more technical point of view. Section 2.8 provides information about the ways in which Insight realizes the loading and saving of the setup of the processing pipeline and the camera and viewport setup. Both the script-based and the XML-based serialization methods offered by Insight are described in this section.

In Sect. 2.9, the first of the two closing sections, an example case study is presented in which

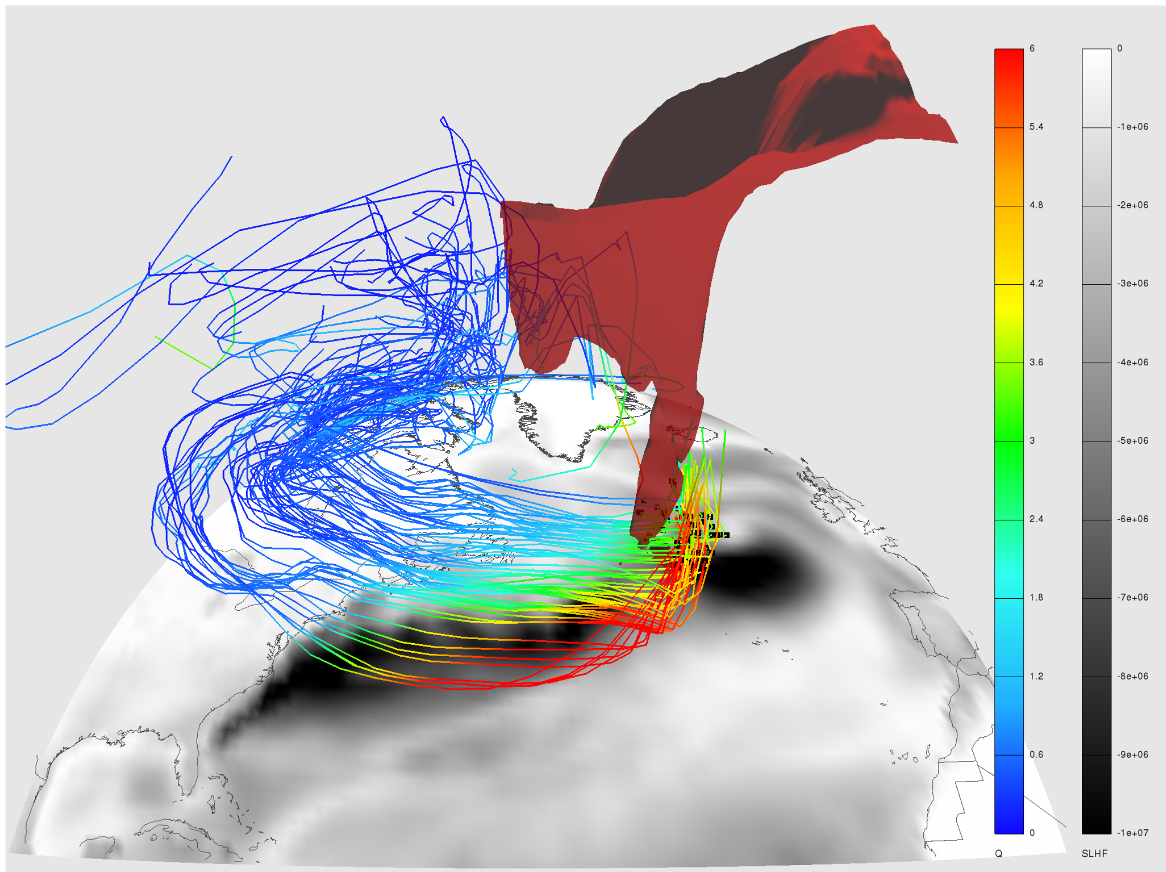


Figure 2.8.: The backward trajectories are started at the center of a storm and their color represents the specific humidity in g/kg. The red isosurface shows PV values of 1.5 pvu. The surface color shows the surface latent heat flux (SLHF) in W/m^2. Image taken from Jana Campas doctoral presentation with permission from the author.

different ways of utilizing Insight for the visualization of data from idealized cyclone simulations are shown. The section also contains the development of a script for the creation of a series of images which were later combined into a video sequence. At the end of this chapter, a summary and an outlook on possible future steps of Insight is given in Section 2.10.

2.2. Data processing concept

2.2.1. Introduction

The main task of Insight is the transformation of scientific data sets into three-dimensional, visual representations. Independent from the actual data and the concrete visualization technique, several elemental tasks have to be carried out in varying contexts over and over again.

As a motivation of the ideas behind Insight's data processing concept, we have a look at

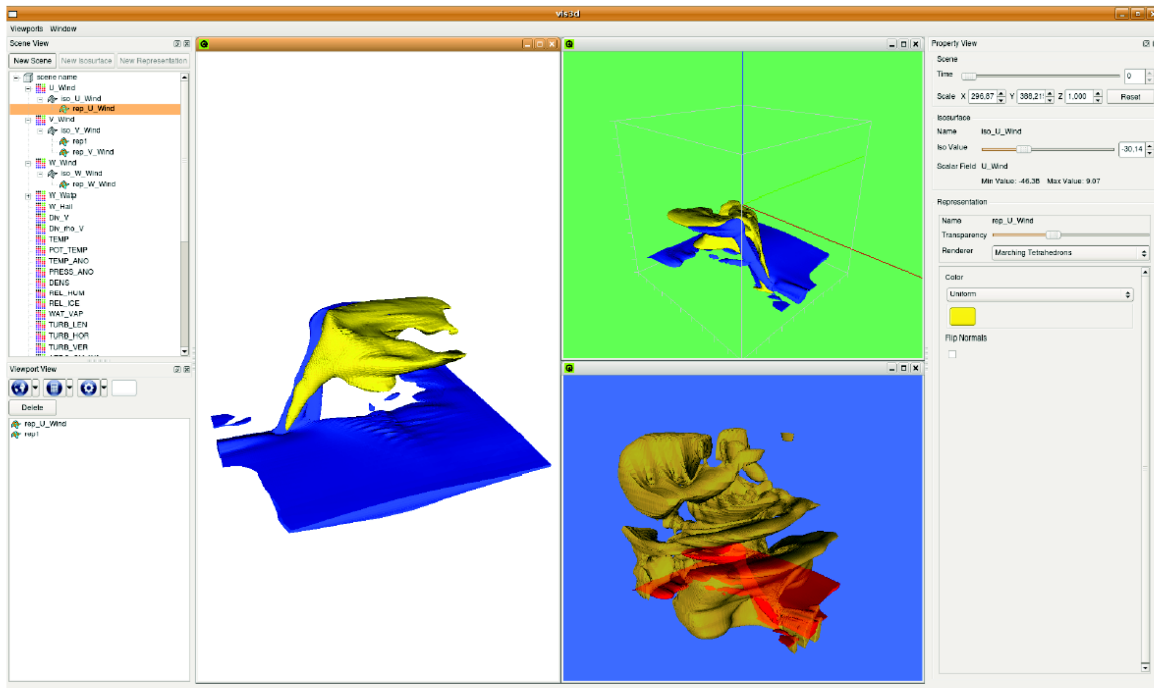


Figure 2.9.: Screenshot of Vis3d, Insight's predecessor. Image taken from Marto (2008).

three examples of typical basic visualization tasks:

1. A two-dimensional filled plot of a horizontal cross section of a temperature data set.
2. A two-dimensional contour plot of a horizontal cross section of the same three-dimensional temperature data set.
3. The visualization of a pencil of trajectories, colored using the potential vorticity along the trajectories.

In order to fulfill the first and the second task, the first step we have to do is to read in the temperature data set from the filesystem into the memory of the PC. We do not only need the basic data, but also meta information, such as the horizontal and vertical positions associated with the data, and the number of time steps available in the file. The next step of the first task is to define a mapping $M_0 : \mathbb{R} \rightarrow \mathbb{R}^3$ that associates each scalar temperature value with an RGB color value. Finally, we have to pick a vertical level we want to plot and draw a triangulated grid pattern using OpenGL primitives, whose vertices are at the positions associated with the data, and whose colors are determined through the mapping M_0.

For the contour lines, we have to pick the vertical level we want to visualize and compute a representation of the contour lines as sets of vertices delimiting piecewise linear curves. Next, we have to assign a fixed color value to the set of vertices and finally to visualize each contour line using an OpenGL line strip in the simplest case.

The third visualization task operates on a different set of data, namely on trajectories. These trajectories have to be read in from the filesystem as well. They are represented in memory as a set of vectors containing representations of all vertices each trajectory consists of. This

information is the position of the air parcel, as well as all scalar values traced along the trajectory at the respective points in time. For our example, we have to map the PV values of all vertices to RGB color values using another mapping $M_1 : \mathbb{R} \to \mathbb{R}^3$. Now we are able to visualize the trajectories, again using OpenGL's line strip primitives along with the positional information and the colors given by the mapping M_1.

These three simple examples already show some recurring tasks:

- reading in data sets from a netCDF file
- mapping scalar values to RGB color values
- visualizing piecewise linear curves using OpenGL line strip primitives

Other tasks, however, were required only once in our examples:

- reading in trajectory data from the filesystem
- computing contour lines out of a two-dimensional data set
- associating data with a fixed color
- visualizing two-dimensional data sets as a grid using OpenGL triangle primitives

Nevertheless, countless different situations exist, in which these tasks could be re-used as well. To allow the user a flexible combination of such simple tasks, and in order to avoid code repetition in our implementation, we decided to provide Insight's functionality in form of a flexible data processing pipeline. The core elements of this pipeline are combinable, atomic data processing objects called *data entities*. They are grouped into the categories *data source*, *data processor* and *visual representation*. The data exchange is based on entities providing or requesting *data variables*, each of which has its own possibly empty set of associated, hierarchically grouped *data dimensions*. Any variable may additionally be assigned with *positional information* and a *data grid*.

A possible setup for the three example tasks discussed above is shown in Fig. 2.10. The single elements of this concept are described in the following sections.

2.2.2. Data dimensions

Data dimensions are the objects of Insight that define the shapes of the data variables and control the access to particular data values. They can be part of the output of a data entity, which is called the dimension's *provider*. The most important properties of a dimension are its *size* and its *index* (or *current position*).

The index of a data dimension is an integer value which can vary over time. It represents a certain position in the interval defined by the dimension's size. The index value of a dimension is typically either controlled by a data entity (more precisely a data processor or visual representation), by another data dimension, or directly by the user of Insight. The current position of a data dimension is called *valid* if its value is greater than or equal to zero and smaller than the dimension's *size*. If this is not the case, the current position is called *invalid*.

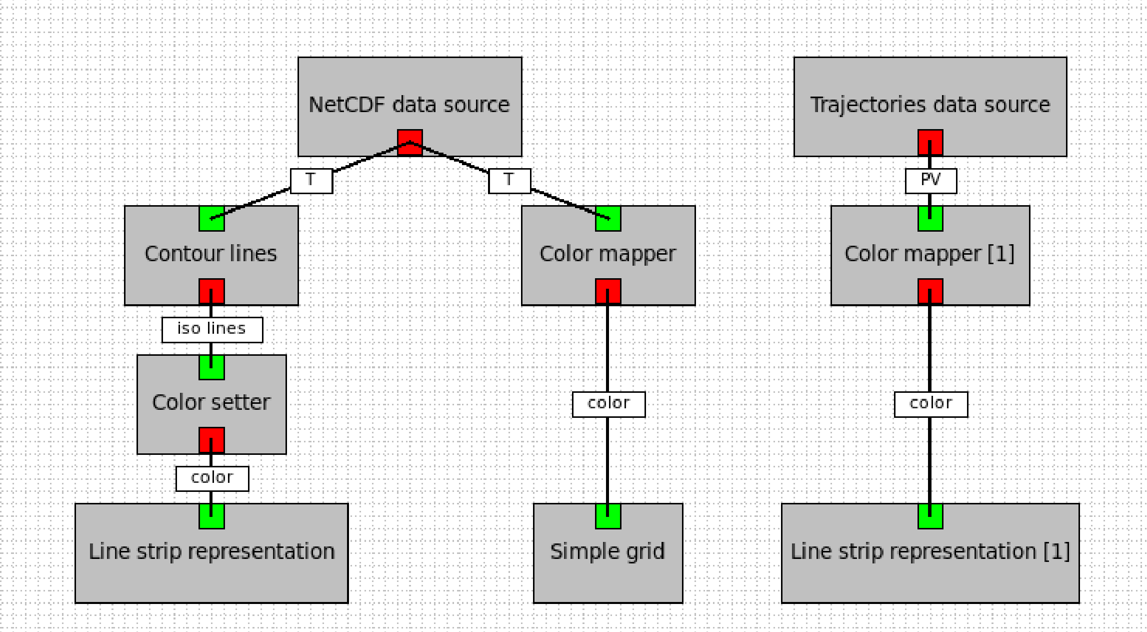


Figure 2.10.: Setup for three simple visualization tasks: A contour plot and a filled plot of a horizontal cross-section of temperature data, as well as trajectories colored with the PV values along each trajectory. Each box represents a data entity. The lines represent connections between data entities, for which the labels indicate the involved data variables.

The size of a dimension is either given as a fixed value, or as a function of the indices of a set of other dimensions. These other dimensions are called *parent dimensions*. An example of varying dimension sizes is the "vertex" dimension of the "2D-model data source" entity of Insight, which can be used, for example, to read in the geometry of the coastlines as a set of individual polygons. The size of the "vertex" dimension depends on the current position of its parent dimension, the "polygon" dimension, since each polygon consists of an individual number of vertices. The size of the "polygon" dimension is fixed, as it is the case for all data dimensions without any parent dimensions.

2.2.3. Data variables

Data variables represent a set of scalar data values associated with a finite, possibly empty set of data dimensions. The data variable is provided as part of the output of a data entity. It can be accessed by data processors and data visualization entities, or directly by the user through the scripting interface of Insight. Often, the data comes directly from sources such as netCDF files or trajectory files. In other cases, the data is processed or computed anew whenever it is requested.

Each data variable is associated with a finite set of data dimensions. The sizes of these dimensions define the shape of the represented data. Accessing different scalar values represented by a data variable requires a corresponding manipulation of the indices of the associated dimensions.

Data variables and data grids

In meteorological applications, many variables share common sets of dimensions which belong to the same underlying spatio-temporal *data grid*. There are different types of common grids which provide a mapping of tuples of dimension indices to geographic coordinates and certain points in time. Sometimes, for example in case of trajectory data, the variables are associated with a position and time, but not with an underlying grid.

2.2.4. Variable and dimension requests

The data processing and visualization algorithms operate on input variables and input dimensions. Each *variable request* is connected to a fixed number of *dimension requests*. If a data variable gets connected with a variable request, the dimensions associated with the data variable can be mapped to the dimension requests associated with the corresponding variable request. Insight tries to automatically find a good mapping of the dimensions to the requests. However, the user has full control over the mapping of dimensions. Multiple dimensions can be mapped to the same dimension request, and unmapped input dimensions can be set to fixed user-selected positions. For further information, refer to the implementational details discussed in Section 2.5.2. Some dimension and variable requests are optional and do not have to be fulfilled for the data processing or visualization algorithm to work.

2.2.5. Data entities

Data entities are the core building blocks of Insight's flexible data processing pipeline. The task of each entity is to perform one distinct, individual operation. The entire behaviour of any data entity is completely determined by up to four different settings:

1. The **input variables** and **dimensions** connected to the entity.
2. The **mapping** of the **input dimensions** to the respective requests.
3. The selected **positions** of all **unmapped dimensions**.
4. The state of all **properties** of the entity.

Examples of operations performed by data entities are the provision of data by reading in a netCDF file, the computation of a set of samples representing the result of a segmentation, or the visualization of input data in form of, for example, cutting planes or contour lines. Data entities can be grouped into the three categories *data source*, *data processor* and *visual representation*, depending on the requested and provided data, as well as on the operations performed. Some entities fit into more than one group, and despite of some functionalities of the visual representations being treated in a special way, this is a conceptual grouping which is not reflected by the actual implementation.

Data sources

All entities that provide data without requiring any input data are called *data sources*. The most common type of data source is the “NetCDF data source”, which is responsible

for reading in netCDF files and providing the corresponding data variables and dimensions given in the file as output. An important task of any data source entity is the creation and association of each output variable with a correct grid or position-provider object.

Data processors

Entities which require one or more input variables or input dimensions and provide at least one output variable or dimension are called *data processors* in Insight. Many data processors use the input data to compute a new data set which is then provided as output. An example for such a processor is the “Complete segmentation (4D)” entity as seen in Fig. 2.11. It takes input data on a grid, iterates over it and outputs (among others) a set of samples representing the segments found. This output variable has the dimensions *sample*, *feature*, *segment* and *time*[14]. Some other data processors, for example the “Color mapper” (also present in Fig. 2.11), transform the input data by direct application of a simple function into a different type of output (value to color, in this example). In this process, all dimensions and grid information of the input variable are passed on to the output variables. Internally, the values of the first type of data processors are often computed *en bloc* and the output data set is stored in memory or saved to disk, whereas the output of the second type is typically computed and returned *on-the-fly*, as soon as a single value is queried.

Visual representations

All entities with the ability to draw something on the computer screen are called *visual representations*. In most cases, a visual representation provides no output but requires some input data. An example of an important exception is the “Color mapper” already mentioned in the section before, which draws a color bar legend reflecting the applied color mapping. A more typical visual representation entity is the “Simple grid” entity for drawing axis-aligned cutting planes, or the “Iso surface representation” entity responsible for the visual representation of isosurfaces. Insight provides two types of data display. The three-dimensional, projected drawing area and one overlay view, which can be used for additional, two-dimensional information (for example plot captions or the just mentioned color bar legends). Insight supports multiple viewports (drawing windows), and there is a mechanism for setting the visibility of each visual representation separately for each viewport. In addition, most visual representations provide the two properties called “Is visible” and “Transparency”. The first property is a boolean switch for setting the visibility globally. A visual representation is drawn on a given viewport, if the global visibility is enabled *and* the visibility with respect to the distinct viewport is enabled. The “Transparency” property represents the transparency of a visual representation as a floating point value between zero (no transparency) and one (full transparency). The correct display of multiple transparent visual representations is not trivial. Insight adapted the per-pixel sorting of transparent representations using fragment shaders from the original Vis3d implementation, see Marto (2008) for details.

[14] The output variable lacks the time dimension if there is no valid mapping for the time input dimension request.

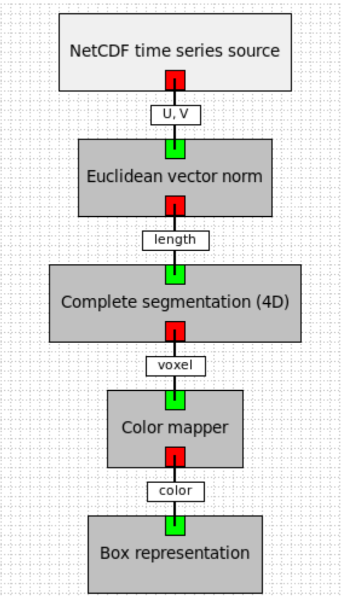


Figure 2.11.: Example of a data composition consisting of five data entities, visualized by Insight's data composition grid view.

2.2.6. Data compositions

A given set of data entities together with their data connections form a closed unit in Insight. This unit is called a *data composition*. Each existing data entity object belongs to exactly one composition. Entities from different compositions are not able to interact with each other directly. However, the visual representations of all data compositions share the same viewports. Insight represents data composition visually in form of a graph with the data entities as nodes and input/output connections as edges. An example for such a visualization of a data composition, consisting of five data entities for the segmentation and visualization of jet streams, is depicted in Fig. 2.11.

The reason for the introduction of data compositions is that some data processing setups can be useful in different settings. For example, a setup for reading in and visualizing the polygons forming a map of the coastlines and different countries of the world can be used every time a visualization requires the display of a world map.

2.3. Data visualization

In this section, we describe the main visualization techniques provided by Insight, as well as the different types of transformations and projections which are available. The visualization of the data sets is performed in two steps by Insight. Firstly, the abstract data is transformed into sequences of OpenGL primitives to be rendered. Examples of OpenGL primitives are points, lines, triangles and quads (see Woo et al., 1999, for an introduction). For this transformation, several different visualization techniques are implemented. Secondly, the vertex positions of the generated OpenGL primitives are transformed in order to map the positions associated with the data to their final coordinates in the OpenGL coordinate system.

Although this transformation is applied after the generation of the OpenGL primitives, it is described first in the upcoming sections, together with a short discussion of the ways in which Insight connects input data with positional data.

2.3.1. Transformation types

Insight tries to associate all data variables with geographic coordinates. This is not always possible, since some input data files do not provide positional meta information, and some data simply cannot be associated with one single geographic position (for example a globally averaged statistical value). Insight represents the geographic position by using a triple $\vec{p} := (\lambda, \varphi, p)^T$ of coordinates representing longitude, latitude, and the elevation (most commonly as a pressure value). Insight tries to convert positional data given in different coordinate systems into such a lon/lat/p-representation. Examples for coordinate systems that require a conversion are systems representing the elevation as geopotential height in meters above sea level or systems which represent (relative) horizontal and vertical locations in units of meters or kilometers. The height may be given in other units or measures by some data sources. In case of two-dimensional data, a fixed vertical position of 1013.25 hPa is associated with the data, in general. Some data sources offer additional settings for restricting the range of data or for setting a custom fixed vertical position.

If the input data comes from a netCDF file, the required information about the mapping of the data's grid indices to the (typically geographic) coordinates required by Insight is given in form of XML files. These files contain meta information about how the required information for the mapping can be obtained from variables and attributes of netCDF files following different conventions. See the detailed description in Sect. 2.6 for details on the implementation and the supported types of mappings.

Insight offers several different types for the final transformation of geographic coordinates $\vec{p}$ into positions of the OpenGL coordinate system $\vec{p'} := (x, y, z)^T \in \mathbb{R}^3$. The applied type can freely be chosen by the user.

Horizontal vs. vertical ratio

The vertical extent of the atmosphere is very small compared to its horizontal scales. Approximately 99 % of the mass of the atmosphere can be found in the lowest 30 km above sea level (cf. Wallace and Hobbs, 1977, page 13). It is therefore often useful for visualization purposes to exaggerate the data values vertically in order to get a better visual impression of the vertical distribution of the data. Insight maintains a global, user controlled factor for the scaling of the vertical positions. This factor is applied to the z-coordinate of $\vec{p'}$.

Direct transformation

For certain types of coordinates it is sufficient to directly take the coordinates of $\vec{p}$ as coordinates of the OpenGL world coordinate system.

Cylindrical transformation

The cylindrical transformation corresponds to a equidistant cylindrical map projection. The longitudinal and latitudinal coordinates are directly mapped to the x and y components of the OpenGL coordinate system. For this transformation, it is assumed that the third component of $\vec{p}$ contains the elevation as pressure with the unit hPa. The z component of $\vec{p'}$ is derived from this value using the barometric height-formula (cf. Roedel and Wagner, 2011, their Section 2.1) with a sea level pressure $p_0 = 1013.25$ hPa. Summarized, the transformation is given as:

$$\begin{pmatrix} \lambda \\ \varphi \\ p \end{pmatrix} \mapsto \begin{pmatrix} \lambda \\ \varphi \\ -7990 \cdot \log(\frac{p}{1013.25}) \end{pmatrix}$$

Spherical transformation

The spherical transformation maps the horizontal coordinates $(\lambda, \varphi)^T$ of $\vec{p}$ onto the surface of a sphere. The radius of the sphere depends on the pressure value p. The whole transformation reads as follows:

$$\begin{pmatrix} \lambda \\ \varphi \\ p \end{pmatrix} \mapsto r \begin{pmatrix} \sin\lambda\cos\varphi \\ -\cos\lambda\cos\varphi \\ -\sin\varphi \end{pmatrix}$$

In this transformation, the radius r is proportional to the logarithm of p, scaled by a constant factor $s \in \mathbb{R}$ and offset by a fixed value $t \in \mathbb{R}$, giving $r := t + s \log p$.

Stereographic transformation

Insight provides two types of stereographic transformations. Both transformations project the points of a unit sphere with spherical coordinates λ and φ onto a plane. The two types differ in the projection point, which is in one case the north pole and in the other case the south pole. The elevation of $\vec{p}$ is directly mapped to the z-coordinate, resulting in the following projection for the north-pole-case:

$$\begin{pmatrix} \lambda \\ \varphi \\ p \end{pmatrix} \mapsto \begin{pmatrix} \frac{\sin\lambda\cos\varphi}{(1-\sin\varphi)} \\ \\ \frac{-\cos\lambda\cos\varphi}{(1-\sin\varphi)} \\ \\ -7990 \cdot \log(\frac{p}{1013.25}) \end{pmatrix}$$

The point at $\varphi = \pi/2$ (the north pole) is the projection point and is excluded from the domain of this projection.

For the south-pole-case, the sign of $\sin\varphi$ changes, resulting in:

$$\begin{pmatrix}\lambda\\ \varphi\\ p\end{pmatrix} \mapsto \begin{pmatrix}\frac{\sin\lambda\cos\varphi}{(1+\sin\varphi)}\\ \\ \frac{-\cos\lambda\cos\varphi}{(1+\sin\varphi)}\\ \\ -7990\cdot\log(\frac{p}{1013.25})\end{pmatrix}$$

Here, the south pole ($\varphi = -\pi/2$) is excluded from the projection.

2.3.2. 3D data on a (deformed) regular grid

A wide variety of techniques exist for the transformation of three-dimensional data sets into geometric primitives and finally into two-dimensional images displayable on a computer screen. Among the most common techniques are the ones described in the next sections, all of which are implemented in Insight.

Cutting planes

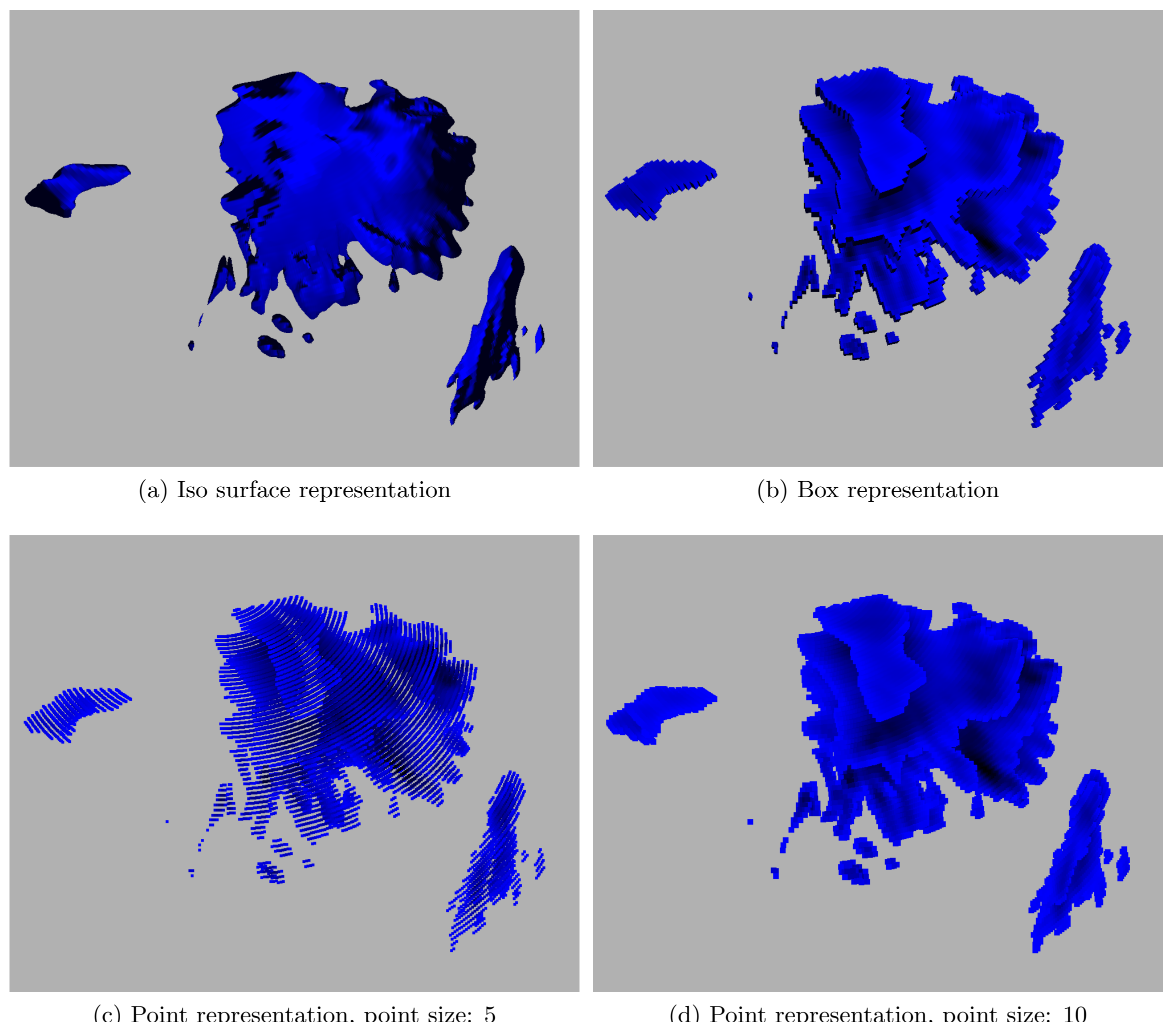
Insight offers a data entity for the visual representation of axis aligned two-dimensional cutting planes. The alignment is meant with respect to the underlying data grid's dimensions, where one spatial dimension is fixed while the implementation iterates over the other two dimensions and builds a triangle mesh connecting neighbouring grid points. The geographic coordinates of each individual grid point are used for the computation of the actual OpenGL coordinates, as described above. This could lead to a deformation of the surface, for example a horizontal cutting plane could have a varying height profile in accordance with the pressure values at each horizontal position.

Isosurfaces

Another common technique for the visualization of three-dimensional data is isosurfacing. Insight uses a marching tetrahedron algorithm for this task. Each vertex of the tetrahedra is colored either uniformly, or by interpolation values of a different input variable associated with the adjacent grid cells. These isosurfaces are closely related to the simple data segmentation method of thresholding. Details on the implementation of these isosurfaces are given in Marto (2008).

2.3.3. Subsets of 3D data on a (deformed) regular grid

Some data, for example the results of data segmentation, identify a subset of grid points of the (possibly deformed) regular source grid. Each such date is called a "voxel" (volumetric picture element) and is associated with the indices of a single grid point. In many cases,



(a) Iso surface representation

(b) Box representation

(c) Point representation, point size: 5

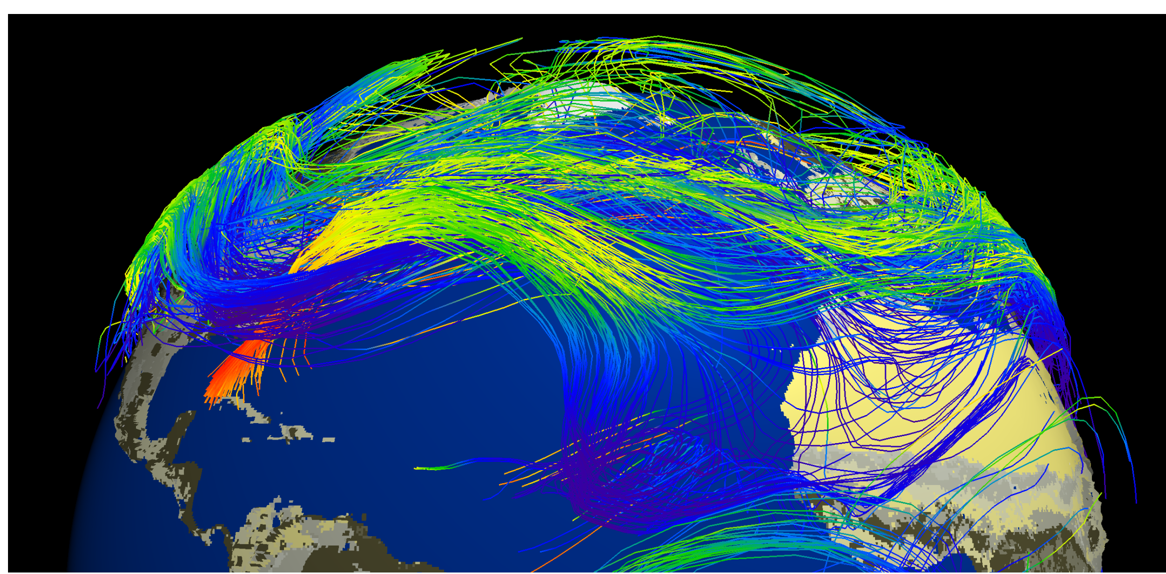
(d) Point representation, point size: 10

Figure 2.12.: Comparison of different types of visual representations. The isosurface representation works on data given on (possibly deformed) regular grids, whereas the point and box representations work on voxel data. These examples show a temperature isosurface at $-1°$C, whereas the voxel data is the result of data segmentation selecting all grid points with a temperature of more than $-1°$C.

the actual data value of the voxel is the value of the source data at the given point. A distinct set of visualization approaches is provided by Insight for the visualization of such voxel sets.

Point clouds

A very simple way for the visualization of subsets of a three-dimensional data set is using point clouds. For this, a simple `GL_POINT` primitive is drawn at the position of every grid point. The color of each point may be either fixed or varying with respect to the values of the input variable.



(a) OpenGL line strips

(b) Shaded cylinder curves

Figure 2.13.: Comparison of line strip visualizations with and without shading using OpenGL line strip visualizations and piecewise linear cylinder segments.

A variation of this method is to alter the size and the transparency of each individual point with respect to the value of the input variable. This technique is a simple example of a volume rendering technique. In contrast to the simple point cloud visualization, this method allows for a visible representation of structures inside solid 3D blocks of voxels.

Box representation

Sometimes, it is more convenient to visualize subsets of three-dimensional data as a connected object, and not as separate colored points. One way of realizing this is to visualize the corresponding grid cells in form of little boxes. The boxes fill the whole cell, which guarantees a seamless transition between neighbouring boxes, resulting in one large and solid geometry for connected voxel regions.

A comparison between the point cloud and the box representation of voxel sets, as well as a corresponding isosurface created from the base data set on the regular grid, not from the voxel set, can be found in Fig. 2.12.

2.3.4. Trajectories

Trajectory data requires a different visualization approach compared to the visualization of data given on a regular grid. Trajectories represent paths of single air parcels moving through the atmosphere, offering a Lagrangian perspective on atmospheric data. Data given on a regular grid, in contrast, correspond to the Eulerian way of looking at atmospheric variables.

Since pencils of trajectories describe the paths of a set of air parcels during a certain period, an obvious way of visualization is to use `GL_LINE_STRIP` primitives. Each vertex of each line strip can again be given a uniform color, or a varying color representing a variable which was traced along the trajectory.

The disadvantage of visualizing trajectories as line strips is that OpenGL lighting has no influence on their appearance. Lighting, however, can significantly increase the comprehension of the three-dimensional structure of a trajectory by the user. Therefore, Insight offers an additional way of visualizing trajectories as a curve composed of cylindrical segments. A direct comparison of both visualization techniques can be found in Fig. 2.13.

Currently, the only data format for trajectories recognized by Insight is the `.lsl` file format of Lagranto (Wernli and Davies, 1997).

2.4. Software architecture

In this section, an overview of the software architecture of Insight is given. At the beginning, the main components of Insight are introduced, followed by a description of the *Model View Controller* design pattern, which is an integral part of the design of many of Insight's software components.

The software architecture of the two most important parts of Insight, namely the data processing and visualization component, as well as the Python scripting component, are

described in detail in their own sections. The data processing and visualization component is subject of Sect. 2.5, with a separate discussion of the important topic of netCDF file handling in Sect. 2.6. Finally, the scripting component is covered in Sect. 2.7.

2.4.1. Overview

Insight's source code can be split conceptually into four main components.

- The **data background** is the most important component of Insight. It covers all classes for the representation and handling of all the data entities which are available in Insight. This includes all the code for reading the input data, the processing of the data in various ways, for example, the data segmentation algorithms, all the visualization code, as well as numerous auxiliary classes, for example for the caching of the data.
- The **scripting** component contains all classes related to the Python scripting interface offered by Insight. It is subdivided into sets of C++ and Python classes. On the C++ side, there are classes providing the core functionality together with the respective Python bindings, and classes for the GUI integration of the interactive scripting console. On the Python side, there are modules on top of the core functionality containing classes for the representation of the most important Insight objects and methods. These Python classes provide a more intuitive access to all object related functionalities (for example, for connecting data output and input).
- The **main window and viewports** component takes care of the setup of all other components, and their connection to the GUI. It is also responsible for the viewport handling, which includes, among others, the setup of the OpenGL environment, camera control and screenshot export.
- Finally, the **utilities and tools** component contains some basic implementations of fundamental data types (for example vectors and quaternions), as well as implementations of basic patterns, such as base class implementations for observer and observable classes.

An overview of the most important classes of the *data background* and the *scripting* components is given in Fig. 2.14. For the sake of simplicity, the utility and tool classes are omitted in this diagram.

The centerpiece of the *main window and viewports* component of Insight is an object of the `vis3d` class. Such an object represents the main window and is responsible for the management of all viewports. In addition, it contains the necessary code for the creation and setup of all objects related to the data background and scripting. Because of this, it is interacting with many classes throughout all of Insight's components. Figure 2.15 contains these connections to the *data background* and *scripting* classes, as well as the major classes that are responsible for representing and managing the viewports.

2.4.2. MVC concept

The general design of the data background and the scripting components of Insight follows the *Model View Controller* (MVC) design pattern, see Gamma et al. (1995) and Freeman

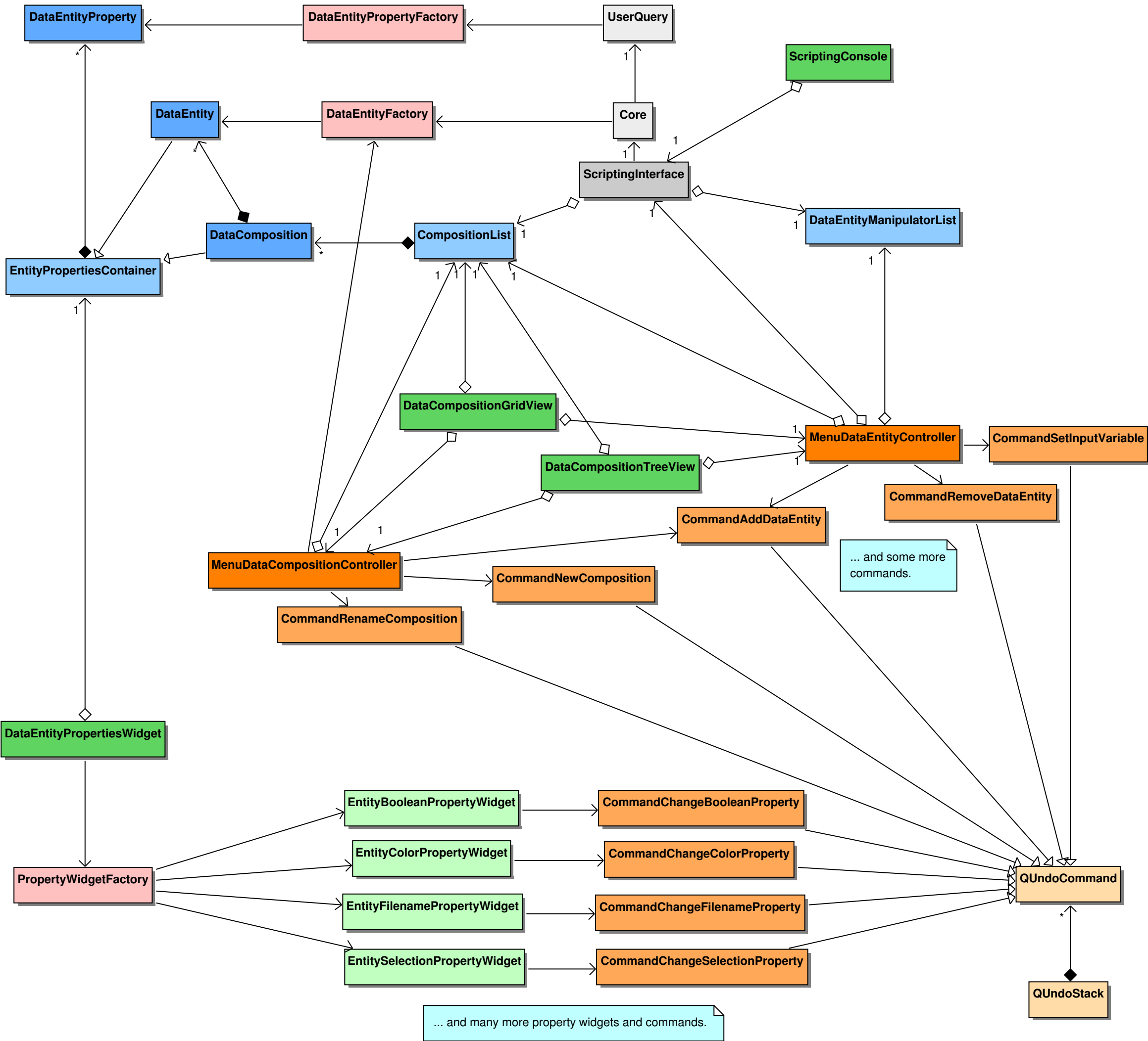


Figure 2.14.: UML class-diagram containing the most important classes of the data background and the scripting components of Insight.

et al. (2004). The basic idea is the separation of the state and core data of every object from the different representations of the object's data. Examples of these representations are the abstract representation in form of a GUI element, a 3D visualization of the model, or the representation of the data in form of a document to be printed out. The current state, the data and all related functionality of the objects are gathered in a *model* class.

Typically, one or more *view* classes are responsible for providing a visual representation of the state and data of a single model, as well as for providing means of user interactions. Sometimes, however, multiple models are represented through the same view. The model is responsible of informing all views about changes of its own state or data. The view classes react on these changes by updating themselves.

The *controller* classes are responsible for the communication from the views to the models, for example for the translation of GUI feedback events into single or multiple model object function calls. Due to the fact that most of the GUI drawing and handling code is available

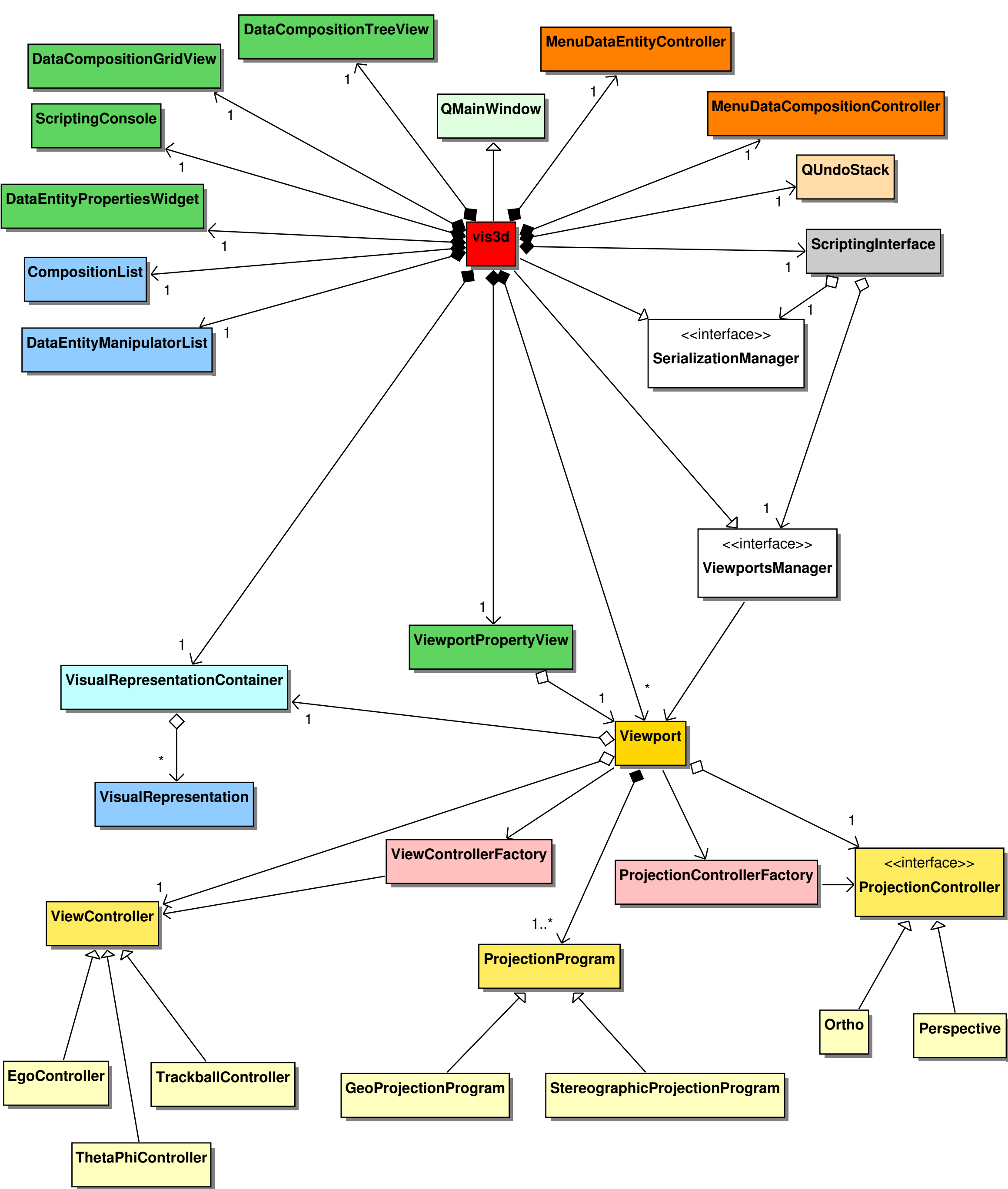


Figure 2.15.: This UML class-diagram shows the major classes of the *main window and viewports* component of Insight, together with the most important classes of other components the `vis3d` class is composed of. Further connections between these classes from other components, as well as all classes from the *utilities and tools* component are omitted.

in form of Qt classes, our view classes are very compact and usually handle only the setup and some basic interactions of the views. This allowed us to circumvent, for simple cases, the implementation of a separate controller class and to directly invoke the corresponding functions of the models. Complex commands and less complex commands that are invoked by multiple views are, however, implemented in the form of separate controller classes.

The communication between the models, views and controllers is realized using another design pattern, namely *Observer and Observable* (cf. Gamma et al., 1995).

2.4.3. Third-party components

A variety of external software libraries and frameworks are utilized in the implementation of Insight. In this section, a brief introduction to the most important third-party components used by Insight is given.

OpenGL and GLU

The foundation of all three-dimensional visualizations in Insight is the **Open G**raphics **L**ibrary, or *OpenGL*[15], for short. The first API specification for OpenGL was finished in 1992 by Silicon Graphics, Inc. (SGI)[16] and other companies with the goal of creating an open industry standard (cf. Rost, 2004). This first version of OpenGL was compatible with Iris GL, a proprietary API from SGI.

The OpenGL specifications undergo a continuous development process which is currently managed by the non-profit consortium Khronos Group[17]. Since OpenGL 1.0 was finalized in 1992 (see Segal and Akeley, 1994, for full specifications), three successive major versions of the API were released. OpenGL 2.0 (Segal and Akeley, 2004) was released in 2004 and introduced programmable vertex and fragment shader using *GLSL*, the OpenGL shading language. In 2008, OpenGL 3.0 (Segal and Akeley, 2008) was released. One goal of this new version was the transition away from the state-based system according to which OpenGL was designed up to this point. This resulted in many fundamentally changed functions for the creation and manipulation of objects, for example the complete removal of the matrix stack and all matrix-related operations. These old functions, however, were not removed directly in version 3.0 but marked as deprecated in order to sustain backward compatibility. The last major version released was OpenGL 4.0 (Segal and Akeley, 2010) in 2010. Insight was developed for OpenGL 2.0.

Another library used by Insight that is closely related to OpenGL is *GLU*, the GL Utilities library. The library is built on top of OpenGL and uses OpenGL commands to provide easier access to some features, as well as some completely new features. GLU contains support for mipmapping, matrix manipulation, polygon tessellation, quadrics, NURBS, and error handling, cf. Chin et al. (1998).

[15] `http://www.opengl.org`, last accessed: 18-June-2013

[16] Silicon Graphics, Inc. was bought by Rackable Systems, Inc. in 2009, which was later renamed back into Silicon Graphics International Corp.

[17] `http://www.khronos.org`, last accessed: 18-June-2013

Qt

Qt[18] is a cross-platform C++ class-framework for the creation of applications with a graphical user interface (GUI). The development of Qt was started in 1991 by Haavard Nord and Eirik Chambe-Eng. They both founded the software company Trolltech in 1994. Qt became available to the public for the first time in 1995. Trolltech was acquired by Nokia in 2008 and was renamed later as "Qt Development Frameworks"[19].

Qt provides a variety of different GUI-elements, so called *widgets* and a system for the invocation and receiving of notifications using *signals* and *slots* (cf. Blanchette and Summerfield, 2008). In addition, Insight also uses further components of Qt, for example for reading and writing XML files through Qt's `QtXml` module and for the creation of windows with an OpenGL rendering context through the `QtOpenGL` module.

One guideline throughout the development of Insight was to avoid the usage of Qt as much as possible in all classes not directly related to the GUI. The idea behind this was that if we ever want to create an Insight version without a GUI or with a completely different type of GUI that uses another GUI-framework, the code for the data processing background should be reusable without any major adaption. See Section 2.10 for some further discussion of some of these ideas.

Python

Python[20,21] is a programming language released to the public in 1991 by Guido van Rossum. Some of the language's major features are full support of object-oriented programming, a dynamic type system, garbage-collection, late binding ("duck typing"), and a comprehensive standard library. Python programs are usually not compiled and linked into native machine code, but executed by a Python bytecode interpreter. Own Python modules can be written using C or C++ code. For the inverse way, C libraries exist to include a Python interpreter in own C or C++ programs.

Insight uses Python as its scripting language for several different applications. Scripts for the creation and manipulation of a data visualization pipeline can be executed from within the running program, or they can be passed to Insight at startup in form of a command line parameter. An interactive scripting console is offered by Insight's GUI for the direct execution of single Python commands. Further, Python is used to implement macros (sequences of commands) which can be executed for specific types of data entities from their respective context menus. This way, the functionality of Insight can be extended without the need of a recompilation. Python is also used by one data entity directly, giving the user the option to use individual Python scripts for the processing of data. Finally, Python scripts can be automatically generated to save the current state of the program, as an alternative to the program state serialization via XML files.

[18] `http://qt.nokia.com`, last accessed: 30-August-2012
[19] cf. `http://qt.nokia.com/about/who-we-are`, last accessed: 29-August-2012
[20] "Named after the funny TV show, not the nasty reptile." (van Rossum and de Boer, 1991)
[21] cf. `http://www.python.org`, last accessed: 30-August-2012

Boost

Boost[22] is a collection of C++ libraries for a wide range of different tasks. Boost libraries are portable and peer-reviewed and are designed for a seamless collaboration with the C++ standard library. Several boost libraries are included as standard library components of the new C++11 standard.

Insight's main application of Boost is the usage of the Boost.Python library for easier integration of and export to the Python scripting language (Abrahams and Grosse-Kunstleve, 2003).

NetCDF C++ library

NetCDF (short for **net**work **C**ommon **D**ata **F**orm) is both the name of a file format and of a set of libraries for accessing and creating netCDF files[23]. NetCDF was developed by Unidata, which is a program of the *UCAR Community Programs* (UCP), formerly known as *UCAR Office of Programs* (UOP). UCAR stands for **U**niversity **C**ooperation for **A**tmospheric **R**esearch[24]. Over 70 North American universities are members of UCAR[25].

Insight uses the *NetCDF C++ library*[26] for reading and writing netCDF files.

XML

XML[27] stands for **E**x**t**ensible **M**arkup **L**anguage, and is the name of a markup language developed in 1996 by the World Wide Web Consortium (W3C) (Bray et al., 2008). XML is based on SGML, the **S**tandard **G**eneralized **M**arkup **L**anguage. Both XML and SGML are meta-languages as they allow the definition of own customized markup languages[28]. XML is used by many different applications and part of several protocols and web services[29].

In Insight, meta-information about netCDF file formats and the serialized program state are represented using XML files.

2.5. Realization of the data processing concept

In Sect. 2.2, the fundamental design of Insight's data processing concept was already described. In this section, we focus on the actual implementation. The *model* classes implementing the data processing concept contain the main functionality of Insight. However, since Insight is an interactive tool providing a GUI, there are not only classes for the data

[22] cf. `http://www.boost.org`, last accessed: 30-August-2012
[23] cf. `http://www.unidata.ucar.edu/software/netcdf/`, last accessed: 30-August-2012
[24] cf. `http://www.unidata.ucar.edu/software/netcdf/docs/`, last accessed: 30-August-2012
[25] cf. `http://www.ucar.edu/governance/members/institutions.shtml`, last accessed: 30-August-2012
[26] `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-cxx/`, last accessed: 30-August-2012
[27] `http://www.w3.org/XML/`, last accessed: 18-June-2013
[28] cf. `http://www.isgmlug.org/`, last accessed: 30-August-2012
[29] see `http://xml.coverpages.org/xmlApplications.html`, last accessed: 18-June-2013, for examples

processing itself, but also for realizing the different views and user interactions of the processing pipeline. These latter classes (the *controller* and *view* classes) allow for the representation and the interactive modification of the data pipeline and are described in a follow-up section.

2.5.1. Data processing models overview

As it is good practice in software engineering, all basic functionalities of the data processing pipeline are implemented in a flexible and modular way. This allows the software to fulfill a variety of complex visualization and data processing tasks by combining basic building blocks which share common interfaces for data input and output. These basic blocks are called *data entities* and are realized in form of `DataEntity` classes which implement interfaces for the provision (`DataProvider`), the request (`DataReceiver`) and for the visualization (`VisualRepresentation`) of meteorological data. Figure 2.16 shows an UML class diagram containing the main classes of Insight's data processing background and their relations.

The data which is interchanged between data entities is realized by means of two main classes: `DataVariable` and `DataDimension`. These two classes are closely related and interact in numerous ways. Each data variable has a fixed number of dimensions. The sizes of these dimensions define the shape of the variable's underlying data set, whereas the positions of these dimension define the current value of the data that can be accessed through the variable.

2.5.2. The DataDimension class

In order to enable a consistent and detailed understanding of the functionalities and the behaviour of the `DataDimension` class, we formally define the concept of Insight's data dimensions first. A data dimension d has both a size and an index (or current position). The size of d is either constant or determined by an integer-valued function $s_d : \mathbb{N}_0^n \mapsto \mathbb{N}_0$. The domain of this function itself depends on the sizes $s_{d_0}, \ldots, s_{d_{n-1}}$ of the n *parent dimensions* of d. A dimension d has a possibly empty, finite set P_d of parent dimensions. The conditions $d \notin P_d$ and $\forall d' \in P_d : \forall d'' \in P_{d'} : d'' \in P_d$ must hold. In other words, the set of parent dimensions of a dimension must contain all parent dimensions of any of its parent dimensions. We can define a hierarchy of parent dimensions of P_d by grouping the dimensions based on the cardinality $|P_{d'}|$ of any dimension $d' \in P_d$. A consequence that follows from the fact that P_d is finite is that there must be at least one dimension without any parent dimensions in P_d, if P_d is not empty. When we talk about the *index* or *position* of a dimension, we refer to an integer value i from the interval $[0, s_d[$. If this interval is empty, we call the dimension's position *invalid*.

This core information of a dimension (its current size, its parent dimensions, and its currently selected index) is stored in each `DataDimension` object, together with a name and the dimension's provider. In practice, there are cases in which the maximum size of a dimension may be unknown or where the computation of the size or a random setting of the position is too expensive. Because of this, iterations over dimensions are usually realized by resetting a dimension, and then by increasing the dimension's position as long as the dimension's position is still valid. The validity of a dimension's current position can be queried through

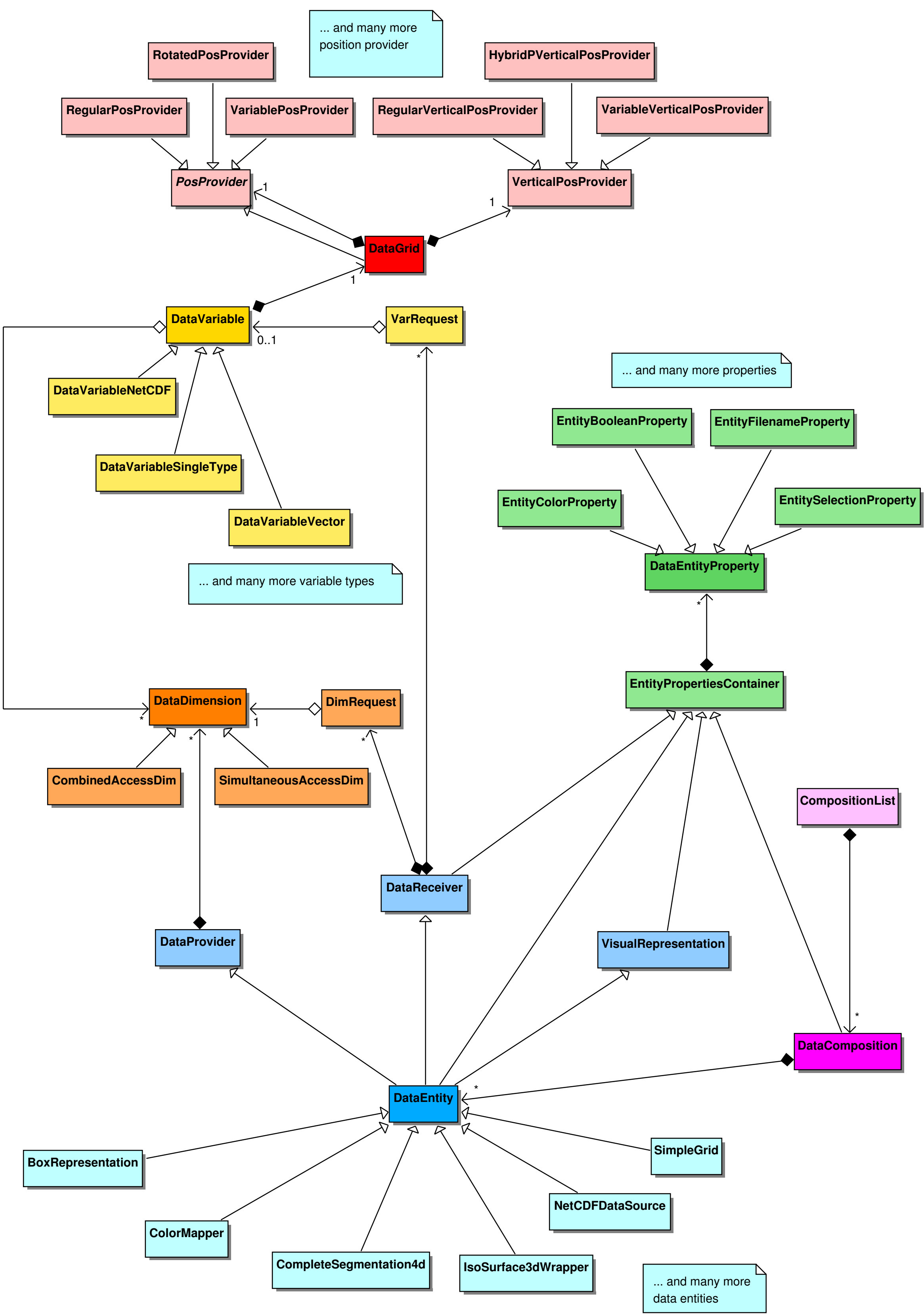


Figure 2.16.: UML class diagram containing the most important classes of the data processing pipeline's *model*. The controller, view, and factory classes are not shown in this diagram.

a corresponding member function. A dimension's position may be set to an arbitrary integer value. This operation is, in the worst case, realized by a corresponding number of single increase steps, possibly preceded by a reset if the new position is smaller than the current one. In some cases, the maximum or current size of a dimension needs to be known, for example for the selection of an unmapped dimension or for the netCDF export of a variable. The required member functions exist as part of the `DataDimension` class, but not all actual dimensions provide this information. Note that in contrast to data dimensions, an associated data grid of a variable always provides static size information.

Insight allows to delimit the valid positions of a dimension to a fixed, user-defined *restricted range*. This restriction is realized as a feature of the `DataDimension` class and can be applied both to dimensions with fixed and with varying sizes. The attributes representing the current size and the current position of the dimension take into account the restricted range. A set of additional member functions is provided for accessing the *real* position and the *real* size of the original, unrestricted dimension. As long as the upper bound of the restricted range corresponds to a valid index, the current size of the dimension equals the size of the specified range. The indices of the restricted dimension still start at zero, but they are internally shifted about the value of the lower bound of the range, compared to the original indices. The range of an unrestricted dimension is not fixed, but varies with the dimension's real current size.

Insight has the capability to access multiple dimensions through the same interface as a single dimension. This feature is realized by means of the `CombinedAccessDim` and the `SimultaneousAccessDim` classes. Both classes are implemented as *decorators* of the interface of the `DataDimension` class. *Decorator* is yet another design pattern described in Gamma et al. (1995). A decorator provides extended functionality through the interface of the base class (in our case, the `DataDimension` class). But in contrast to simple derivation, instances of the base class are used to store the internal data, and calls to functions with unaltered functionality are directly handed over to these objects of the base class.

SimulataneousAccessDim

The `SimulatenousAccessDim` class manages the simultaneous access to a set of dimensions. A typical case in which such access is required is the combination of data variables from different data sources (for example from two different netCDF files) as input for certain data entities. Even if these variables were associated with grids of equal sizes, the actual dimensions associated with the variables would be distinct. This may lead to erroneous results during the computation of a data entity if both variables are part of the input but only the dimensions of one variable are mapped to the entity's dimension requests. In such cases, the `SimultaneousAccessDim` class allows the entity to access the grid dimensions from multiple sources simultaneously. The size of the `SimultaneousAccessDim` is the minimum size of all of its mapped dimensions, and setting the position of the `SimultaneousAccessDim` is passed on to all corresponding setters of the mapped dimensions. Note, that in Insight's current implementation only dimensions with fixed sizes can be accessed simultaneously. The use of dimensions with varying sizes is prohibited, since dimension sizes depending on other mapped dimensions may lead to non-trivial sequences of valid dimension positions.

CombinedAccessDim

The `CombinedAccessDim` realizes the sequential access of all valid combinations of positions of a set of dimensions. This corresponds to the projection of multiple dimensions into a single one-dimensional dimension. A typical application is the visualization of the result of a segmentation. The resulting grid points (voxels) associated with all segments are grouped by the dimensions *voxel*, *feature*, and *segment*. Each four-dimensional segment has its own number of three-dimensional features, and each feature consists of a individual number of voxels. If we want now, for example, to draw each voxel as a single point using the “Point representation” data entity, we need to map all these dimension to the single *point dimension* which is requested by the “Point representation” entity. Internally, such a mapping is realized by utilizing `CombinedAccessDim` objects.

The previous example clearly shows that, in contrast to the `SimultaneousAccessDim`, it is essential for the `CombinedAccessDim` to be capable of handling dimensions with varying sizes. In such cases, the required mapping of an index position p_0 of the combined dimension to the vector $(p_1, \ldots, p_n)$ of positions of the n original dimensions is, however, not always trivial. The problem is that the sizes of each dimension with respect to the current position of all parent dimensions could be completely arbitrary. Exploring and storing of these size-structures require non-negligible amounts of memory and computing time. In contrast to this, advancing from a current position p_0 to the next or previous position $p_0 \pm 1$ is quite simple and much more common during the computations of Insight’s data entities.

Therefore, the implementation of the `CombinedAccessDim` handles sequential access very efficiently, whereas the less often required random access is broken down into a corresponding number of single steps from the current to the new position. The most expensive operation is querying the size of the combined dimension, which would in general require a sequential iteration over all combinations of valid positions of the first $n-1$ mapped dimensions. For this reason, it is not part of the current implementation, which is no problem in practice, since a combined dimension is never part of an output and therefore never subject of a dim-selection or a netCDF export.

The iteration itself is performed by the following strategy: At construction, the ordered list of all mapped dimensions is sorted such that all parent dimensions appear in front of their child dimensions. When the combined dimension is reset, all positions of the mapped dimensions are set to zero. For advancing to the next valid position, the mapped dimensions are increased in lexicographical order. The procedure starts by increasing the last dimension in the list of mapped dimensions by one. If this leads to an invalid position, the dimension is reset and the previous dimension in the list is increased. The validity of this dimension is then checked and the method is iterated until either the last increase led to a valid position, or the first dimension in the list was increased to an invalid position. Since we want to skip any invalid combination of dimension’s positions, after a dimension was increased to a valid position it is still necessary to check the validity of all subsequent dimensions in the list (which have just been reset). If any dimension’s size is now zero (and therefore, its position is invalid), the increase is continued at the dimension which was increased last.

2.5.3. The DataVariable class

In Insight, each data variable corresponds conceptual to an n-ary real-valued function $f : \mathbb{N}_0^n \to \mathbb{R} \cup m$, where m denotes a *missing data value* or *fill value*. The domain of the function parameters $(p_0, \ldots, p_{n-1}) \in \mathbb{N}_0^n$ of f is determined by the sizes of the n associated dimensions $D := \{d_0, \ldots, d_{n-1}\}$ of the variable. As discussed above, these sizes of the dimensions do not have to be fixed. However, the condition $\forall d \in D : \forall d' \in P_d : d' \in D$ must hold (the set D of all dimensions of a variable must contain all parent dimensions of all dimensions of D). If this was not the case, the domain of the function f would not be well-defined.

The interface of the `DataVariable` class offers methods for querying basic information about the variable (its name, data type[30], dimensionality, associated grid and more). The data access itself is provided by the two basic `toInteger` and `toDouble` member functions. The functions return the single scalar value with respect to the current positions of all associated data dimensions. A data variable with one or more dimensions with invalid positions has an undefined value.

The association of a data variable with an underlying data grid is represented by managing a pointer to a `DataGrid` class. Sometimes, for example in case of trajectory data, the variables are associated with a position and time, but not with an underlying grid. In these cases, only a `PositionProvider` class is connected with those variables.

Data representation and management

Many different types of data from different sources and with varying internal representations, either in memory or on a storage device, can be accessed through the interface of `DataVariable`. Examples are values from a fixed function without the need of an explicit representation of all values in memory, results from the processing of input data which are stored in memory in form of a list or a multidimensional array, or data from a netCDF file stored on a hard disk drive which is read into memory in chunks on demand. For the different types of data, several sub-classes of `DataVariable` exist. These sub-classes range from variables representing a single scalar value which never changes (represented by the generic `DataVariableSingleType` class), over variables for storing multidimensional data with fixed dimension sizes in a one-dimensional `std::vector` (`DataVariableVector`). NetCDF file access, either of a single files or of time series or clusters of netCDF files with equal structure, are also handled by corresponding sub-classes, namely by `DataVariableNetCDF`, `DataVariableNetCDFSeries`, and `DataVariableNetCDFCluster`.

Data access

The preparation of the data behind each `DataVariable` object follows the *lazy loading* strategy (Fowler, 2002, pages 200ff.). That means that the data is neither computed nor loaded until the first access. At the end of an access, the data may be dropped in order to make the memory available again for other data entities and variables. However, the amount of data available through a single `DataVariable` can be greater than the total available memory. As a consequence, the data variable classes need some meta-information about

[30] currently, the two C++ types `double` and `int` are supported

the planned access in order to provide efficient access to the requested data. For this, all `DataVariable` classes provide the `setAccessDims` function, for specification of the dimensions that are likely to change rapidly during the upcoming data access. This allows for a specific prefetching and caching of the correct data, for example by loading the respective parts from a netCDF file into memory. For the deletion of the data from memory after the access, the `freeMemoryHint` function can be used. As the function name suggests, a call does not guarantee a deletion of the data.

In order to further speed up every single variable access, no test is performed on whether or not any dimension of a data variable has changed which is not part of the access dimensions. Because of this, it is important that any variable access is preceded by a call to `setAccessDims` with a full list of all changing dimensions. The association of access dimensions with a variable request of a data entity is represented by a `VarRequest` class. It is discussed in detail in the upcoming section on the implementation of the `DataEntity` class.

Another important aspect of accessing a data variable is the correct handling of the positions of the non-accessed dimensions. Multiple variables may share the same dimensions and since the mapping of the dimensions to the respective requests is controlled by the user, and since access to variables may be nested due to the lazy loading strategy, Insight has to ensure that the positions of all dimensions are in well-defined states at the beginning of the computation of a data entity, and that they are reset at the end of the computation. The correct handling of the positions of the dimensions is implemented as part of the `VarRequest` class, as well. In order to make the required function calls for checking the dimension mappings, setting and re-setting the positions of the unmapped dimensions, setting the access dimensions and cleaning it all up at the end more transparent, the `VariableAccess` class can be used. An object of this class handles the required function calls for a variable access in the correct order and provides access to the required data and grid information. In the normal case, the `VariableAccess` class requires a reference to a `VarRequest` in its constructor. It calls the `beginAccess` function of the `VarRequest` during construction, and the `endAccess` function during destruction.

2.5.4. The DataGrid and PosProvider classes

A `DataVariable` class may be associated with an object of the `PosProvider` or the `DataGrid` class. The `DataGrid` class inherits from `PosProvider`. Both classes provide additional information about the geographic position of the provided data. The interface of the `PosProvider` class only provides one function for querying the current position with respect to all current dimensional positions, whereas the `DataGrid` class offers additional access to the dimensions of the underlying grid.

A `DataGrid` object has up to four grid dimensions and maps the positions of these dimensions to three-dimensional vectors representing the associated geospatial position in form of a (lon, lat, P) triple. The interface also offers a function for querying the current grid index, that is, a vector containing the current positions of the grid dimensions with respect to the dimensions of the associated variable.

The `DataGrid` objects are composed of a position provider for the horizontal position, which shares the `PosProvider` interface, and of a `VerticalPosProvider` object providing the ver-

StaticPosProvider	This position provider outputs a static position. The static position is set to a fixed value during construction. This is useful in cases where the data consists of only one point or the data's position varies only vertically.
RegularPosProvider	This class outputs horizontal positions on an equidistant longitude/latitude-grid. It takes two dimensions and the positional bounds of each dimension as input during construction. Depending on the particular indices of the dimensions, the horizontal position is computed using these initially stated bounds.
VariablePosProvider	This class provides the horizontal position based on the values of two DataVariable objects representing the latitudinal and longitudinal components of the position. These variables are specified during object construction.
RotatedPosProvider	This is a decorator class, taking a PosProvider object and the position of a rotated pole during construction. The original position provided by the given PosProvider object is rotated by this decorator, such that the original north pole is aligned to the given position of the rotated pole. The idea behind such a rotation is to take advantage of the high horizontal resolution of regular lon/lat-grids near the poles in the specific area of interest of a data set.

Table 2.1.: The available horizontal position providers, derived from the PosProvider class.

tical position. The interface of the VerticalPosProvider returns only a scalar value (representing the pressure). Table 2.1 shows all available horizontal provider classes, table 2.2 shows the vertical position provider classes, derived from PosProvider and VerticalPosProvider, respectively. In table 2.3, all other position provider which provide a full three-dimensional position are listed. The tables contain a short description of each position provider.

2.5.5. The DataEntity class

Data entities are the fundamental building blocks of any data processing and visualization setup. They are responsible for the provision, the processing and the visualization of the data. Each data entity consists of input variable requests and input dimension requests (represented by VarRequest and DimRequest objects), output variables and output dimensions (DataVariable and DataDimension objects), as well as a set of properties in form of DataEntityProperty objects. The DataEntity class also contains functionality for the creation and propagation of update notifications whenever the properties of the entity or

`StaticVerticalPosProvider`	This vertical position provider outputs a single fixed pressure value.
`RegularVerticalPosProvider`	The vertical position is output at equidistant points within a fixed vertical interval. The interval bounds, as well as the dimension specifying the number of divisions and the current position are determined during construction.
`VariableVerticalPosProvider`	This class queries the current vertical positions from a `DataVariable` object which is passed to the constructor of the position provider.
`ScaledVerticalPosProvider`	This is a decorator class which scales the vertical position from another `VerticalPosProvider` object by a given factor. This is used for example for Pa to hPa conversion (Insight assumes all pressure values to be given in hPa.
`HybridPVerticalPosProvider`	Some meteorological data sets represent height by using the formula $a_k + b_k p(\lambda, \varphi, t)$, where k is the index of the model level, t is the time index, a_k, $b_k \in \mathbb{R}$ are parameter values given at each model level, and $p(\lambda, \varphi, t)$ is a (usually) time-dependent two-dimensional field representing the pressure at surface level. This provider takes two vectors containing the a_k and b_k values, as well as a `DataVariable` containing the surface pressure for the computation of the vertical position.
`MeterToPressureVerticalPosProvider`	Another decorator class which converts a vertical position given in meter to hPa, using the barometric height-formula (cf. Roedel and Wagner, 2011, their Section 2.1) with a sea level pressure of 1013.25 hPa. An additional scale factor can be specified in order to handle input data of different magnitude as well, for example data given in kilometers.

Table 2.2.: The available vertical position providers, derived from the `VerticalPosProvider` class.

ContourLines	This DataEntity class is also a PosProvider that provides the three-dimensional position of each vertex of the computed contour lines.
Model2dDataSource	This class is also both a DataEntity and a PosProvider, providing the three-dimensional position of the vertices of the lines defined in an external 2D model file.
PositionAdder	This DataEntity and PosProvider adds the positional data associated with a variable to another data variable. For this, the PositionAdder works as a decorator and passes on the original position to the new output variable.
TrajectoriesDataSource	This DataEntity class outputs the positions of trajectory data, read in from a file, through the PosProvider interface.

Table 2.3.: The available complete position providers, derived from the PosProvider class. All of them are also data entities, see Appendix B for more details.

the connection between entities changes. Additionally, functions for the serialization of the entity's current state are provided (see Section 2.8 for an overview).

Relation to the DataVariable and DataDimension classes

The DataVariable and their associated DataDimension classes fulfill two different tasks within a DataEntity object. They represent data which is *provided* by a data entity, and they provide access to the *requested* data in combination with the variable and dimension requests of an entity. In order to improve the program's structure, the DataEntity class is split up into the DataProvider and DataReceiver interfaces providing the functions associated with each different role. The DataReceiver interface contains functionality for the management of all requested data variables and dimensions. A data entity can request a set of input variables and input dimensions by providing the corresponding VarRequest and DimRequest objects. These objects contain additional information about the requests, for example the name of the request, the required positional information that has to be associated with an input variable, or the way in which unmapped dimensions are treated.

Automatic mapping of input dimensions

Each VarRequest contains an internal list of associated DimRequest objects. Whenever a DataVariable object is set as input for a variable request of the entity, the dimensions of the DataVariable object (the *input dimensions*) are mapped automatically to the DimRequest objects associated with the variable request. This match is based on the names of the input dimensions and dimension requests. Insight internally manages distinct lists of name prefixes associated with different dimension groups. Table 2.4 contains all currently implemented

dimension groups together with the corresponding name prefixes. The automatic mapping of dimensions is performed through the following steps:

1. Any input dimension whose name matches exactly the name of a dimension request is mapped to that request[31].
2. Any input dimension whose lower-case name matches exactly the lower-case name of a dimension request is mapped to that request.
3. The input dimensions and remaining requests are matched based on the groups their names belong to (cf. Table 2.4).
4. Any dimension whose lower-case name is completely contained in the lower-case name of a dimension request is mapped to this request.
5. Finally, all unmapped input dimension which do not belong to a dimension group are mapped to the still unmapped dimension requests, one after another. All unmapped input dimensions belonging to any dimension group are not mapped at all.

The user has always full control over the final dimension mapping. However, the automatic mapping of the input dimension often makes a manual adjustment unnecessary.

lon, **x**, **dimx**, **west_east**	longitude
lat, **y**, **dimy**, **south_north**	latitude
p, **z**, **lev**, **dimz**, **bottom_top**	vertical
t, **date**	time

Table 2.4.: The input dimension name prefixes and the groups they are associated with.

DataEntityProperty

Each `DataEntity` class possesses a list of `DataEntityProperty` objects representing the user options which influence the behavior of the data entity. Maintaining such a list of property objects, instead of just implementing different member variables in the individual sub-classes, simplifies many tasks and aspects of the handling of the different types of data entities. The serialization of the data entities, as well as the manipulation of the properties through a graphical user interface or through a script are decoupled from the individual data entity and can be handled uniquely for all current and future data entities. Properties can easily be added or removed from data entities without the need to adapt the GUI, the scripting code or the code used for the serialization of the entity. Different properties are represented by sub-classes of the `DataEntityProperty` base class. The property classes currently available in Insight are shown in Table 2.5.

The list of properties is usually created in the constructor of each individual sub-class of `DataEntity`. This initial list of properties may change at runtime. Such changes occur, for

[31] The matchings during these tests are performed by nested iterations over the input dimensions and the dimension requests. So if there were multiple potential matches, Insight always considers the first discovered match during these iterations.

`EntityBooleanProperty`	a boolean switch
`EntityColorMapProperty`	an arbitrary number of value/color pairs
`EntityColorProperty`	a single color
`EntityDoubleProperty`	a floating point value with double precision
`EntityFilenameProperty`	one or more filenames
`EntityIntegerProperty`	an integer value
`EntityMultiSelectionProperty`	a subset of currently selected values from a set of possible selections
`EntityNamedValuesProperty`	an arbitrary number of string/floating point number pairs
`EntitySelectionProperty`	one currently selected value from a set of possible selections
`EntityStringProperty`	a (typically short) string
`EntityTextProperty`	a string which is supposed to span across multiple lines
`EntityVerticalProfileProperty`	an arbitrary number of integer/floating point number pairs

Table 2.5.: The available sub-classes of the `DataEntityProperty` class and the option they represent.

example, for the selection of the fixed position of the unmapped input dimensions. The position of each unmapped dimension is selected through an additional `EntityIntegerProperty` property which is added to the entity's property list. These additional properties are called *dimension selections*, or *dim selects*, for short. The user can query and set the states of these dim selects through either the graphical user interface, or by using the respective commands of the scripting interface, just as it is the case for all other properties of the data entity.

Update notifications

The output of a `DataEntity` is completely determined by the mapped input dimensions and variables, as well as by the state of the provided properties. Due to the lazy loading principle followed by Insight, changes in the input mappings or the properties do not automatically trigger a recomputation of the entity's output data. The information about a change of the output data is, however, passed on to all affected entities by calling the `informSourceDataChanged` functions of the `DataVariable` objects. All connected entities receive a notification about the change of their input data, enabling them to decide if their own output data is affected by this as well. If so, the information is passed on in the same way. This leads to the information being passed on through the affected parts of the directed acyclic graph of connected entities, stopping at entities which have no output data or which are not affected by the specific change of input data. A flag of each `DataEntity`

indicates whether the computed data is still up-to-date and already computed or needs to be updated.

Insight allows the user to decide whether an actual recomputation of outdated data entities is triggered automatically or manually. After all entities are informed about the change of their input data, the recomputation is requested (in case of enabled automatic updates) by a call of the `recompute` function of the data entity which was the source of the changed data. In this function, the request for a recomputation is passed on through the entity graph the same way as the information about the source data change was passed on. As soon as a visual representation is reached, the actual `compute` function of the entity is called, which is responsible for the update of the entity data. This function also triggers the computation of all entities that provide input data for the computation. This "detour" of passing the information on to the leaf nodes of the graph, and then back to the source of the change when the computation is performed has two desired side-effects. First, we avoid an explicit iteration over all entities of the composition and have only to deal with entities that are really involved in the update process. Second, no computation is performed if there is no visual representation present (lazy loading strategy). The reason for the decoupling of the passing on of the information about source data changes and the passing on of the information about the wish for a recomputation, using the `recompute` function, is that all data entities need to know whether or not they have to update themselves before their output is used in any recomputations.

If the automatic recomputation is disabled, the user can manually force a recomputation. In this case, we iterate over all entities of the composition and call the *recompute* function of all outdated entities. The computation of an entity is also forced for all data entities that are involved in a netCDF file export.

Data entity types

An overview of the currently available data entities implemented in Insight can be found in Appendix B.

2.5.6. The DataComposition class

The directed acyclic graph of all required data entities for a certain visualization or data processing task, together with all input and output connections, often gets quite complex. The user therefore may create different separate *data compositions*, which are closed units containing any number of data entities and connections. Each such composition is internally represented by an object of the `DataComposition` class. All active compositions of Insight are grouped again in a `CompositionList` singleton object.

Since data compositions are closed units, they can be serialized individually, such that a single composition may be used in different applications of Insight. A common composition that is reused in multiple examples shown in this thesis, is the component responsible for drawing the map of the coastlines and countries of the earth (see for example Fig. 3.1 and Fig. 3.4).

The `DataComposition` provides functions for adding, removing, and for the connection of

DataEntity objects. The reason why the entity connections are set up in this class and not through the interface of the individual DataEntity objects is that the DataComposition can ensure the belonging of the involved entities to the same composition. In addition, it is responsible for checking whether or not a new connection would create a cycle in the directed graph of entity connections. Further responsibilities of the DataComposition objects are the serialization of all connections, the renaming of existing entities (ensuring unique names), the triggering of a manual recomputation of the whole composition, as well as the export of netCDF files.

Another feature of the DataComposition is the maintenance of an additional list of properties, the so called *composition properties*. These properties are a subset of all entity properties. The idea behind this is that the user can pick the most important options of a data entity setup such that these options can be shown and manipulated together at a single location of the GUI. Integer properties of equal name are unified, such that, for example, a single "time" selection property could be used to change the time settings of all other entities of the composition.

2.5.7. Data processing controllers, factories, and views

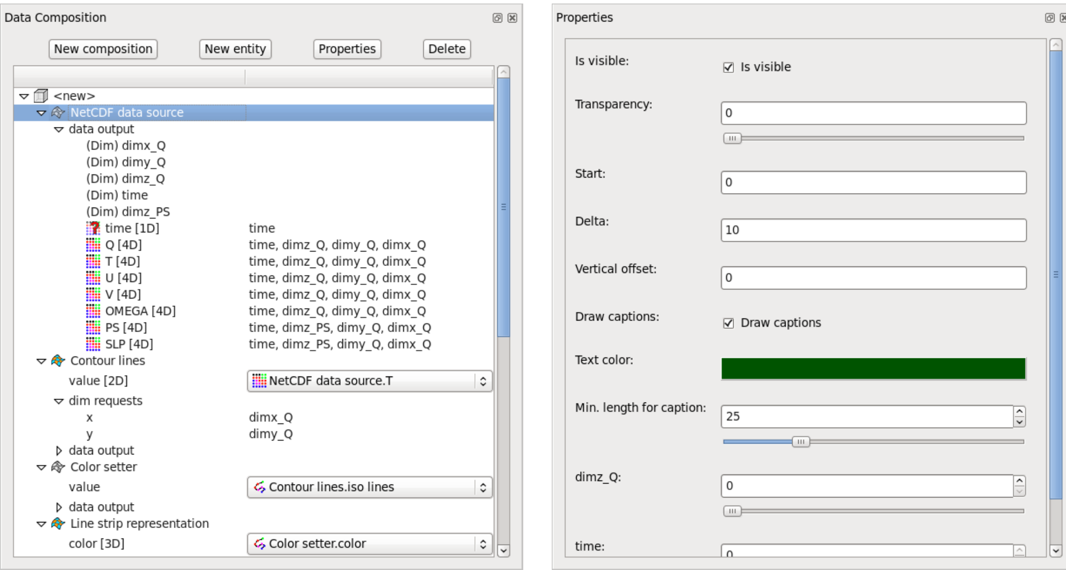
The previous sections contained a detailed description of the most important models of the data processing concept of Insight. In this section, we describe the remaining parts of the MVC architecture: the controllers and views of the data processing concept. In addition, the factory classes for the dynamic construction of several data models and views are introduced.

The views

Insight offers different views on the models of the data processing concept. The two major views are provided by the DataCompositionTreeView and the DataCompositionGridView classes. The DataCompositionTreeView provides a hierarchical view of all entities of a composition using a tree-view widget. The DataCompositionGridView visualizes data entities as boxes and their data connections as lines between the boxes on a two-dimensional grid. An example of the way the entities are visualized by the grid view can be found in Fig. 2.10 on page 30. The user can select entities by left-clicking on the gray boxes representing them. The smaller red and green boxes can be clicked to show dialogs providing options regarding the input mappings and the output data. Data connections are established by clicking and dragging a connecting line from the red output boxes to a green input box, then selecting the variables for each input request from separate drop-down list widgets.

The tree view of a data composition is shown in Fig. 2.17a. This was the first view of the DataComposition objects that was implemented in Insight. Entities are selected by clicking on their names, connections are established by using the drop-down boxes.

Each property has its own individual widget, derived from the Qt class QWidget. These widgets are responsible for the visualization of the current state of a property, and for the user interaction. In Insight, all property view widgets of either the currently selected data entity or the current data composition are created, managed, and visualized together by a single DataEntityPropertiesWidget object. As soon as the selection changes, the displayed



(a) Data composition tree view example

(b) Entity properties view example

Figure 2.17.: Views of data processing models.

list of properties changes accordingly. If no entity is selected, the properties of the current composition are shown.

The factories

The individual subclasses of the `DataEntity`, the `DataEntityProperty`, as well as the individual widgets for the properties are created by means of *simple factories* (see Freeman et al., 2004). A simple factory is responsible for the construction of different objects at runtime based on some given information. The `DataEntityFactory` class takes a type string identifying the type of the data entity and creates and returns the corresponding subclass of `DataEntity`. The `DataEntityPropertyFactory` class and the `PropertyWidgetFactory` do the same for `DataEntityProperty` subclasses and for their individual widgets (subclasses of `QWidget`). The advantage of having these factory classes is that they encapsulate the creation code in one class which can then be used every time a dynamic creation of the objects is required. For `DataEntity` objects, this is the case whenever new entities are added to a data composition, either through GUI interaction or via the Python scripting interface, as well as during serialization. The `DataEntityFactory` class additionally provides a list of all available data entity types, such that adding a newly implemented data entity class to Insight requires only changes inside the scope of the `DataEntityFactory` class. While most of the `DataEntityProperty` objects are directly created as members of a `DataEntity` class, the main task of the `DataEntityPropertyFactory` is the dynamic creation of different properties during runtime for the setup of user queries, which are a component of Insight's scripting interface (see Section 2.7.2). The `PropertyWidgetFactory` is used by the `DataEntityPropertiesWidget` for the dynamic creation of property widgets whenever a new data entity or data composition is selected.

The controller classes

The controller classes are a part of the MVC architecture. Their task is to provide an intermediate layer between the code that is responsible for drawing the GUI (the views) and the code representing the model. The main responsibility of a controller is the translation of GUI events (e.g. the clicking of a button) into the corresponding function calls of the model objects. The controller is also capable of directly invoking changes of the GUI as a response to a user action. The idea behind this additional layer is to prevent a direct dependency between the model and the view classes, such that, for example, a view may be used for the representation of different model classes (this additionally requires the view to access the state of the model through an abstract interface).

Since we use the Qt library for the design, the placement/layouting, drawing, and (sometimes) even for updating our GUI components, the view classes are either completely generated automatically by Qt, or contain automatically generated components, such that the source code visible to the programmer remains relatively compact. As described in Sect. 2.4.2, we therefore omit controller classes for simple user interactions, provided that we have a lean view class and the task itself consists only of simple calls of functions of the model. An additional controller class would lead to significantly more amounts of code compared to some additional functions consisting of only a few lines of code.

Insight uses the `QAction` and `QUndoCommand` Qt classes for the implementation of user commands, both in the controller classes and, in cases where the controller classes are omitted, directly in the corresponding views. In accordance with Qt's undo/redo mechanism, all basic tasks are realized by implementing subclasses of `QUndoCommand`. Each subclass encapsulates one single command, for example the creation of a data entity or the change of a specific entity property. These classes also contain code for undoing the command. After their instantiation, the objects are passed on to an instance of a `QUndoStack`, which automatically causes the execution of the respective command. While a `QUndoCommand` represents an invertible change of the current data processing model, an instance of a `QAction` class represents a general command available through the main menu, a context menu, a tool bar, and/or directly through a keyboard shortcut.

The available user commands related to the data processing concept can be split into four groups: Commands available through the "Composition" main menu or context menu, commands available through the "Entity" main menu or context menu, commands related with input mapping which are invoked mostly by the widgets and dialogs of the grid view and the tree view of the data composition, as well as commands for the changing of entity properties, which are invoked by the respective property widgets.

Entries of the main menus and context menus are implemented by instances of the `QAction` class. These actions are managed by objects of the `MenuDataCompositionController` and `MenuDataEntityController` classes. These controller classes contain also the code that is invoked when an action is triggered. Simple commands are often realized directly using the corresponding `QUndoCommand` classes, more complex commands may contain the display of additional dialogs and combinations of multiple `QUndoCommands` or direct manipulations of the models. For other user commands, e.g. the commands for the manipulation of entity properties, the views interact directly with the controller classes (in this example case with the corresponding `QUndoCommand` sub classes).

2.6. NetCDF file format handling

This section covers the handling of netCDF files in Insight. NetCDF files play an important role in meteorological and other scientific applications[32]. First, a short description of the netCDF file format is given. Afterwards, the netCDF import and the netCDF export features of Insight are described in separate sections. For a general introduction to the netCDF file format, refer to Sect. 2.4.3.

2.6.1. Description of the netCDF format

A netCDF file consists of *dimensions*, *variables* and *attributes*. An attribute has a name and a value that can be of different types (e.g. string, integer, float, etc.). Attributes are either global to the whole file or connected to a single *variable*. Variables also have an associated type and an arbitrary number of dimensions. Dimensions are either limited and have a fixed size, or are unlimited and have a current size. Older netCDF files support only one unlimited dimension per file, however, the new netCDF-4 file format also allows multiple unlimited dimensions.

The netCDF format does not enforce any standard way for the provision of meta information about the data stored in a netCDF file. Nevertheless, there exist several conventions for the usage of netCDF attributes in order to add meta information to a file[33]. The most common of these conventions are the *CF Metadata Conventions* (Lawrence et al., 2006).

2.6.2. NetCDF import

Insight is capable of handling netCDF data from many different sources. The import of a netCDF file into Insight and the creation of corresponding `DataDimension` and `DataVariable` classes is implemented straightforward. Difficulties arise when it comes to the extraction of information about the geographic positions and the setup of the related `DataGrid` instance. The problem is that many different ways exist in which the positional information of the data is provided, utilizing different attributes, variables and even separate, additional netCDF files with supplemental attributes and variables. Insight divides the acquisition of the positional meta information into the computation of the horizontal coordinates (λ, φ) and the computation of the vertical position (a pressure value p). All of Insight's netCDF variables are associated with a `DataGrid` object. As described in Sect. 2.5.4, the different implementations of these computations are distributed to subclasses of the `PosProvider` and `VerticalPosProvider` classes, which are used as components of the `DataGrid` class.

Insight possesses a set of configuration XML files containing descriptions of several types of netCDF files. These configuration files determine the respective horizontal `PosProvider` and `VerticalPosProvider` objects to be created for the initialization of the associated `DataGrid`.

[32] cf. `http://www.unidata.ucar.edu/software/netcdf/usage.html`, last accessed: 28-November-2012
[33] see `http://www.unidata.ucar.edu/software/netcdf/conventions.html`, last accessed: 18-June-2013, for a list of conventions

Each configuration file consists of three parts:

1. The root element <**INSIGHT−netcdf−format−file**>, containing attributes for global settings. The root element contains all the following child-elements.
2. The <**x−dim**/>, <**y−dim**/>, <**z−dim**/>, and <**time−dim**/> empty elements which define the names of all spatial and temporal dimensions.
3. Finally, the <**mapping**/> elements specify in all detail how the position of a variable with the given dimensions can be determined.

The <**x−dim**/>, <**y−dim**/>, <**z−dim**/>, and <**time−dim**/> have only a single attribute, name, which specifies the pattern of the corresponding dimension names. The wildcards *, ? and, [. . .] (for specific character sets) can be used. A detailed description of the allowed attributes of the other elements can be found in Table 2.6 and in Table 2.7.

Below follows an example of such a configuration file for a simple netCDF format with only one type of positional mapping.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <INSIGHT-netcdf-format-file name = "p/s-files (without 
      constfile)" priority = "1">
3     <x-dim name = "dimx_*"/>
4     <y-dim name = "dimy_*"/>
5     <z-dim name = "dimz_*"/>
6     <time-dim name = "time"/>
7
8     <mapping x-dim = "dimx_*" y-dim = "dimy_*" z-dim = "dimz_*" 
          time-dim = "*" horizontal-type = "regular" 
          x-min-attrib = "xmin" x-max-attrib = "xmax" 
          x-reach-max = "true" y-min-attrib = "ymin" 
          y-max-attrib = "ymax" y-reach-max = "true" 
          vertical-type = "regular" z-min-attrib = "zmin" 
          z-max-attrib = "zmax"/>
9 </INSIGHT-netcdf-format-file>
```

In line 2 of this example, all important global and meta information about the format are specified by the attributes of the <**INSIGHT−netcdf−format−file**> element. This information consists, in this example, of the name of the format and the priority tag. In the following lines 3 to 6, the names of the spatial and temporal dimensions are specified. The names may contain certain wildcards, and are used to categorize all dimensions of a file. Finally, line 8 contains the single positional mapping with all required information for the creation of the grid object of the data variables. All variables of the file are tested one-by-one on the compatibility of their dimension names with the names specified as attributes of the mapping elements. Only dimensions of a corresponding category are involved in each of these tests. If a dimension name attribute consists only of an asterisk, as it is the case for the **time-dim** attribute in this example, the corresponding dimension is not required for a match. If an optional dimension attribute is not given, a variable is not allowed to have a dimension of the corresponding category. The first positional mapping with matching dimension names is assigned to a variable. Therefore it is important to place more specific mappings in front of the more general mappings in the configuration file. The type of the horizontal and vertical

name (required)	An arbitrary name identifying the netCDF format.
priority (required)	The priority is used in the automatic detection process of the format of a file. See the section on format scoring for details.
const-file-name	The name of an additional netCDF file whose variables and global attributes can later be used for the positional mappings.
const-file-attrib	Instead of stating the name of an additional netCDF file directly through **const-file-name**, it can also be taken from a global netCDF attribute. The name of such an attribute can be given by this XML attribute.
pole-lon-var **pole-lat-var** **pole-lon-attrib** **pole-lat-attrib**	The position of a rotated pole is read in either from two netCDF variables or from two global netCDF attributes. These XML attributes define the names of the corresponding variables or attributes.

Table 2.6.: The available attributes of the <**INSIGHT−netcdf−format−file**> XML element.

mapping is in this example “regular”, which indicates that the longitudinal, latitudinal, and vertical grid dimensions are partitioned uniformly by the grid cells.

All valid XML files in the `netcdf_format` subdirectory of Insight are read in and parsed by individual `NetCDFFormat` objects upon program start. The `NetCDFFormat` classes has a function to test whether or not (and how good) a format matches a given netCDF file, and a function to create and associate all grid objects to a set of Insight variables.

Format scoring

Whenever a netCDF file is opened by Insight, the software checks all available netCDF format description objects on their compatibility with the netCDF file. The quality of the matching is represented by an integer score value. If the format is not applicable to the file, the score is set to -1. If all required prerequisites are met, for each variable that could be matched with a mapping, 10 points are added to the score (which is initially zero). At the end, the priority value, stated as an attribute of the <**INSIGHT−netcdf−format−file**> element is added to the score. This value becomes the decisive factor in cases where several format descriptions have the same number of applicable positional mappings. For example, if there is a format with disabled rotated pole, it should have a lower priority than the normal format description including the rotated pole. The current data entities for reading in data from a netCDF file all contain a property for the selection of the file format out of all formats with a positive score.

x-dim (required) **y-dim** (required) **z-dim** **time-dim**	The names, possibly containing wildcards, of the horizontal, vertical, and temporal dimensions a variable must possess in order to apply the current positional mapping. As for all other attributes, x usually refers to the longitudinal position, y to the latitudinal position, and z to the vertical position.
horizontal-type (required)	This attribute determines the way in which the horizontal position is computed. Possible values are either *variable* or *regular*.
vertical-type	This attributes determines the way in which the vertical position is computed. Possible values are *variable*, *regular*, and *hybrid*.
x-var **y-var** **z-var**	If the horizontal or vertical type was set to *variable*, these attributes determine the names of the netCDF variables which provide the respective positions.
horizontal-x-stag-dim **horizontal-y-stag-dim** **horizontal-z-stag-dim**	If the horizontal or vertical position is provided through netCDF variables, it may be the case that these variables have dimensions that do not match the dimensions of the original variable. Instead, if the position is given on a staggered grid, the dimension sizes might be off by ± 1. If these dimensions with alternative sizes are specified through these attributes, the direct variable access is replaced by multiple accesses for querying the values at the nearest grid intersections, followed by a linear interpolation.
x-min x-min-attrib x-max x-max-attrib x-step-attrib x-reach-max y-min y-min-attrib y-max y-max-attrib y-step-attrib y-reach-max z-min z-min-attrib z-max z-max-attrib z-step-attrib	If the horizontal or vertical position type is set to *regular*, these attributes control the range of the grid. While the ***-min*** attributes control the start value, the mutually exclusive ***-step-attrib** and ***-max*** attributes define the extent of each dimension.
hybrid-a-var **hybrid-b-var** **ps-var**	If the vertical position type is set to *hybrid*, these attributes specify the names of the variables to be used for obtaining the a_k, b_k, and $p(\lambda, \varphi, t)$ variables used by the `HybridVerticalPosProvider` (cf. Table 2.2).
z-unit	This attribute indicates that the vertical component of the position is given in another unit than hPa and therefore has to be converted. Valid values are *Pa*, *m*, and *km*.

Table 2.7.: The available attributes of the <**mapping**/> XML element.

2.6.3. NetCDF export

Insight is capable of creating output netCDF files containing the values of selected output variables. The export command is provided through the MenuDataCompositionController class. This is one of the reasons why Insight requires all variables of one export to come from the same data composition. The major part of the implementation is realized by means of the NetCDFExporter class.

Dimension remapping

Several difficulties had to be addressed for the implementation of the netCDF export. First of all, netCDF files do not support data dimensions of variable size. Therefore, the netCDF exporter transforms the output data of variables with dimensions of varying sizes into data arrays with fixed dimension sizes. If dimensions of varying size are present, Insight has to determine their maximum size. The general implementation of a DataDimension with parent dimensions for obtaining the maximum size is to search for the maximum by iterating over all of the possible positions of all parent dimensions. The corresponding netCDF dimensions are created with this maximum size, which may lead to a significant increase of memory usage for storing the data of the variable. Data at formerly invalid dimension positions is replaced by a fixed missing data value, also known as *fill value*.

Additional dimensions from integer properties

Another difficulty arises from the fact that many data entities create selection properties for the positions of unmapped input dimensions instead of passing the dimension on to the output. A prominent example is the GridMapper entity which creates a new set of output dimensions for the newly created data grid, and replaces all unmapped input dimensions, for example an unmapped time dimension, with corresponding dimension selection properties. However, such unmapped dimensions should be available for export as well. The implementation of the NetCDFExporter therefore allows for the creation of temporary output dimensions that can replace any property of the type EntityIntegerProperty of the entity providing the output variable. These temporary dimensions can then be exported along all other output dimensions associated with that variable. The calculation of the dimension sizes for the creation of the netCDF variable has to take into account that the sizes of the output dimension may depend on the currently selected positions of these dim selects.

Splitting of large output files

The last difficulty to overcome is the limited file size that output netCDF files can have. Several circumstances impose limits on the maximum size a netCDF file and the contained variables can have. The first limiting factor is the total file size the file system is able to handle. On some 32-bit systems the maximum file size is limited to two gigabytes by the usage of signed 32-bit values as file offsets. The second factor is the offset used to store the beginning position of variables inside the netCDF file. Classic netCDF files exist in two different versions, *version 1* (or *CDF1*) with 32-bit offsets and *version 2* (or *CDF2*) with

64-bit offsets[34]. Insight currently exports CDF1 files with signed 32-bit offsets, which means that all variable records must start within the first two gigabytes of the file. A third limiting factor is the size of the `size_t` type of the C platform the netCDF library was compiled on. Due to the way the netCDF library is implemented, this size limits the total size of a regular variable without an unlimited dimension to a little less than four gigabytes. So for very large data sets it is not possible to export all of the data as one CDF1 file. In order to avoid files with such large variables, Insights offers an option to split output files into a sequence of smaller files. However, this splitting is currently limited to the splitting of a single additional dimensions created from an integer property. Such a dimension can be divided into regular sized intervals, for which all variables are exported separately. A general implementation allowing the subdivision of all types of dimensions is subject of future work, as well as the support of other file formats which support variables of arbitrary sizes.

2.7. Scripting support

This section deals with the several different applications of the Python scripting language, which is an integral part of Insight. All core functionalities of Insight, such as the creation of data compositions, their management, the creation and manipulation of data entities together with their properties and data connections, as well as the controlling of viewports and their individual 3D camera setups are accessible through a set of Python functions and classes. In addition, the data provided through the output variables and dimension can be queried, which allows for flexible post-processing of the available output data (e.g. statistical analysis or individual serialization in form of text or binary files).

The following list gives an overview of the different applications of the Python scripting language within Insight:

1. Insight possesses an integrated Python interpreter for the direct execution of Python statements from within Insight's user interface.
2. Files containing Python code can be executed via a corresponding command of the main menu, or by passing the filename of the script to Insight in form of a command line parameter.
3. Insight uses a set of short scripts, so called *manipulators*, for providing additional user commands for the manipulation of data entities. These manipulators are associated with a set of entity types and can be accessed through the context menus of each individual data entity.
4. Insight offers the export of automatically generated Python scripts for the serialization of the current program state.
5. The "Scripted operation" data entity allows the user to write a Python function which computes an output value out of the set of input data values of the entity.

This section is divided into two parts, each of which presents a different perspective on the scripting support of Insight. The first one is the user's perspective on the provided Python functions and classes for controlling Insight. For this, we cover the basic ideas

[34] see `http://www.unidata.ucar.edu/software/netcdf/docs/faq.html#lfs`, last accessed: 18-June-2013

and functionalities of the two Python modules provided by Insight: `insight_core` and `insight`.

The second perspective provides a technical view on the realization of the scripting support, describing the integration of Python into Insight's C++ code. We show how Python bindings of C++ functions are provided for a set of important core functionalities, and how the actual `insight` Python package is implemented on top of these core functionalities, making use of some of Python's special language features. In addition, we show how Python scripts are invoked from within Insight and how the Python shell is integrated into Insight's user interface.

2.7.1. The user's perspective

The two most common ways in which a user can invoke Python code within Insight is either through the interactive scripting console (also called *Python shell*) which is integrated into the GUI, or by writing a separate script file that contains the code to be executed. These files can either be passed to Insight on startup at the command line, or they are invoked from the corresponding command in the "Scripts" menu.

All commands and script file that are passed to Insight are executed through one single object providing an embedded Python interpreter. In the interpreter's global namespace some Python modules and packages (`sys`, `insight_core`, and `insight`) are automatically imported on startup. `insight_core` provides all available fundamental functionalities of Insight as a set of plain global functions. These global functions are imported into the global namespace, such that they are directly accessible via the respective function name without the `insight_core.` prefix. The `insight` module provides a set of Python classes to give the user a more accessible interface for interaction with Insight. For example, the classes `insight.composition.Composition` and `insight.entity.Entity` can be used for the representation and manipulation of data compositions and data entities. These two classes are also automatically included into the global namespace, such that they can be referred to directly as `Composition` and `Entity`[35].

Below, we will give some fundamental examples of how the functions of the `insight_core` module and the classes from the `insight` package can be used to control Insight and to create scripts for all kinds of different tasks. A full reference to all core functions and classes of the `insight` package can be found in Appendix C. All of the following commands can be directly entered into the Insight Python console. At the end of this section, we give a general outlook on how the scripting functionality and some of Python's standard library features can be used for the implementation of a rudimentary WMS[36] server.

Data compositions

At startup, Insight possesses exactly one empty data composition. A data composition is identified through its unique name, which is `<new>` in case of the first default composition.

[35]Internally, the `Composition` and `Entity` attributes refer to factory functions returning unique object instances for representing each data entity and composition. See Sect. 2.7.2 for a detailed description.

[36]Web Map Service, a protocol by the Open Geospatial Consortium (OGC) for the exchange of two-dimensional, georeferenced maps (de la Beaujardiere, 2006).

The `Composition` class can be used to represent any existing composition, or to create a new one. The constructor takes the name of the composition as its only argument. If no name is specified, the argument is set to the name of the currently selected composition, if present, or to `<new>`, if there is no current selection. In any case, if the name passed to the constructor belongs to any existing composition, the new object corresponds to the respective composition. If there is no composition with the selected name, a new composition of that name is created.

For example, the command

```
comp = Composition()
```

creates a new object `comp` referring to the current composition. You can check this by typing either

```
comp.name
```

or

```
print comp
```

which both output the default name `<new>`. To print out a list of the names of all entities of the respective composition, one can use the following command:

```
comp.entities
```

This list is, of course, empty at the beginning by default.

The composition object allows the addition and removal of entity properties through the `add_property` and `remove_property` functions. A list of all current entity properties can be accessed through the `properties` property of the `Composition` objects. These properties, however, contain instances of `insight.entity_property.Property` classes for representing each property. In order to get an overview of the names of the properties one could type

```
[p.name for p in comp.properties]
```

Of course, this list is empty in case of the initial default composition.

Data entities

For the creation, referencing, and manipulation of data entities, the `Entity` class is provided. It's constructor takes the arguments `type`, `name`, `comp`, and `id`. In order to create a new entity, at least a string identifying the desired type of the entity needs to be passed to the constructor. For example, if we want to create a new "Color mapper" entity, we can invoke the following command:

```
Entity("Color mapper")
```

The command

```
core.list_entity_types()
```

displays a list of all available types of entities. In order to refer to an existing entity, we have to identify it by either passing an existing name as `name` argument to the constructor of the `Entity` class, or by specifying the unique ID of the respective entity. Often it is desirable to assign the object to a custom variable, as in the following command:

```
cm = Entity(name = "Color mapper")
```

This command assigns an `Entity` object representing the existing "Color mapper" entity (whose name is "Color mapper" as well) to the Python variable `cm`. In order to demonstrate some ways of connecting data entities, we need to create a second data entity.

```
sg = Entity("Simple grid")
```

The interface of the `Entity` objects provides access to the requested input dimensions and input variables, as well as to all output dimensions and output variables.

```
# accessing all requests of the "Simple grid"
sg.var_requests
sg.dim_requests

# the available output of the "Color mapper"
cm.vars
cm.dims

# explicit access to the output and input requests
cm.get_var("color")
cm.get_dim("component")
cm.get_dim("component").size

sg.get_var_request("color")
sg.get_var_request("color").dim_requests
sg.get_dim_request("Color component")
```

The properties of any data entity can be accessed in a very similar way.

```
# output of all property names of the "Color mapper"
cm.properties

# accessing a single property
ts = cm.get_property("Text size")

# querying the value of the property
ts.get()
```

Dynamic attributes of the Entity class

As a shortcut for providing convenient access to the input requests, the available output, and to the properties of any data entity, the `Entity` class uses a mechanism based on dynamically provided class attributes. The idea is to allow simple commands as in the following example.

```
# change the "text size" property of the "Color mapper"
cm.text_size = 23

# connect the "color" output of the "Color mapper"
#   to the "color" input request of the "Simple grid"
sg.color = cm.color
```

In order to realize this kind of commands, all names of the objects to be accessed are converted into suitable attribute names first. This is done by converting all characters to lower-case, removing unsuitable characters such as ",", ":", and any brackets, and by replacing all spaces with underscores. Whenever an attribute of an `Entity` class is queried, the following objects are checked in sequence: all "native" functions and attributes of the `Entity` class → properties → output variables → output dimensions → variable requests → dimension requests. As soon as a match is found, the respective object is returned. Whenever any object is hidden by any other object of the same converted name, the user has to fall back to the explicit `get_`...-functions. When an attribute is to be set, the converted names of the same objects are compared to the attribute name in the same sequence, except for output variables. Output variables are skipped because there is no meaningful application for assigning a value to them.

Data entity properties

Though there are several different types of properties an entity may possess, all of them are controlled through a unique interface. Properties are represented by `insight.entity_property.Property` classes. For easier querying and setting of properties, each property provides one *primary* attribute representing its current value. This could be the selected value of an integer-property, or a string representing the current selection of a selection-property. In addition, each property provides a dictionary containing all of its attributes, through which additional information, for example the maximum and minimum allowed value of an integer-property or the set of all allowed selections of a selection-property can be queried. Some of these additional attributes of a property can also be set by the user. The following example demonstrates different ways to access and to manipulate a property.

```
# examples for querying the values of a property
print cm.legend_pos
cm.legend_pos.get()
cm.legend_pos.get_ext()
cm.legend_pos.get_ext("availableOptions")

# examples for setting the values of a property
cm.legend_pos = "none"
cm.legend_pos.set("rightmost")
cm.legend_pos.set_ext([("this",0), ("is", 1),
                       ("just", 2), ("a",3), ("test",4)],
                      "availableOptions")
```

Input/output connection

One way to connect a variable requests with an output variable is to use attribute assignment on an `Entity` object as seen above. If this is not applicable, the explicit `connect` function of the variable request can be used.

```
# the following functions are equivalent
sg.color = cm.color
sg.color.connect(cm.color)
sg.get_var_request("color").connect(cm.get_var("color"))
```

To remove any connection, the `disconnect` function can be used. Alternatively, one can assign `None` to the input variable attribute.

```
sg.color = None
sg.color.disconnect()
```

As usual, the dimensions are mapped automatically if a corresponding match is found. To manually change the mapping of the input dimensions, a list of dimensions can be assigned to any input dimension attribute of an entity, or the explicit `set_mapping` function of an input dimension request object can be called. In the latter case, an additional boolean `combineDims`-argument specifies whether multiple dimensions are projected into one sequence (`True`) or are accessed simultaneously (`False`). See Section 2.5.2 for details on the mapping of multiple dimensions to a single request.

```
# the combineDims flag is ignored for less than two dimensions
sg.color_component.set_mapping([], combineDims=False)
sg.color_component = [cm.component]
```

Output variable access

The value of an output variable depends on the current position of all of its dimensions. The variable values can be queried through the interface of the output variable objects. For this, either the `value` property or the `get` function of these objects can be used. The access must be preceded by a call to the `begin_access` member function and succeeded by a call to the `end_access` member function. These functions allow the data of the output variables to be computed and prepared for imminent access (e.g. loaded into memory), and to be cleaned up at the end of the access.

```
cm.color.begin_access()
cm.color.value
cm.color.end_access()
```

Note that the color mapper only outputs missing data values (-999.9) in our example, since there is no input connected to it. The output variable has only one output dimension in our case, the "component" dimension of size three. All values of the color variable can be output in the following ways.

```
cm.color.begin_access()
```

```
# dimension control using member functions
cm.component.reset()
while cm.component.valid:
    print cm.color.value
    cm.component.next()

# dimension control using get
print cm.color.get(0), cm.color.get(1), cm.color.get(2)

# dimension control using named parameters of get
print cm.color.get(component=1)

cm.color.end_access()
```

Sometimes it is necessary to explicitly state that a data dimensions won't change rapidly during data access. This way, Insight can adapt its caching strategy accordingly. In some cases, this information is required in order to access large data variables without running out of memory.

```
cm.color.begin_access()

# we set the "component" dimension after the begin_access() call
#   but prior to the first access
cm.color.fix_dim(cm.component, 1)
cm.color.value

cm.color.end_access()
```

Viewports and camera control

The viewports, the applied coordinate transformation, as well as all camera settings (position, projection, etc.) can be directly controlled through the global functions of the `insight_core` module. For example, the following commands query two lists of all available options and then change the current options of the coordinate transformation and the projection. In addition, the camera view, the global translation and the zoom factor of the current active viewport are set.

```
list_transformations()
list_projections()

set_transformation("stereographic north pole")
set_projection("ortho")

set_translation((0, 0, 0))
set_camera((0, 90, 0))
set_zoom_factor(2)
```

Insight supports multiple viewports. The scripting interface enumerates the viewports starting from zero. Whenever a viewport is added or removed, the enumeration is adapted. As a consequence of this, the number associated with a viewport may change if a lower-numbered viewport is removed. All viewport related functions can be called with an additional integer parameter which indicates a specific viewport to be affected by the function call. The current number of viewports can be queried.

```
# we add a second viewport ...
new_viewport()

# ... and query the number of viewports
get_viewports_count()

# change of properties of different viewports
set_background_color((1,0,0), 1)
set_background_color((0,1,0), 0)

# removal of a viewport
remove_viewport(0)
```

There is also a command for taking screenshots. The file format of the screenshot is defined through the given file extension. A wide range of formats, for example `.png` and `.jpg`, are supported. The second and third parameter specify the resolution of the screenshot. These parameter are optional. This function can also be applied for any specific viewport.

```
take_screenshot("image.png", 800, 600)
take_screenshot("another_image.png", 0)
```

Exporting netCDF files

For the export of a netCDF file containing a set of output variables, the `export_netcdf` function can be used. The function requires two parameters: the name of the netCDF file to be created and a list of all output variables to be included into the file.

```
export_netcdf("output.nc", [cm.color])
```

Optional parameters are `int_property_dim`, which takes an integer property object and adds it as an additional dimension to the file, and `slice_size`, which allows the splitting of the output into multiple files, each of which contains the given number of steps of the additional dimension, see Section 2.6.3 for details.

This function is a small wrapper around the `insight_core._export_netcdf` function which requires some additional arguments and cannot handle variable objects directly, see Appendix C for details.

Example application: Implementation of a simple Insight-WMS-server

The previous sections contained a general overview of the available functionality of Insight's Python interface. In this section, we give a short outlook on a possible application: An

Insight Python script for the creation of a rudimentary WMS server.

The idea of a WMS server is to provide access to visualizations of geographical data in form of two-dimensional images by means of answering HTTP requests with a defined set of parameters. The parameters are passed to the server in form of a query string using the HTTP-GET method. In our short example here, we will only use a subset of all possible parameters offered by the WMS specifications. A more detailed introduction to WMS is given in the corresponding Section 4.2.7 of the IWAL chapter.

We begin the implementation with the setup of the data entities required for drawing the tiles of our map. In this example, we visualize the temperature variable of a short time series of analysis data, available as netCDF files with the names `P20070114_00`, `P20070114_06`, `P20070114_12`, and `P20070114_18`. The basic visualization setup is prepared by the following script commands[37]. It shows the setup of the visualization of the temperature variable. In the final script, we prepare additional visualizations (coast lines and contour lines).

```
# prepare the data source, the color mapper, and the simple grid
ncdf = Entity("NetCDF time series source")
ncdf.netcdf_file =
    "P20070114_00;P20070114_06;P20070114_12;P20070114_18"

cm = Entity("Color mapper")
cm.mapping = rainbow.transformed(-20, 30)
cm.legend_pos = "none"

sg = Entity("Simple grid")
sg.fixed_vertical_position = True

# Connect the entities
cm.datain = ncdf.t
sg.color = cm.color
```

The `rainbow` object used for the setup of the color mapper, is a global object from the `insight.colormaps` module that is imported into the main namespace during start-up of the interpreter. For more details, see Appendix C.

A WMS-request contains the definition of the bounding box of the image to be plotted, as well as the target image resolution. Our script has to map the bounding box coordinates into a corresponding setup of the respective viewport properties. First of all, we perform a general setup of the viewport that remains constant during the production of all plots.

```
# invariant viewport settings
set_draw_cube(False)
set_draw_axes(False)
set_background_color((0,0,0))

set_camera((0,90,0))
set_transformation("xyz-box transformation")
```

[37] The example code of this section contains excerpts of the full final script for the WMS server. The complete script can be found in Appendix D.

```
set_projection("ortho")
```

If we assume a quadratic viewport (e.g. a screenshot with an aspect-ratio of one) without any translation and a zoom factor set to one, the orthographic projection of this setup leads to a visible area which covers OpenGL coordinates in the interval $[-1, 1]$ in both horizontal and vertical directions. The `xyz-box transformation` maps the geographic coordinates of the data to OpenGL coordinates without any modifications. Due to a hard-wired, global scaling factor of 0.05[38], the visible area corresponds to geographic coordinates in the range from 20°W to 20°E and from 20°S to 20°N. The size of these intervals is inversely proportional to the zoom factor setting, that is a zoom factor of 2.0 results in intervals of half the size. With this information at hand, we are ready to define a function that performs the remaining, dynamic setup of the viewport with respect to the given bounding box and image resolution.

```
# dynamic viewport settings
def prepare_camera(resx, resy, left, right, top, bottom):
  ratio = float(resx) / resy

  x_scaling = float(top - bottom) / (right - left) * ratio
  set_scaling((x_scaling, 1.0, 1.0))

  zoom_factor = 40.0 / (top - bottom)
  set_zoom_factor(zoom_factor)
  set_translation((-ratio / zoom_factor - left * x_scaling / 20.0,
                   1.0 / zoom_factor - top / 20.0,
                   0))
```

The `x_scaling` factor is introduced to compensate the differences between the aspect ratios of the bounding box and the target image resolution. The `zoom_factor` and the translation vector are computed such that a screenshot of the given resolution covers exactly the specified area, taking into account the current transformation and projection modes as mentioned above.

The setup of a basic HTTP server is very simple in Python. For our demonstration, we use the `HTTPServer` class from the `BaseHTTPServer` module that is part of Python's standard library. The constructor of the class requires a `BaseHTTPServer.BaseHTTPRequestHandler` class (not object!) as a parameter. We implement our own derived class for request handling, which overrides a single method named `do_GET`. This method is invoked every time a GET-request is sent to the server. For the parsing of the parameters passed as a query string, we use two additional methods from Python's standard library, namely `cgi.parse_qs` and `urlparse.urlparse`.

First, we need to import all required modules.

```
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
from cgi import parse_qs
from urlparse import urlparse
```

Now we are ready to implement our custom HTTP request handler.

[38] This scaling factor is a “historic” relict from the original `Vis3d` implementation.

```
class MyHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    params = dict([[k, v[0]] for k, v in
        parse_qs(urlparse(self.path)[4]).iteritems()])

    # obtain the bounding box borders and the resolution
    bbox = [float(v) for v in params['BBOX'].split(',')]
    resx = int(params['WIDTH'])
    resy = int(params['HEIGHT'])

    prepare_camera(resx, resy, bbox[0], bbox[2], bbox[1], bbox[3])

    if params['LAYERS'] == 'map':
      take_screenshot('wms_image.png', resx, resy)
    else:
      self.send_error(500)
      return

    # return the image
    image = open('wms_image.png', 'rb')
    self.send_response(200)
    self.send_header('Content-type', 'image/png')
    self.end_headers()
    self.wfile.write(image.read())
    image.close()
```

In the code excerpt above, we only provide a single type of visualization, indicated by the named parameter `LAYERS` set to `map`. The full example script in Appendix D contains additional map layers, as well as separate handlings of the two mandatory WMS requests `GetMap` and `GetCapabilities`.

At the end of the script, it remains to start the server itself:

```
try:
  server = HTTPServer(('', 8000), MyHandler)
  server.serve_forever()
except KeyboardInterrupt:
  server.socket.close()
```

If we put this code into a script file called `wms_script.py` and place it in the directory of the Insight installation, we are able to start the server with the command `./insight wms_script.py`. The server runs on port 8000 until it is terminated by ctrl+c. It answers queries such as the following[39]:

`http://localhost:8000?LAYERS=map&BBOX=-180,90,180,-90&WIDTH=1024&HEIGHT=512`

[39]This query is not a strict WMS query, since the mandatory parameters `VERSION`, `CRS`, `REQUEST`, `STYLES`, and `FORMAT` are omitted.

2.7.2. Technical realization

The integration of the Python language into the C++ source code of Insight is realized using the Boost.Python library, see Section 2.4.3. Boost.Python itself is internally based on the Python C-API, which is an integral component of the Python language[40]. The integration of Python via the Boost.Python library can be divided into two different main parts:

1. The *export* of a set of core functions for the control of Insight's visualization pipeline in form of a custom Python module called `insight_core`.
2. The embedding of a Python interpreter into Insight for the execution of user scripts, macros, and single Python commands.

Another important building block of the Python integration is a custom package named `insight`, which provides several modules purely written in Python. These modules contain Python classes which use object-oriented programming concepts and some of Python's special language features for the provision of a more convenient and easier access to the functionalities of the `insight_core` module. For example, the output and input variables, as well as the properties of a data entity can directly be addressed through custom functions for getting and setting arbitrarily named attributes of an object instance.

Most of the Python-related source code of Insight can be found in the `scripting_models` and `scripting_views` subdirectories of the Insight source directory. The former directory contains all the core classes and function required for the provision of the Python bindings, as well as the source code of the `insight` Python package. The latter directory contains code for the Python console, which is implemented as an element of Insight's GUI.

In the next parts of this section, the C++-classes responsible for the export of the `insight_core` module are described. Following, we discuss the design of the components of the `insight` Python package in detail. Finally, we present the important C++-classes for the embedding of the Python interpreter in Insight, and show how this embedding is used for the execution of scripts, the implementation of the interactive Python console and the "Scripted operation" data entity, as well as for the realization of the manipulators.

The `insight_core` module

The `insight_core` module consists of a set of Python functions and classes that are implemented in C++ and exported to Python. All interaction between Python code executed by the embedded Python interpreter and the data background of Insight is channeled through this module.

Most of Insight's functionalities are accessible through a set of over 80 functions that are directly available at the global scope[41]. Examples of these functions are `new_entity` for the creation of new data entities, `list_output_vars` for the listing of all output variables of an existing data entity, `connect` for connecting an output variable to a variable request, as well as functions like the `take_screenshot` function. The parameters and return values of these functions are basic Python types such as integer values or strings, as well as composite types

[40] cf. `http://docs.python.org/c-api/index.html`, last accessed: 30-August-2012

[41] The embedded Python interpreter in Insight imports the whole `insight_core` module into the global namespace, such that these functions are directly available without the module name as a prefix.

(e.g. lists and tuples) containing basic Python types. Objects of Insight's data background, such as compositions and entities, are represented by combinations of basic Python types in the following ways:

- *Data compositions* are identified by their unique name (a string).
- *Data entities* are identified by the name of the composition and their unique ID (a string and an integer value).
- *Data entity properties* are identified by the composition name, the entity ID, and the name of the property (string, integer, and another string).
- *Output variables* and *output dimensions* are identified by the composition name, the entity ID, and the name of the output variable or output dimension (string, integer, string).
- *Variable requests* and *dimension requests* are identified by the composition name, the entity ID, and the name of the variable request or dimension request (string, integer, string).
- *Viewports* are identified by their unique ID (integer).

An overview of all core functions of the `insight_core` module can be found in Appendix C. In addition to these functions, the module contains the `PythonObserver` class, which provides an interface for connecting Python classes to Insight's event notification system. Such connections are used by classes of the `insight` package to be informed about state changes, for example about the change of the name of a data composition.

Another class of the `insight_core` module is the `UserQuery` class which provides an input mechanism primarily used by manipulator scripts. A set of properties can be associated with each query through a corresponding member function. Another function triggers the showing of a modal dialog window containing the visual representations of these properties. The properties are identified by a type string and are dynamically created on the C++ side using a `DataEntityPropertyFactory` object. After the user has selected the desired values of all properties and has closed the dialog, the Python script can query these selected values and use them for the continuation of the script.

The `insight_core` module provides two additional auxiliary classes, `OutputHook` and `InsightPythonException`, which are described in detail, together with all other classes of this module, in Appendix C.

The `insight` package

The `insight` package consists of several modules providing Python classes for the representation of data compositions, data entities, data entity properties, output variables and dimensions, as well as input variable and dimension requests. In addition, it contains several other classes, for example for the representation of different color mappings to be used as input for a `EntityColorMapProperty` setter. In this section, we focus on the main classes for representing objects of the data processing pipeline. For details on the other members of the package, see Appendix C.

The implementations of the classes for representing data compositions and data entities share a common approach: The core functionalities for representing and manipulating data compositions and entities are realized in the `_Composition` and `_Entity` Python classes, whereas the identifiers `Composition` and `Entity` refer to factory functions creating the actual composition and entity objects. These factory functions contain code for avoiding multiple object instances representing the same composition or the same data entity. Both the `_Composition` and the `_Entity` classes are derived from the `insight_core.Observer` class, which connects instances of these objects to Insight's notification system. This way, the objects are capable of reacting on events invoked by the C++-objects representing the respective compositions or entities. Sending an event notification is, however, a very common task that is carried out frequently during the processing of data. The handling of these events is carried out significantly more slowly in a Python object than in other C++-objects. Therefore, it is important to keep the number of Python objects to be informed about an event at a minimum, which is the reason for having the separate factory methods. For the sake of simplicity, we refer to `Composition` and `Entity` objects, whenever we actually mean `_Composition` and `_Entity` objects created by the `Composition` and `Entity` factory functions.

The `Composition` and `Entity` objects provide a set of member functions and properties for performing the tasks that are originally available through the `insight_core` module. These member functions require fewer parameters than the corresponding core functions, since the parameters for the identification of the objects can be omitted. This is because the `Composition` and `Entity` objects internally store the required information to identify themselves.

Python allows a class to provide custom functions for the getting and setting of dynamic attributes of an object. The custom getter function, `__getattr__`, is called if no instance attribute of the requested name already exists. Some classes of the `insight` module use the custom attribute getter functions for providing access to components that can dynamically change at runtime, for example to the properties of a data entity. Since the features that are to be accessed can have any name storable in a C++-string in general, but an attribute identifier is limited to characters, numbers (not as the first character), and underscores in Python, the names are converted into valid Python identifier using the `convert_name` function from the `insight.globals` module. This function transforms a given string to an all lower-case version, replaces all spaces by underscores, and removes all invalid characters. For example, the name “Core area top (hPa)” of a property of the “Complete segmentation (4D)” entity would be transformed into the Python literal `core_area_top_hpa`.

The custom getter function for attributes is used by classes of the `insight` package in the following ways:

- `Entity` objects try to match the attribute name with the converted names of their *properties*, their *output variables*, their *output dimensions*, their *variable requests*, and their *dimension requests* (in this order). If a feature with a matching name is found, a corresponding `Property`, `OutputVar`, `OutputDim`, `VarRequest`, or `DimRequest` Python object is created and returned.
- `OutputVar` objects search for dimensions of the variable with the given attribute name and return corresponding `OutputDim` objects, if found.
- `VarRequest` objects search for requests of dimensions with the given attribute name and return corresponding `DimRequest` objects, if found.

The custom setter function for attributes, `__setattr__`, is used by the `Entity` class in the following way: The entity matches the attribute name with the *properties* of the entity first. If a match is found, the property is set to the value passed to the attribute setter. If no property is found, the *output dimensions* are searched. The position of any dimension with a matching name is set to the given value, which is assumed to be an integer. If no matching dimension exists, the *variable requests* are searched for a request with a matching name. If such a request exists, the value passed to the setter is assumed to be an `OutputVar` object and this output variable is connected with the request[42]. Finally, if no variable request matches the attribute name, the *dimension requests* are searched, and if a matching name was found, the value is supposed to be a single `OutputDim` object or a list of `OutputDim` instances to be connected to this dimension request.

Embedding of the Python interpreter

For the execution of Python code from within Insight, we use the Python C-API, as well as Boost.Python in order to prepare and operate the embedded Python interpreter. There is only one instance of the Python interpreter running in Insight which executes all scripts and commands stated in any of the ways described in the previous sections. This instance is created as part of the single `ScriptingInterface` object of Insight. The setup of a Python interpreter is straight-forward. First, the interpreter is initialized by a call of the `Py_Initialize()` C-function. After that, the `insight_core` module is prepared by calling the `initinsight_core()` function. This function is automatically created by the Boost.Python macros that are used for the export of the functionality of the `Core` class of Insight.

By default, the output of the Python interpreter is connected to the `stdout` stream. For the scripting console of Insight to work correctly, we need to redirect the output to our own text widget of the GUI window displaying the scripting console. For this, the default `sys.stdout` and `sys.stderr` objects of Python are replaced by an instance of the `insight_core.Output` class. This class has a `write` method which is a Python binding of the corresponding C++-function that stores the output in a member variable accessible from within the implementation of the `ScriptingConsole` C++-class. The original `sys.stdout` and `sys.stderr` objects are backed up and restored for the execution of scripts that are passed to Insight via the command line.

Finally, the `ScriptingInterface` executes some initial Python commands for the import of the elements of the `insight_core` module, and of some elements of the `insight` package into the main module.

At this point, the setup of the Python interpreter is completed. Single Python commands can now be executed by the interpreter using either the `PyRun_SimpleString` or the `PyRun_String` command. The main difference between these commands from Insight's point of view is that the `PyRun_String` command produces the same output as the standard interactive Python interpreter does (i.e., the value of the expression passed to the function), whereas the `PyRun_SimpleString` command itself does not output the resulting value of the executed expression. Therefore, `PyRun_SimpleString` is used for the evaluation of internal commands

[42] In the special case where the value parameter of the attribute setter is a `Property` object instead of an `OutputVar` object, the entity associated with the property is searched for an output variable of the same name. This way, the shortcut syntax for connecting a variable can be used even if the entity of the data provider has a property of the same name as the output variable that should be connected.

(e.g. during the setup of the interpreter's environment), and the `PyRun_String` command is used for the execution of the user input given through the scripting console.

Script files

For the execution of a whole script file, the `PyRun_SimpleFile` function is used. Prior to the execution of a script, the Python list `insight.params` is created and filled with all command line parameters that were passed to Insight after the script name itself. Parameters that are directly recognized by Insight are excluded (see Appendix A.2). If a script file is passed to Insight via command line parameters, the main window is automatically closed after the execution of the script. Insight can explicitly be forced to keep the window open after the execution of the script, either by adding the `-s` ("stay open") command line option, or by setting the Python variable `insight_core.window_stay_open_hint` to `True`. The `ScriptingInterface` checks this variable after the script has been executed.

Manipulators

Manipulators are short scripts that are loaded at startup of Insight. They are associated with one or multiple data entity types and can be invoked from the context menu of all entities of these types. Technically, manipulators are Python functions taking one parameter which represents the entity whose context menu was triggering the manipulator call. Each manipulator is implemented in form of an individual file inside the `insight.manipulator` package. The files are imported by the `ScriptingInterface` using the command "**from** insight.manipulators **import** *". For this to work, a new manipulator needs to be added to the corresponding `__all__`-list in the `__init__.py` file of this package. Each manipulator function is registered for each applicable entity type by a call to the `insight_core.register_manipulator` core function.

Some manipulators require user input to fulfill their task. For example, a manipulator of a "NetCDF data source" entity for visualizing a two-dimensional cross-section through the data needs to know the name of the variable to be visualized. This user input is realized through the `insight_core.UserQuery` class, which is a wrapper of the corresponding C++-class `UserQuery`. The query itself is based upon the creation and display of `DataEntityProperty` objects. After the instantiation of an `insight_core.UserQuery` object, the properties are associated with the query by calls to the `add_property` member function. Internally, the property objects are created at runtime using the C++ `DataEntityPropertyFactory` class. After all property objects have been created and added to the query, a dialog containing a `DataEntityPropertiesWidget` with all corresponding property widgets is created and shown to the user by calling the `query_user_input` function. After the user has closed the dialog, the final values of the properties can be inquired by the `get_property` function.

"Scripted operation" data entity

The "Scripted operation" data entity allows the execution of arbitrary scripts on any number of input data. It takes an arbitrary number of input variables and provides exactly one output variable. The entity has no dimension requests, so it does not iterate over any input

dimensions itself. It doesn't select the dimension's positions either, but adds the dimensions of the input variables to the output variable instead.

The entity possesses two text properties. One property contains the main script to be executed for each query, and the other contains an initialization script that is executed only once and can be used to prepare the computation (e.g. import required modules). Each time the value of the output variable is queried, the user-defined main script is executed using the global namespace of the embedded Python interpreter, and an individual local namespace containing the current values of all input variables as variables with the names `var1`, `var2`, and so on. At the end of the execution of the script, the floating point value that is present in the local variable `result` is returned as the current value of the output variable.

In order to speed up the execution of the script, it is compiled into bytecode represented by a Python code object using the `Py_CompileString` function[43]. This code object is then executed for each individual query via the `PyEval_EvalCode` function.

2.8. Realization of the serialization

Insight offers two different approaches for storing and restoring the program state. The first approach uses XML to represent all required information in a single file. The second approach is to create a series of Python commands and store them in a script file which, when executed, restores Insight to the previous program state. The realization of both approaches is described in the following two sections.

The complete state of Insight that is serialized consists of the following objects:

- The set of data compositions, including their names and composition properties and the currently selected composition.
- For each composition, the set of all data entities, including the type and the name of the entity.
- For each data entity, the current state of each property of the entity.
- The data connections between the entities.
- The restricted ranges of all output dimensions.
- The set of all viewports, including the selected view controller[44] and projection controller, the current transformation and the camera setup.
- For each viewport, the viewport-dependent visibilities of all visual representations.
- The setup of the grid view of the composition.

Currently, the export in form of a Python script is only available for the whole set of all data compositions at once, whereas the export in form of XML files is either available for all compositions or for individual compositions separately. This allows for the creation of new data visualizations by combining existing compositions. If the user wants to restore the whole

[43] cf. `http://docs.python.org/reference/datamodel.html`, last accessed: 05-September-2012

[44] A view controller realizes different methods of user interaction for controlling the parameters of a viewport's camera (e.g. position and orientation).

set of compositions from an XML file, Insight offers the choice between replacing and keeping all current compositions before the new compositions are loaded from the file. The execution of an exported script, in contrast, replaces all existing compositions by default.

The output of the required information for the serialization either in form of XML or as a sequence of Python commands, as well as the import of the object states from an XML file are implemented locally in the respective classes of the involved objects. Both the import and export are hierarchically organized. For example, the `Composition` class is responsible for the invocation of the serialization functions of each `DataEntity` object representing an entity of the composition. This includes, in case of the import of the program state, the creation of the corresponding `DataEntity` object. The same is true for the `CompositionList` and the `DataComposition` classes, as well as the `DataEntity` and the `DataEntityProperty` classes. Table 2.8 provides an overview of all involved classes and their specific tasks during serialization.

2.8.1. Serialization using XML files

The export and import of the program state via XML is implemented straightforward, using the Qt library for the handling of the XML files. All settings are stored as sets of XML-elements and attributes. The serialization starts at the `Vis3d` object which provides the actions and slots of the “File” menu of Insight. The slots which are connected to the serialization actions are responsible for querying the filenames by showing a file selection Qt dialog. By default, all names of XML files storing the program state of Insight end with `.insight.xml`. The serialization itself starts with a call of the corresponding serialization function of the `CompositionList` object. From there, execution is passed on to the single `DataComposition`, `DataEntity`, and `DataEntityProperty` objects. When an entity is to be saved, the entity’s name and type are stored, in order to allow the `DataComposition` class to create the correct entity on loading. Thereafter, the serialization function of all properties that are not dimension selects are called. The reason for treating the dimension selects separately is that on loading, the properties of a data entity are restored in advance of the input connections. So at the time when the properties are restored, the entity has no information about the future input dimensions and their mappings. Because of that, the `DataEntity` stores the information about the dimension selects in separate data structures on loading, such that the correct values can be restored as soon as the input connections are established.

2.8.2. Serialization using Python scripts

The export of the program state in form of a Python script is organized almost in the same way as the XML file export. However, there are some differences which have to be taken into account.

Since the “import” of a program state stored in a Python script is carried out by simply executing the corresponding script, care has to be taken that the data entities are serialized in a correct order. The exported script has to establish and to set up all data connections of an entity before any of its properties can be set. The reason for this is that the Python scripting interface offers no mechanism for the special handling of dimension select properties.

Vis3d	This class provides the serialization-related actions and slots of the “File” menu. It is responsible for showing the file selection dialogs and for passing on the serialization to the CompositionList and Viewport classes, as well as to the DataCompositionGridView.
CompositionList	This class passes on the serialization to the DataComposition objects. On import, it is responsible for the creation of the compositions.
DataComposition	The main tasks of this class during serialization are the invocation of the corresponding functions of the DataEntity objects (which are created on import), as well as the serialization of the composition properties and the data connections between entities.
DataEntity	The serialization functions of all DataEntityProperty classes are invoked by this class. The properties representing dimension selects require some special treatment, since they have to be created before the input is connected during the import of an XML file.
DataEntityProperty	Each property is responsible for storing/restoring their own state.
Viewport	Each viewport stores its own state including the current projection controller and view controller, the coordinates for the global rotated pole correction, the background color, and the visibility of the coordinate axes and the scale bars. In addition, the function for the serialization of the camera setup, which is part of the ViewController object, is called.
ViewController	This class stores and restores the camera setup.
DataCompositionGridView	This class is responsible for storing and restoring the arrangement of entities on the grid view for each composition.

Table 2.8.: The classes involved in the serialization of the program state.

Because of this, all unmapped input dimensions have to be known before the setup of the properties can be carried out.

The sequence of the serialization of the entities can be determined by recursively serializing all data entities providing input data to a data entity before its own state is serialized. Since cycles are not allowed in the graph of data connections between entities, this recursion will terminate at entities without any input data requests.

In order to obtain a more readable script, the `Entity` Python object from the `insight` Python module is used for creating and manipulating data entities. For this, a unique variable name has to be associated with each data entity. These default names are determined

by a static string that is provided by each `DataEntity` C++-class of the respective type. For example, the standard Python name of an "NetCDF data source" is `ncdf_source`. If more than one entities of the same type exist in one composition, their names are extended by numeric suffixes.

2.9. Example use case: Visualization of cyclone simulations

2.9.1. Introduction

Now that all major conceptual and technical aspects have been covered in the previous sections, we are ready to take a closer look at an actual application of Insight. In this section, an example use case based on the work of Sebastian Schemm and colleagues from the ETH Zurich is presented. In this use case, Insight is utilized for the visualization of data from a simulation of the development of idealized cyclones. In the work of Schemm et al. (2012) dry and moist simulations are compared and the influence of warm conveyor belts (WCBs) on the formation and intensification of the cyclones in the moist case is investigated.

Schemm utilized the interactive user interface of Insight for real-time exploration of the simulation results. He created a script for the automatic production of a sequence of images, which he later composed into a video showcasing certain results of his work. The video can be found on the front page of the Insight web presence at `http://insight.zdv.uni-mainz.de/trac`. An example visualization of the work can be found in Fig. 2.7 on page 26. Another visualization, directly taken from the published article, is depicted in Fig. 2.18. This image shows the combination of different visualization techniques (contour lines, isosurfaces, and colored cylinder curves) for obtaining insight into different types of simulation data and their spatial relations.

Outline

This section is divided into two parts: First, we take a closer look at the underlying data for the visualization. We discuss the ways in which the data is created and look at its properties, such as the variables taken into account, as well as the spatial and temporal resolution of the underlying grid. Next, the technical realization of the data processing and visualization is described, leading to a discussion of the script for the production of the screenshots that provide the basis for the composition of the video.

2.9.2. Input data

The data for the visualization by Insight is created in two consecutive steps. The first step is the simulation of the idealized cyclones using a weather prediction model. The model output is given in form of netCDF files containing several different data variables located on a regular grid. The second step consists of trajectory calculations using the wind speed variables that are part of the model output from step one. The trajectories of interest are selected and several variables are traced along these selected trajectories. Again, the data from step one is used as basis for the tracing.

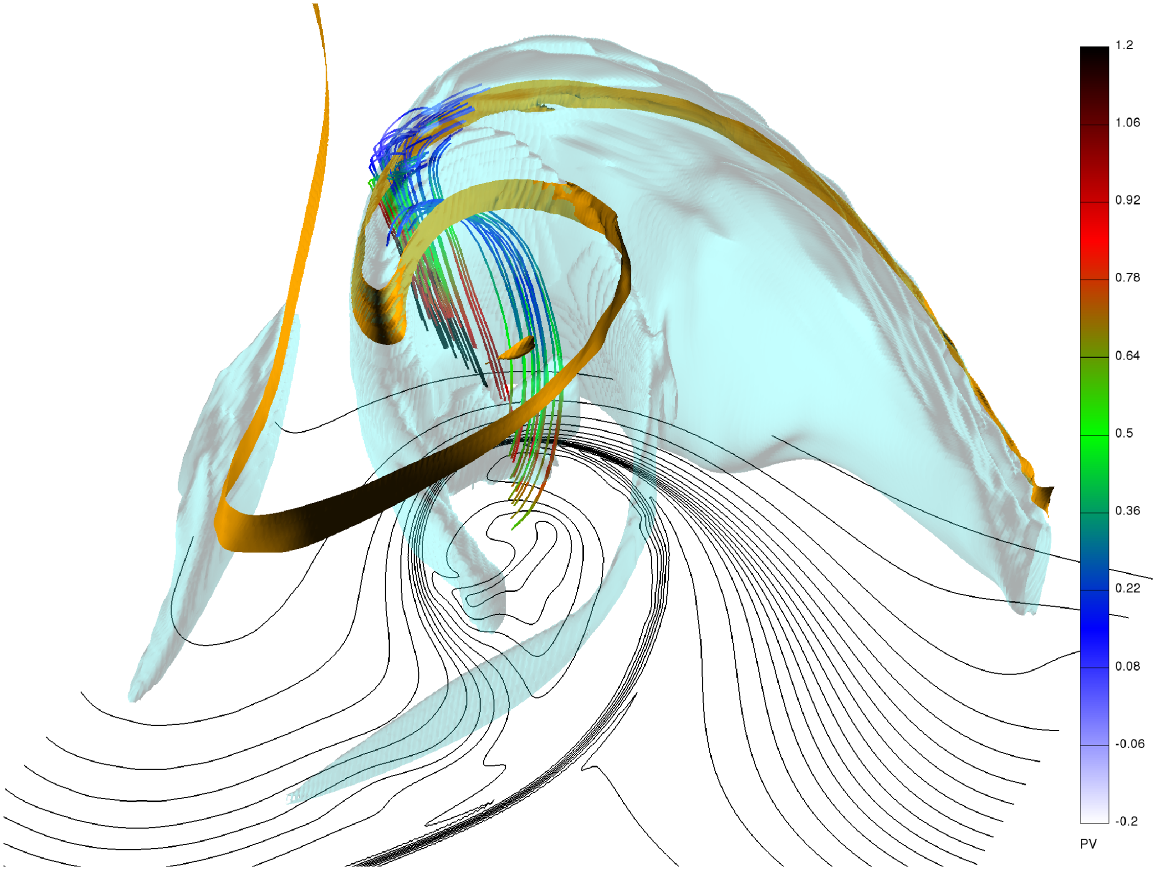


Figure 2.18.: Combination of several visualization techniques for the analysis of the cyclone simulation data. The black contour lines depict surface isentropes from 266 K to 298 K with an interval of 2 K. The blue isosurface shows cloud water content at 0.2 g/kg, the brown isosurface shows PV at 2 pvu. This isosurface is truncated to heights between 6.0 and 6.5 km. The WCB-trajectories show PV in pvu, the colors correspond to the scale on the right. Image taken from Schemm et al. (2012).

COSMO model data

The fundamental data of this study comes from a modified version of the COSMO (**CO**nsortium for **S**mall-scale **MO**delling)[45] weather prediction model (formerly known as "Lokal Modell" (LM), see Steppeler et al., 2003). The simulation is initialized by setting the required variables to well-defined start values given by a set of adaptable, multivariate functions. The horizontal resolution is 800x400 grid points, covering an area of 16800x8400 km. The data set has 60 vertical levels equally distributed over 21 km. The coordinate system is rotated such that the original north pole of the grid is now positioned at 50°N/140°E. The temporal resolution of the output is one hour. The output consists of several variables, including the three wind components, temperature, pressure, and more. The data is divided into separate files for each time step. Additionally, primary and secondary (derived) variables are stored in separate netCDF files, identified by the filename prefixes "P" and "S", respectively.

[45] `http://www.cosmo-model.org/`, last accessed: 05-September-2012

Trajectories

The study also contains trajectory data for the identification and analysis of warm conveyor belts. The trajectory calculations are based on the model output of the moist simulations and are carried out by the Lagranto software tool (Wernli and Davies, 1997). The starting points for the calculations are spread horizontally over the whole model domain and are placed vertically in the low troposphere. From the set of trajectories resulting from this calculation, only trajectories along the WCB are of interest. They are selected based on a fixed threshold on their maximum two-days ascent. The final step is the tracing of variables along the selected trajectories, including PV, potential temperature and specific humidity. These traced values are obtained by linear interpolation of the corresponding variables from the model output.

2.9.3. Realization

Visualization techniques

Three different visualization techniques are used for the creation of the 3D plots and the video from the model output data: isosurfaces, isentropic cross-sections (realized via colored isosurfaces), and iso- or contour lines. The trajectory data is visualized using colored OpenGL line strips and colored curves consisting of connected, piecewise linear cylinder sections. Schemm uses two different coloring schemes for visualizing the trajectories: a traditional mapping of the values of a traced variable to a set of colors, and an alternating black/white color pattern for the visualization of the temporal progression of the air parcels where each uniformly colored trajectory interval corresponds to an equally lasting time period of air parcel movement.

One aspect of the input data that requires additional consideration is the handling of the underlying data grid of the COSMO output data. The desired projection type for the visualizations is the standard equidistant cylindrical projection. The rotated data, however, covers an area that is not aligned with the regular longitude/latitude grid and which contains the north pole and hence appears distorted under this type of projection. By suspending the rotation of the grid we obtain data covering an aligned rectangular area. The drawback is that the positions where data from other sources appears, for example the trajectory data, is no longer consistent with the COSMO data. For solving this, Insight offers an option to globally rotate all data. This option takes the new position of a rotated pole as arguments and performs a rotation that reverses the grid rotation. Since this global rotation is applied to all visualized data, the relative positions of data from different sources remains consistent.

The remaining setup of the visualization pipeline and the interactive exploration of the data is straightforward. The data is read in through “NetCDF time series source” and “Trajectories data source” entities. Using the GUI, the context menus of these entities provide access to macro scripts for the setup of the isosurfaces, the contour lines and the trajectories. The data given on the rotated grid is automatically associated with the correct position, such that it matches the positions of the non-rotated trajectories.

Script for the creation of the video

Insight was used to produce the single frames of the video showcasing some results of the study (the video can be found at `http://insight.zdv.uni-mainz.de/trac`). Subsequently, the most important parts of this script are presented. The full script can be found in Appendix E.

At the beginning, the script performs some file system operations in order to obtain all filenames of the primary and secondary input netCDF files, as well as the associated dates. The filenames of the primary and secondary netCDF files are stored in the `input_p_files` and `input_s_files` lists, respectively. The list of dates is stored in the `dates` variable. The dates are extracted by parsing the filenames using the `strptime` function of the `datetime` class from Python's standard `datetime` module. Finally, the input filename of the trajectories is given by the `lsl_file` variable.

With these environment dependent variables set, the script is ready for setting up the necessary data sources.

```
ncdf_p = Entity("NetCDF time series source")
ncdf_p.netcdf_file = ';'.join(input_p_files)

ncdf_s = Entity("NetCDF time series source")
ncdf_s.netcdf_file = ';'.join(input_s_files)

lsl = Entity("Trajectories data source")
lsl.trajectories_file = lsl_file
```

The video contains the following visualization elements:

- a colored isosurface representing potential vorticity on the 317 K isentrope,
- a legend relating the colors of this isosurface to PV values given in pvu,
- contour lines depicting air pressure at the surface,
- the potential temperature (θ) at the surface,
- a red isosurface representing the area of latent heating,
- a cyan isosurface representing saturated air,
- contour lines representing zonal wind speeds at 11 km height for identification of the jet stream,
- colored cylinder curves representing the warm conveyor belt, the black and white intervals indicate the time steps of one hour,
- and a text-caption that shows the current day and hour of the simulation.

As an example, we show the setup of the PV isosurface here:

```
# -- define the PV colormap
cmpv                = Entity("Color mapper")
cmpv.legend_pos     = "rightmost"
cmpv.caption        = "[PVU]"
```

```
cmpv.mapping          = pv_color.transformed(0, 4)
cmpv.text_size        = 20
cmpv.text_color       = (0.0,0.0,0.0)

# -- create the TH isosurface
iso                   = Entity("[wrapper] Iso surface")
iso.value             = ncdf_s.th
iso.iso_value         = 317
iso.use_fixed_color   = False
iso.transparency      = 0.4

# -- color the TH isosurface with PV
cmpv.datain           = ncdf_s.pv
iso.color_optional    = cmpv.color
```

These basic setups do not change during the animation sequence. The main loop of the animation iterates over all input files and updates the selected time steps of the visual representation entities, the length of the trajectory visualization (controlled through the range of the sample-dimension of the trajectories data source), the caption, and the camera setup.

The update of the time dimension selection is straight-forward:

```
for timestep in range(0, len(input_p_files)):
    # -- Set the time step of the visual representations...
    iso.time = timestep
    isolh.time = timestep
    isosat.time = timestep

    th.time = timestep
    ps.time = timestep
```

Since the starting date of the trajectories may differ from the starting date of the simulation output data, the update of the trajectory visualization requires some additional calculations for finding the correct time step.

```
    # loop step for trajectories
    date     = dates[timestep]
    # find the trajectory time step matching the current date
    traj_diff = date - startdate
    t = traj_diff.days * 24 + traj_diff.seconds / 3600
    lsl.sample.range = (0,int(t))
```

Note, that `statedate` contains the first date of the trajectory calculations. The invalid range that would result from a negative value of t simply leads to a dimension size of zero, such that nothing would be visualized.

It remains to update the camera position and the caption, and to finally take the screenshot.

```
    # zoom the camera into the plot
```

```
    set_zoom_factor(4.2 + timestep * 0.065)

    # set the camera angle
    set_camera((20 + timestep * 0.5, 18 - timestep * 0.02, 0))

    # set a camera translation
    set_translation((-0.22, 0.1 - timestep * 0.001, 0.09))

    # update the "date" caption
    cap.caption = "Day " + str(date.day) + " - Hour " + 
        str(date.hour).zfill(2)

    # finally, we take the screenshot
    take_screenshot(plot_prefix + str(int(date.day)-10) + 
        str(date.hour).zfill(2) + '.png', 2560, 1600)
```

2.10. Outlook

Insight, in its current state, proved to be a versatile software tool for several "real-world" visualization and data processing tasks. It has been utilized for the production of plots and images which now can be found in several published theses and papers (for examples, see Sect. 2.1.3). In addition to its application as a visualization tool, it also forms the fundament for the feature extraction and tracking techniques presented in the upcoming chapter.

The architecture of the implementation was chosen carefully bearing in mind its maintainability and extensibility. However, Insight is far from being finished. At the moment of this writing, it is neither completely bug free nor does it contain all the features that were discussed up to now or on the wish lists of some of Insight's users.

A few points that are still missing or potentially could be improved are:

- Several new visualization techniques could be implemented, for example new volume rendering techniques, cutting planes that are not aligned to the grid axes, proper rendering of vector-valued data sets and vertical cuts along trajectories.
- The Visualization Toolkit (VTK[46]), see Schroeder et al. (2006), could be integrated into Insight. VTK is a C++ class library providing, among other things, a wide range of three-dimensional visualization techniques. The usage of VTK for the implementation of Insight's visual representations was discussed several times, right from the very beginning of Insight's development. We decided to stick to the basic OpenGL visualization that was already implemented in Vis3d (Insight's predecessor) as long as the effort of directly using OpenGL for the implementation of additional visualization techniques was lower than a conversion of the whole project to VTK visualizations. For the visualization techniques discussed above, we need to reconsider whether this decision would still hold.

[46] http://www.vtk.org/, last accessed: 11-September-2012

- Currently, data dimensions in Insight are essentially represented by two integer values: their size and their current position. Some sets of dimension indices, however, could directly be associated with meaningful data, for example with a pressure value indicating a height level or a date representing a certain point in time. Data grids are currently utilized to obtain information about the geographic coordinates associated with any combination of grid dimension indices, but there is no general way to associate dimensions with a meaningful interpretation of their current position. Such an association would be useful for example for the selection of dimension positions. It would be much more intuitive for the user to select a real date instead of a time index, or to select a height by input of a pressure value instead of a model level.

- An extension of the previous point would be the implementation of data dimensions which allow real numbers as position values, together with a corresponding interpolation of the associated values of the data variables. For example, the user could access data at height levels between the available model levels, without the effort of setting up an explicit data interpolation.

- The mapping of dimensions to meaningful data only makes sense if the data itself contains information about its own format and unit. At the moment, a data variable in Insight is conceptionally either an integer-valued or a real-valued function, without any information about the interpretation of the provided integer or floating point values. In a future version of Insight, it should be possible to associate a data variable with metadata that allows an interpretation of the stored values (for example as a height value or a date).

- Currently, Insight is only capable of exporting data in form of very basic netCDF files (format version 1). In the future, further file formats could be supported. As soon as data variables contain additional meta information, we could export netCDF files that fulfil the full CF-conventions (Gregory, 2003), or which at least contain the corresponding variable attributes for fulfilling the Unidata udunits conventions[47]. Currently, Insight export CDF1 files with 32-bit offsets, which imposes several constrains on the allowed size of single variables and on the total file size. Switching to CDF2 files with 64-bit offsets would increase the limit on the total file size. Possible alternatives to the CDF1 and CDF2 file formats in order to avoid the limit on the size of variables would be the export of netCDF-4 files or a direct export of HDF[48] (**H**ierarchical **D**ata **F**ormat, see Folk et al., 2011) files.

- The export of Python scripts for the serialization of the program state should be further refined, for example by accessing entity features through the shortcut names realized by dynamic object attributes. For this, additional tests for ensuring a correct mapping of the shortcut names to the corresponding objects have to be implemented. As it was stated in Section 2.7.2, the attribute names of entities are matched in a fixed sequence to the converted names of properties, output variables, output dimensions, variable requests and dimension requests. If there are any collisions of names, a direct access may not be possible. In the long term, the script file export may replace the serialization via XML files completely.

- Although Insight is focused on the interactive creation of data visualizations, it is

[47]cf. `http://www.unidata.ucar.edu/packages/udunits/`, last accessed: 12-September-2012
[48]cf. `http://www.hdfgroup.org/`, last accessed: 12-September-2012

already utilized for data processing tasks which do not require Insight's GUI. For such tasks it would be advantageous to have a stand-alone version of Insight without a GUI, which is solely controlled through the Python scripting interface. For the production of plots, Insight needs to perform offscreen OpenGL rendering, which is not trivial to implement. Further, the view components of the MVC architecture of the implementation need to be revised.

Information on the current state of the development of Insight, as well as an overview of all upcoming and completed tasks, and of the progress of the next major milestones, can be found on the project homepage of Insight at `http://insight.zdv.uni-mainz.de/trac`.

3. A Novel Algorithm for the Detection, Tracking, and Event Localization of Interesting Features in 4D Atmospheric Data

This chapter covers the theoretical concept, the outline of the implementation, and several examples of application of a novel algorithm for the detection, tracking, and event localization of interesting features in 4D atmospheric data. The major part of this chapter consists of an adapted version of a GMD[1]-article (Limbach et al., 2012) by Heini Wernli (ETH Zurich), Elmar Schömer (Johannes Gutenberg University of Mainz), and by myself. The parts of this chapter ranging from Sect. 3.1 up to Sect. 3.4.3 are directly taken-over from the article and are only slightly adapted. Most parts of the article were written by myself and revised by Heini Wernli and Elmar Schömer. An exception is the case study in Sect. 3.4.1, which was primarily written by Heini Wernli.

The original article was published in 2012, since then the segmentation algorithm was further extended and applied for the detection of additional atmospheric features. This chapter contains two newly added sections covering these topics. Sect. 3.5 provides an overview of the most important extensions. In Sect. 3.6, the details of an additional example of practical application of the algorithm for the segmentation of three-dimensional cyclones is presented, together with some first results.

The abstract of the article (following directly below), as well as the introduction in Sect. 3.1, are adapted to cover the extended content of this chapter. The final two sections, a short outlook on future improvements of the algorithm in Sect. 3.7, and a final conclusion summing up the achieved goals in Sect. 3.8, are based upon the final section of the original article.

In this chapter, we introduce a novel algorithm for the efficient detection and tracking of features in spatiotemporal atmospheric data, as well as for the precise localization of the occurring genesis, lysis, merging and splitting events. The algorithm works on data given on a four-dimensional structured grid. Feature selection and clustering are based on adjustable local and global criteria, feature tracking is predominantly based on spatial overlaps of the feature's full volumes. The resulting 3D features and the identified correspondences between features of consecutive time steps are represented as the nodes and edges of a directed acyclic graph, the event graph. Merging and splitting events appear in the event graph as nodes with multiple incoming or outgoing edges, respectively. The precise localization of the splitting events is based on a search for all grid points inside the initial 3D feature that have a similar distance to two successive 3D features of the next time step. The merging event is localized analogously, operating backward in time. As a first application of our method we present

[1] Geoscientific Model Development, see `http://www.geosci-model-dev.net`, last accessed: 08-April-2013

a climatology of upper-tropospheric jet streams and their events, based on four-dimensional wind speed data from European Centre for Medium-Range Weather Forecasts (ECMWF) analyses. We compare our results with a climatology from a previous study, investigate the statistical distribution of the merging and splitting events, and illustrate the meteorological significance of the jet splitting events with a case study.

Furthermore, we present three extensions to the original algorithm: The support of double thresholding, the dilation of 3D features as a preprocessing step for the overlap-based tracking, as well as extended strategies for the analysis of the event graph. Finally, we present an additional example application that utilizes some of these new features for the detection and tracking of three-dimensional cyclones. An example case study, as well as first results of a five-years climatology of cyclones are presented. Finally, a brief outlook is given on additional potential applications and improvements of the 4D data segmentation technique. The chapter closes with a summarization of the achieved goals.

The algorithm presented below is completely implemented as part of the Insight software project (see Chapter 2). Most of the images to be found in this chapter are created with Insight.

3.1. Introduction

In this introductory section we will first explain the need for and usefulness of developing efficient automated algorithms for identifying and tracking specific features of the highly variable atmospheric flow. In a second subsection, a brief review is provided of previously developed feature detection and tracking algorithms, which will serve as a basis for motivating the novel approach developed in this study.

3.1.1. Identification and tracking of atmospheric flow features

Albeit highly variable, the atmospheric flow can be described by characteristic and frequently recurring flow features, like for instance tropopause-level jet streams, and surface cyclones and anticyclones. These flow features are particularly important since they are typically associated with certain weather conditions (e.g., sunny and dry weather with subtropical anticyclones; stormy weather and intense precipitation with extratropical cyclones) and with specific dynamical processes (e.g., cyclogenesis on the poleward side of intense upper-level jet exit regions).

Therefore, several algorithms have been developed during the last decades to objectively and efficiently identify atmospheric flow features from large data sets. These algorithms make it possible, for instance, to produce synoptic climatologies of specific features and, if applied to homogeneous climate data sets, to quantify potential trends in the frequency of these features.

Early examples of such algorithms are the cyclone identification techniques by Lambert (1988), Murray and Simmonds (1991), and König et al. (1993). The earliest of these approaches considers cyclones as point objects and provides climatological density maps of these features. They are typically identified as local extrema of a particular field (e.g., minimum sea level pressure). The later techniques also considered the temporal coherency of the

features, which led to the computation of feature tracks, and feature genesis and lysis points. For a concise review on cyclone identification and tracking methods, the reader is referred to Ulbrich et al. (2009). The algorithm introduced by Wernli and Schwierz (2006) considers cyclones explicitly as finite-size two-dimensional features (instead of point objects). Similar approaches have been used to identify, for instance, upper-tropospheric jet streams (Koch et al., 2006) and upper-tropospheric cut-off cyclones (Wernli and Sprenger, 2007) as two-dimensional features. The objective identification of these features was based upon either the topology of the two-dimensional field (e.g., considering the outermost closed contour surrounding a local extremum) or a simple threshold (e.g., considering the region where a field exceeds a certain value). So far, all these two-dimensional feature identification algorithms have been applied to individual time steps of climatological atmospheric data (e.g., every 6 h if using recent reanalysis data sets) and additional tracking algorithms have been used to meaningfully connect the identified features in time. As a consequence, these feature identification and tracking algorithms treat the spatial and temporal dimensions of atmospheric data very differently.

In this current study we will introduce a novel approach to the identification and tracking of atmospheric flow features as full 3D objects developing over time. In addition, our method estimates the location of the detected merging and splitting events in grid point space. The output of our segmentation method allows performing specific analyses of the interaction and development of the observed atmospheric features. For instance, a precise event localization is useful for the computation of climatologies of events and for a statistical analysis of the lifetime and stability of an atmospheric feature. In addition, the location of feature events (e.g., the merging of two jet streams or the splitting of an extratropical cyclone) can objectively pinpoint important atmospheric processes. In Sect. 3.4.1, we will present a case study to illustrate this point.

3.1.2. Conceptional view on feature identification and tracking

Feature extraction and tracking are common tasks in different scientific areas, for example in image processing and computer vision (Zucker, 1976; Nikhil and Sankar, 1993; Jain et al., 1995; Davies, 2005), as well as in flow visualization (see Post et al., 2003, for an overview). An important goal of feature extraction and tracking is the creation of reduced data sets and derived attributes characterizing the parts of interest of the original, often much larger input data set. Such a reduced representation allows for a statistical evaluation of the features and for an efficient visualization.

Several methods for the *extraction* of flow features exist. The choice of a method depends considerably on the characteristics of the input data sets as well as on the features of interest. In Post et al. (2003), feature extraction approaches are classified into three groups: based on image processing, on topological analysis, and on physical characteristics. In our current application, we are not aiming for a topological analysis but focus on the physical characteristics of the underlying data set, and on the adaption of image processing methods for our purpose.

Van Walsum (1995) proposed a feature extraction method called "selective visualization". This method uses a boolean *selection criterion* for deciding whether a single grid point belongs to a feature or not, and a *connectivity criterion* in order to cluster neighboring selected

points. Other approaches, for example by Reinders (2001), use this selective visualization method for feature extraction. Silver and Zabusky (1993) propose to apply region growing techniques for detecting features. Region growing is a well-known image segmentation technique, see, e.g., Zucker (1976). The basic idea is to start the segmentation at an initial grid point, commonly representing an extremal value of the data set, and then to iteratively add new neighboring grid points, as long as they can still be associated with the phenomenon to be detected. The usage of a spatial data structure, for example an octree (Silver and Zabusky, 1993; Wilhelms and van Gelder, 1992), can be effective for speeding up the data processing. Siegesmund (2006) applied a region growing method for the segmentation of ozone holes from time-series of two-dimensional ozone data. Muelder and Ma (2009) let an approximation of the boundary grid points of a feature successively shrink and grow, until they obtain a new boundary representation of the feature at the next time step. Their initial approximation is based upon an extrapolation of the results of previous time steps. For the detection of new features that may have formed, they perform a search over all unassigned grid points.

Post et al. (2003) proposed three different basic approaches towards feature *tracking*. The first approach is to extend a three-dimensional feature detection method to the full spatiotemporal domain, as it is done for example by Weigle and Banks (1998), Bauer and Peikert (2002), and Ji et al. (2003). The second approach is to test features for region correspondence on a cell-to-cell basis, as proposed by Silver and Zabusky (1993), and Silver and Wang (1997, 1999). The third type of feature tracking covers methods which compare attributes of the features of consecutive time steps (for example total mass, center of mass, or moments). Samtaney et al. (1994) performed feature tracking by testing whether the differences of the observed attributes of the features stay within preassigned tolerances. Reinders (2001) used correspondence functions, which estimate the similarity of the attributes of different features for feature tracking.

Taking into account the huge amount of atmospheric data we want to process (e.g., a multi-decadal period of atmospheric reanalysis data), we aimed for a method that applies feature extraction, clustering, attribute calculation and tracking in one sequential pass over the data set. Our method for feature extraction uses ideas from many of the approaches mentioned above. Single grid points are selected based on a *local selection criterion*. Clustering is performed based on a *local homogeneity criterion*. We decide to keep or discard single segments afterwards based on a *global selection criterion*. This compensates, to a certain extent, for the selection of a seed-point as in traditional region growing techniques. During the iteration over the data, we represent intermediate features using a union-find data structure (see, e.g., Knuth, 1997, and Cormen et al., 2009) to guarantee efficient operations on the features (for example merging multiple features as soon as a connection is detected). Since we aim for the precise localization of the features and their events, we keep track of all grid points of every feature at all steps of the algorithm.

We realize feature tracking by testing for spatial overlap on a cell-to-cell basis. This was an obvious decision since it corresponds to a plain extension of our algorithm from three to four dimensions. We initially designed our algorithm for the detection and tracking of atmospheric features such as jet streams, which are relatively large and slow-moving with respect to the temporal resolution of our data sets. However, even in case of such features we have to deal with continuations of small, fast-moving features that can not be identified by spatial overlaps. To cover these cases as well, we perform additional tests for continuation

based on comparisons of the center of mass and volume attributes of the features.

Our algorithm represents the temporal relations between features detected during the feature tracking in form of an event graph. An event graph is a directed acyclic graph as proposed by Samtaney et al. (1994). They identified four different events: *creation*, *dissipation*, *bifurcation*, and *amalgamation*. In order to stay in line with the terminology used in atmospheric sciences, we call these events *genesis*, *lysis*, *splitting*, and *merging*. Our algorithm puts no additional constraints on the detection of events and does not detect unresolved events, in contrast to, e.g., the method by Reinders (2001).

We implemented the algorithm as part of a novel software tool, Insight (cf. Chapter 2), for the analysis and segmentation of atmospheric data (Limbach et al., 2009). As one of our first applications, we computed a climatology of upper-tropospheric jet streams and their events, as documented in this study. The data basis of the jet stream segmentation were operational meteorological analyses from the European Center for Medium-Range Weather Forecasts (ECMWF) for the years 2007 and 2008.

In the upcoming section, we provide definitions of some fundamental terms and structures required for a formal description of our algorithm. In Sect. 3.3, the different steps of the novel segmentation algorithm are described in detail. While these two sections cover the general ideas and mechanisms of our segmentation algorithm, Sect. 3.4 deals with the setup and the results of a concrete application of the algorithm for the identification of jet streams. The section starts with an example case study of a Rossby wave breaking and an associated jet stream merging event. Then, results are presented from the computation and analysis of a two-years climatology of jet streams and their merging and splitting events.

In Sect. 3.5, we present enhancements and new features of the algorithm that were added to support the detection and analysis of additional atmospheric phenomena. The double thresholding presented in Sect. 3.5.1 was implemented in the context of first experiments with the segmentation of ice-water clouds, which is a project by Patrick Neis (Johannes Gutenberg University of Mainz). As part of a joint work with Lukas Papritz (ETH Zurich), we adapted the algorithm for the detection and tracking of three-dimensional cyclones. For this, we refined the techniques for analysing the event graph of the 4D segments. These refinements are discussed in Sect. 3.5.2. Christine Aebi (University of Bern) applied our algorithm in context of her Master's thesis (Aebi, 2012) for the tracking of PV-streamers and warm conveyor belts. In her work, she experimented with the dilation of 3D features, a technique presented in Sect. 3.5.3. In Sect. 3.6, the setup and some first results of the cyclone segmentation are presented. The last two sections provide a short outlook on future developments and applications of the novel algorithm, as well as a summary of the achieved goals.

3.2. Foundations of the algorithm

3.2.1. Input

Segmentation algorithms, such as the one presented here, usually require a discretized, sampled signal as input data. In our applications, the input data consists of a series of discretized 3D data sets representing the state of one or more atmospheric parameters at fixed time in-

stants. The resolution of the three spatial dimensions (longitude, latitude and pressure) and of the time dimension may vary with respect to the concrete application and the available data sources. The underlying continuous domain of our data, however, remains the same:

Definition 1 (Data domain)**.** *The atmospheric data we are interested in is defined on the domain* $\Omega := [-180, 180) \times [-90, 90] \times \mathbb{R} \times \mathbb{R} \subset \mathbb{R}^4$*. The first two components of the domain represent the geographic longitude and latitude in degrees, respectively. The third component represents the vertical dimension, either Cartesian height (in m) or more commonly pressure (in hPa), and the last component represents time.*

In their idealized, continuous form, the atmospheric parameters we are interested in can be seen as the mapping $p : \Omega \to V$. The co-domain V of this mapping depends on the actual features we want to track. Most of the time we are interested in n real-valued atmospheric variables, so our co-domain has the form $V \subseteq \mathbb{R}^n$. If we are, for example, interested in the segmentation of jet streams, a value $x \in V$ could represent the horizontal wind speed as a single scalar, or it could represent a horizontal wind vector $(u, v) \in \mathbb{R}^2$. Some more complex atmospheric structures may require the combination of several other, distinct measures.

In our practical application, the input data is a sampled set of discretized values of the continuous atmospheric data lying on a point lattice within the data domain Ω. The exact form of a single sample depends on the concrete application and the objects we want to identify and track.

Definition 2 (Input data)**.** *Our input data consists of the set of samples* $X := \{x_{i,j,k,t} \mid i = 1, \ldots, i_{\max};\ j = 1, \ldots, j_{\max};\ k = 1, \ldots, k_{\max};\ t = 1, \ldots, t_{\max}\}$*. Indices* i, j, k *specify the spatial position of the sample on the point lattice, index* t *indicates the time step. The set* X_t *denotes all samples of a single time step* t*.*

Although we impose no constraints on the actual form of the point lattice, for the sake of reasonable results the indices of the samples should reflect the sample's adjacencies, that is, neighboring samples should only differ by ± 1 in one index. So far we have worked with data on regular longitude/latitude grids (represented by indices i and j, respectively), with varying pressure given as a hybrid combination of the layer of the data set (index k) and the surface pressure (depending on i, j and t). In meteorological terms this corresponds to hybrid $\sigma - p$ coordinates. When using such a non-isotropic lattice, one has to take care as soon as additional attributes are derived from the results of the segmentation, such as the size of a segment or its center of mass. Approximations of volume integrals, as proposed by van Walsum (1995), can be used for the calculation of attributes on curvilinear grids. The handling of the indices at the poles and at the $-180°/180°$ longitudinal transition (i.e., at the date line) requires particular attention as well.

3.2.2. Output

The goal of a segmentation algorithm is to partition the set of input data into connected subsets of samples, where each subset ideally corresponds to the exact location of the phenomenon one wants to identify (cf. Zucker, 1976; Jain et al., 1995). In our case, since our set of input samples X is four-dimensional, the resulting *segments* will be four-dimensional

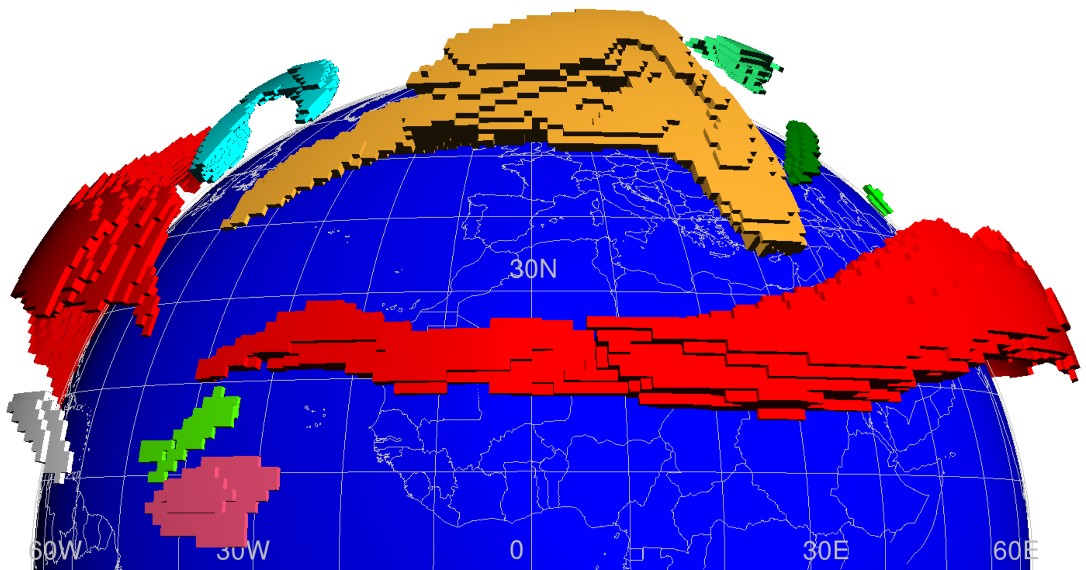


Figure 3.1.: Example illustration showing all detected three-dimensional features at a single time step of a segmentation of jet streams in the Northern Hemisphere (perspective projection, vertically exaggerated). Different colors indicate different features.

objects as well. The construction of these 4D segments is accomplished in two steps:

1. We iterate over all time steps of the input data and partition the three-dimensional set of samples X_t into 3D subsets of samples corresponding ideally to exactly one instance of the atmospheric phenomenon we want to track at the given time step. We call these subsets three-dimensional *features* (see Fig. 3.1). This step is called the *feature detection* step.

2. We *track* and group the features of different time steps, such that we obtain information about the development of the atmospheric phenomena over time. This step is called *feature tracking*, and the resulting sets of connected three-dimensional features are our final four-dimensional *segments*.

More formally, we define the three-dimensional features as follows:

Definition 3 (Features)**.** *The pairwise disjoint sets of connected samples representing one occurrence of the detected atmospheric phenomenon at a single time step are called* features*. We denote the ith feature of time step t as $F_{t,i} \subseteq X_t$. F_t is the set of all features at the time step t. F is the set of all features at all time steps.*

Our algorithm outputs the information obtained during feature tracking in form of an *event graph*, a directed acyclic graph (cf. Samtaney et al., 1994; Reinders, 2001). The set of nodes of the event graph corresponds to the set of all detected 3D features. If a connection between two 3D features of two consecutive time steps is detected within the feature tracking step, this connection is represented as an edge in our graph.

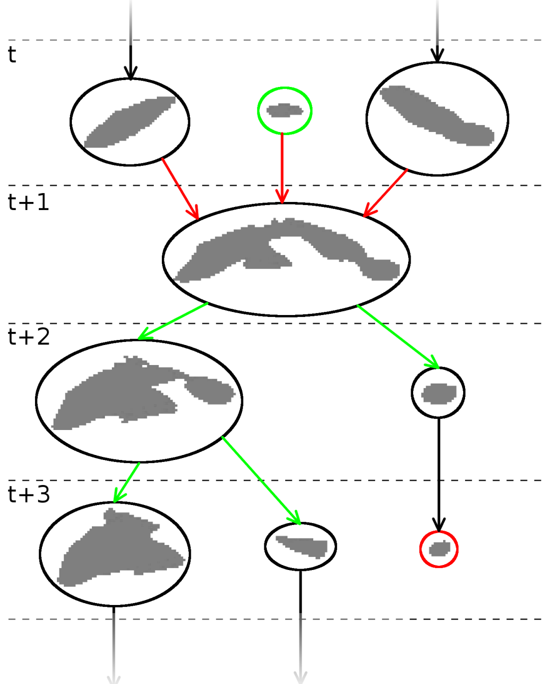


Figure 3.2.: Part of an example event graph of a single 4D segment. Nodes are depicted by ellipses including shapes of the corresponding 3D features. Connecting edges are depicted by arrows, dashed lines indicate the border between features of different time steps. The green ellipse indicates a genesis event, and the red ellipse a lysis event.

The formal definition of the event graph is as follows:

Definition 4 (Event graph)**.** *The graph $G := (F, E)$, where F is the set of all detected features and $E \subseteq \{(a, b) \mid a \in F_t;\ b \in F_{t+1};\ t = 1, \ldots, t_{\max} - 1\}$ is the set of all edges representing a direct connection between features of two consecutive time steps, is called* event graph*.*

Note that there are several ways to define what "direct connection between features of two consecutive time steps" means. We discussed several different feature tracking approaches in Sect. 3.1.2, and discuss the details of the methods used by our algorithm in Sect. 3.3.2.

Our final four-dimensional segments, each representing one atmospheric phenomenon and its development over time, are already contained in the event graph G as the connected sets of 3D features (see schematic example in Fig. 3.2). Such distinct sets of connected nodes of a graph, together with the connecting edges, are called *connected components*.

Definition 5 (Segment)**.** *Let $G_{\mathrm{S}} = (S, E_{\mathrm{S}})$ denote a connected component of G. Then S is called a* segment*, representing all features associated with* one *atmospheric phenomenon as it develops over time and G_{S} is called the* event graph *of S containing all edges E_{S} between connected features of S.*

From the connectivity information provided by each event graph G_S, we can derive information about the occurring genesis, lysis, merging, and splitting events of each segment. The events can be detected through an inspection of the connecting edges E_S of G_S in the following way:

- A *genesis event* is detected if a node in the event graph has no incoming edges. For example, there is a genesis event at the node depicted by the green ellipse in Fig. 3.2.
- A *lysis event* is registered at nodes without outgoing edges. See the red ellipse in Fig. 3.2 for an example.
- If a node has more than one incoming edge, we register a *merging event*. This is the case at the node with the red incoming edges in Fig. 3.2.
- A *splitting event* exists at nodes with multiple outgoing edges. All nodes with green outgoing edges in Fig. 3.2 are associated with a splitting event.

Since features at the first time step have no incoming edges, we cannot tell genesis events, defined the way described above, apart from situations where a phenomenon simply enters the sampled data domain. In order to avoid spurious results, we exclude the first time step from the detection of our genesis events. We have an analogous situation regarding lysis events at the last time step and we therefore exclude them as well. Note that for some applications it can be reasonable to replace genesis events at the first time step and lysis events at the last time step by *entry* and *exit* events, respectively. Reinders (2001) detected entry and exit events as well, but with respect to the spatial, not the temporal boundaries of the observed system.

Our algorithm is capable of estimating the locations of the occurring events not only on a per-feature but on a per-sample basis. In case of genesis and lysis events, all samples of each single involved feature are associated with the respective event. The detection of the locations of merging and splitting events is more involved, as described in detail in Sect. 3.3.3. In all cases, we denote the result of these attributions as follows.

Definition 6 (Event locations)**.** *The localization of the occurring events is represented by the set* $T \subset X \times \{\text{“genesis”}, \text{“lysis”}, \text{“merging”}, \text{“splitting”}\}$. *This set contains all involved samples together with an annotation indicating the event types that occur at the respective positions of the samples on the point lattice.*

3.2.3. Feature detection predicates

The way in which our algorithm selects and clusters the samples of the input data set to create the relevant three-dimensional features depends on the formulation of three different predicates. We incorporated ideas from region growing and other segmentation methods that use different types of binary predicates for feature detection. The “selective visualization” method proposed by van Walsum (1995) uses a *selection criterion* and a *connectivity criterion* in order to select and cluster samples from the input data set. In other methods, a *global homogeneity criterion* is used for the identification of connected regions of interest, see Zucker (1976), Jain et al. (1995), and Nikhil and Sankar (1993).

Our method selects single samples of our input data set using a *local selection criterion* (l). Samples are clustered by means of a *local homogeneity criterion* (h). A *global selection*

criterion (g) is used to select the final segments at the end of the segmentation process. The main task to be fulfilled before the algorithm can be applied to different types of atmospheric phenomena is to find adequate and applicable predicates h, l and g.

Definition 7 (Local selection criterion)**.** *The* local selection criterion $l : X \to \{\text{TRUE}, \text{FALSE}\}$ *decides whether or not a single sample belongs to a potential three-dimensional feature, based on any of its local characteristics.*

This criterion can be applied to discard unsuitable samples right from the start. Depending on the objects to be detected, the formulation of a local criterion may be sufficient for a full classification of the features we want to identify. A simple but nevertheless powerful formulation of such a selection criterion is to test whether the values of the samples lie above or below a given fixed threshold. In our case study on jet streams, for example, we used a height-dependent threshold on the wind speed.

For the clustering of the selected samples, we use a binary predicate called the *local homogeneity criterion*. We define it as follows:

Definition 8 (Local homogeneity criterion)**.** *The* local homogeneity criterion $h : X \times X \to \{\text{TRUE}, \text{FALSE}\}$ *decides whether a given pair of neighboring samples* $x := x_{i,j,k,t} \in X$ *and* $x' := x_{i+i',j+j',k+k',t+t'} \in X$ *with* $i', j', k', t' \in \{-1, 0, 1\}$; $|i'| + |j'| + |k'| + |t'| = 1$ *belongs to the same segment, or not.* h *has to be commutative.*

For scalar sample values, an obvious formulation of such a predicate could be a threshold on the difference between the sampled values of x and the adjacent sample x'. If we analyze a vector field, we could impose a threshold on the angle between the directions of x and x'. In many of our applications, however, we know in advance that our input data field will be homogeneous, such that we may assume $h(x, x') := \text{TRUE}$ for all pairs of adjacent samples x, x'.

Many applications based on region growing methods, for example Silver and Zabusky (1993), start the segmentation with sets containing single seed points. Neighboring sample points are added iteratively to these sets, as long as they are still associated with the same features (for example based on thresholding or on a global homogeneity criterion). The initial seed points often correspond to extremal points, such as local minima or maxima, of the underlying data. Since we aim for a segmentation in only one iteration over the data set, we cannot know in advance whether a connected set of samples will contain any such extremal point or not. To compensate this, we use an additional predicate for discarding segments at the end of the iteration based on any of their global attributes. This predicate is the *global selection criterion*:

Definition 9 (Global selection criterion)**.** *The* global selection criterion $g : \mathcal{P}(F) \to \{\text{TRUE}, \text{FALSE}\}$ *decides whether or not to keep a candidate four-dimensional segment based on any of its global characteristics.*

For the segmentations of jet streams, we decided to use the global selection criterion as a filter on the lifespan of the detected wind events. In order to exclude short-lived peaks of wind speed, we discard segments with a lifespan of less than 24 hours.

Algorithm 1

Input: The set of sampled atmospheric data X, the homogeneity criterion h, the local selection criterion l and the global selection criterion g. **Output:** The event graphs G_{S_i} containing all segments $S_1, \ldots, S_n$ corresponding to the detected atmospheric phenomena together with the precise event locations T.

1: $F := \emptyset; E := \emptyset; T := \emptyset$ ▷ The sets of all features, connecting edges and event locations
2: $c := \emptyset$ ▷ An array of all candidate features.
3: **for** $t := 1, \ldots, t_{max}$ **do**
4: **for each** $x_{i,j,k,t}$ with $l(x_{i,j,k,t}) ==$ TRUE **do**
5: $c_{i,j,k,t} :=$ *new_candidate_feature*$(x_{i,j,k,t})$
6: **for each** already visited neighbor $x_{i',j',k',t}$ **do**
7: **if** $h(x_{i,j,k,t}, x_{i',j',k',t}) ==$ TRUE and $c_{i',j',k',t}$ exists **then**
8: *merge*$(c_{i,j,k,t}, c_{i',j',k',t})$
9: **end if**
10: **end for**
11: **if** $t > 1$ **and** $\exists m : x_{i,j,k,t-1}$ belongs to $F_{t-1,m}$ **then**
12: $E := E \cup (F_{t-1,m}, c_{i,j,k,t})$ ▷ these edges are later replaced by real edges
13: **end if**
14: **end for**
15: $F := F \cup$ *get_real_features*(c, t)
16: *replace_candidate_edges*(E)
17: **if** $t > 1$ **then**
18: $E := E \cup$ *extended_feature_tracking*(F_{t-1}, F_t)
19: $T := T \cup$ *find_event_locations*(F_{t-1}, F_t)
20: **end if**
21: **end for**
22: **return** all connected component (S_i, E_{S_i}) of (F, E) with $g(S_i) ==$ TRUE and T

3.3. Segmentation algorithm

In the previous section, we conceptually defined the input and output of our algorithm, as well as all required predicates for the characterization of the features we want to detect. This is the basis for describing now in detail (and more technically) the implementation of our algorithm, whose general outline is shown in Algorithm 1. In the following subsections, we will investigate in detail the important steps of the algorithm.

3.3.1. Feature detection

Feature detection is the process of constructing all sets of samples corresponding to the features of interest. Single samples are selected by means of the local selection criterion, and clustered based on the homogeneity criterion. During execution, the algorithm successively adds samples to *candidate* features and merges multiple candidate features as soon as a connection is detected.

The algorithm iterates sequentially over all time steps t. At each time step t, the algorithm investigates all samples X_t starting at $x_{1,1,1,t}$ and traversing the remaining samples by in-

creasing the first three indices lexicographically. As soon as a sample $x := x_{i,j,k,t}$ fulfills the local selection criterion l, we know that it belongs to a candidate feature. It remains to check if there are any connections to neighboring samples. For this, we examine the up to three already visited adjacent samples $x_{i-1,j,k,t}$, $x_{i,j-1,k,t}$, and $x_{i,j,k-1,t}$. From these samples we select those which are already associated with a candidate feature and fulfill the homogeneity criterion h. We then merge these candidate features into one.

The algorithm represents the candidate features using a union-find data structure (cf. Tarjan, 1975; Knuth, 1997; Cormen et al., 2009). Each candidate feature is stored internally as a tree. The root of each tree is the unique representative of the candidate feature. Different candidate features are merged by transforming the root of the tree containing fewer elements into a child node of the root of the other tree. In our implementation, each candidate feature internally stores a list of indices of all samples associated with the corresponding feature, as well as a set of attributes, such as the center of mass and an approximation of the volume. The list of sample indices and the set of attributes are updated each time two candidate features are merged. To keep the number of nodes low, we add single samples with only one neighboring candidate feature directly to the existing feature instead of adding a new node to the tree. For the sake of simplicity, we omitted the distinction of this additional case in our algorithm outline. The performance of the union and find operations is further improved by using path compression. With these techniques combined, we achieve an asymptotic runtime of $O(1)$ for the union operation, and an amortized asymptotic runtime of $O(\alpha(n))$ for the find operation (cf. Tarjan, 1975), where n is the total number of samples, $\alpha(n)$ is the inverse of the function $f(n) = A(n, n)$, and $A(n, n)$ is the Ackermann function. For a direct access to the candidate features associated with each sample, we store the references in a three-dimensional array. At the end of each time step, the set of all remaining candidate features corresponds to the set of real features F_t.

3.3.2. Feature tracking

The goal of feature tracking is to identify at every time step t the relations between features of the previous time step F_{t-1} and features of the current time step F_t. This corresponds to a grouping of features belonging to the same instance of an atmospheric phenomenon. There are many possible approaches to achieve this goal, depending primarily on the attributes such as shape, size and speed of the objects to track. Relatively big and slow-moving features, with respect to the sampling frequencies of the data domain, such as jet streams, require a different approach than the tracking of, e.g., potential vorticity cut-offs, which in comparison are typically smaller and move faster. Different general approaches for feature tracking have been discussed in Sect. 3.1. For most of our current applications, it is sufficient to track features based on spatial overlaps of the samples that are associated with the features. This approach is valid since the spatial and temporal sample frequencies are generally high enough with respect to the expected size and speed of the features to track (cf. Samtaney et al., 1994). We keep track of the spatial overlaps by maintaining a set of candidate edges, which represent connections from the features of the previous time step to the candidate features of the current time step. An edge is added to the set whenever a sample $x_{i,j,k,t}$ is associated with a candidate feature and the sample at the same location of the previous time step, $x_{i,j,k,t-1}$, is associated with a feature. For a direct lookup of the candidate features of samples of the previous time step, we use another three-dimensional array. As soon as the final features are

known, the replacement of candidate edges by real edges is straightforward. In the algorithm outline, it is denoted as the *replace_candidate_edges(E)* function call.

Despite the fact that in our applications so far, the majority of continuations could be handled by testing for spatial overlap, we also identified rare cases where a feature was too small and fast moving, compared to the spatial and temporal resolution of the grid, to be tracked by spatial overlaps only. We therefore apply an additional feature tracking step, indicated by the *extended_feature_tracking*(F_{t-1}, F_t) function call in the algorithm outline. In this extended feature tracking step, we compare attributes (currently centers of mass and volumes) of pairs of unconnected features. If the differences of the attributes are below a fixed threshold, the two features are regarded as an additional continuation. This is similar to the approaches described in Samtaney et al. (1994) and Reinders (2001), but without predicting the future state of attributes by means of extrapolation.

3.3.3. Event localization

As soon as all connections between the three-dimensional features are established, we are ready to estimate the locations of the occurring events on a per-grid-point basis (indicated by the *find_event_locations*(F_{t-1}, F_t) function call in the algorithm outline). The localization of events allows us to gain important additional insights in the development of atmospheric phenomena. As illustrated in the upcoming Sect. 3.4, one possible application is the calculation of climatologies of events in order to find areas where events are particularly frequent or rare/absent. The number of events in a certain region can be a measure for the stability of the identified features in this region. In some cases specific event locations can be related to particularly interesting atmospheric processes. An example case study where the location of a single jet stream merging event could be associated with the breaking of a Rossby wave will be presented in Sect. 3.4.1. Previous methods, including those operating on the full set of samples in grid space, for example by Silver and Zabusky (1993) and Muelder and Ma (2009), only detected the existence of events, but not their precise location in grid space.

The localization of the genesis and lysis events is straightforward: We associate all samples of features without a preceding feature with a genesis event, and all samples of features without a successive feature with a lysis event. We exclude genesis events at the first time step and lysis events at the last time step because we have no information about the history or continuation of these features.

In case of the merging events, the estimation of the position is more involved. We now describe the method for merging events in detail; the localization of a splitting event is done analogously by running the procedure in the inverse time direction. The estimation of the position where multiple features merge into a new feature is based on a search for grid points that have a similar distance to any two of the single features (this corresponds to grid points lying on the bisectors of the Voronoi regions of the single features). To find these points, we apply a process similar to region growing. We use the sets of grid points of the single features as seed points and let these regions "grow" by tagging all grid points adjacent to the boundaries iteratively. The positions we are interested in are the grid points where these growing regions first touch. Figure 3.3 provides a two-dimensional illustrative example. We limit the growing to the grid points covered by the new feature. Of course, it is very unlikely

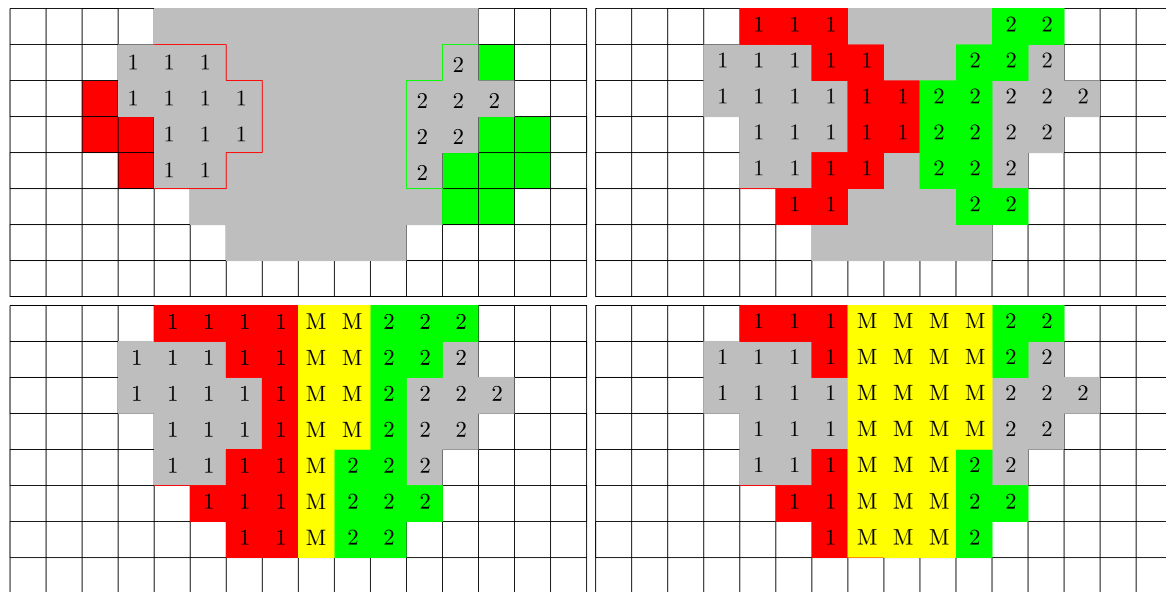


Figure 3.3.: Two-dimensional example of the applied steps for localizing a merging event. In the top-left picture, gray grid points indicate the location of the merged feature at time step t. Red and green grid points indicate the initial location of the separated features at the previous time step $t-1$. Grid points with the tag "1" mark the positions where the red and gray features overlap. Grid points with the tag "2" indicate positions where the green and gray features overlap. The pictures to the right and below show the second step of the first growing phase and the final tagging, respectively. Newly tagged regions are depicted in red and green, the positions where both regions touch are indicated by the tag "M" on yellow background. The picture at the bottom-right shows the final state at the end of the second growing phase.

that the merging occurred exactly at the position of the thin border detected by this growing process. To compensate for this, we enlarge the border into an area allowing for a certain, controllable amount of fuzziness. This enlargement is realized by letting the borders grow for a fixed number of steps in a second growing phase. The choice of the number of steps for this second phase depends on the desired output and on the intended further processing of the event locations. Choosing a greater number of steps leads to a more fuzzy and larger representation of the events.

More formally, the procedure can be described as follows: Assume there is a merging of features $F_{t-1,1}, \ldots, F_{t-1,m}$ into feature $F_{t,1}$. At first, we initialize an empty set N in which we will insert elements of the form $(i, j, k, n) \in \mathbb{N}^4$. We use these tuples in order to relate certain positions on the point lattice of our samples, specified by means of indices i, j, k, to different integer tags n. Initially, for each existing pair of spatially overlapping samples $(p, q) := (p_{i,j,k,t-1}, q_{i,j,k,t})$ with $p \in F_{t-1,g}$ and $q \in F_{t,1}$, we add the tuple (i, j, k, g) to N. In other words, we associate all overlapping sample positions with the number of the respective feature from the previous time step, see the top-left panel of Fig. 3.3 for a two-dimensional example of this process. Next, we iteratively let these tagged regions grow by tagging all untagged positions adjacent to these regions with the respective numbers. This growing is limited to positions of $F_{t,1}$, see the top-right panel of Fig. 3.3. If in any growing step a single

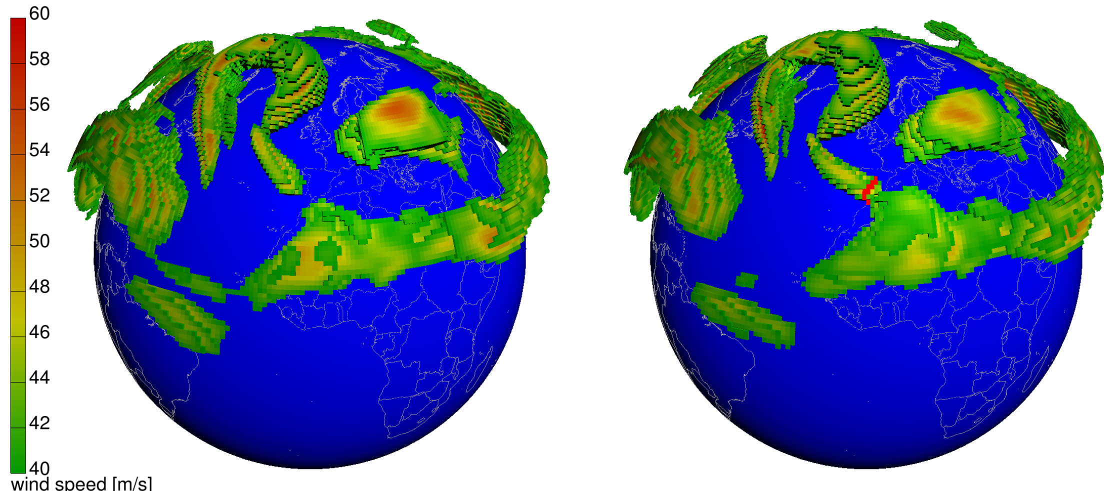


Figure 3.4.: Two successive time steps of a jet stream segmentation at 18 UTC 22 January 2007 (left) and 00 UTC 23 January 2007 (right). The red highlighted region in the panel on the right indicates the location of a merging event. At the previous time step (see panel on the left) the two jet stream features were still separated. All non-event samples are shaded according to the wind speeds at their respective positions.

position is to be tagged by two or more different numbers, or if an already tagged position is to be additionally tagged by a different number, we found a position where multiple growing regions touch. We add a special event tag to N, replacing all other tags at this position, which indicates a merging. This special tag is excluded from any future growing steps. The iteration ends as soon as all positions corresponding to $F_{t,1}$ are tagged, see the bottom-left image of Fig. 3.3. In a final growing phase, we let the regions marked by the special event tag grow for a fixed small number of steps. As before, this region growing is limited to the positions covered by samples of $F_{t,1}$. The bottom-right picture of Fig. 3.3 shows the final result of a two-dimensional example event localization. A real example of the three-dimensional merging localization from our jet stream segmentation is depicted in Fig. 3.4 and discussed later in Sect. 3.4.1.

3.3.4. Efficiency

In terms of computational efficiency, our method profits from simplifications made on the basis of the prior knowledge about the atmospheric features we want to identify and track. Therefore feature identification, tracking, and event localization can be performed within a single iteration over the data set. Other methods, for example the approach by Muelder and Ma (2009), require at least one iteration over the whole data set, if all new features and holes in existing features are to be detected. Related approaches, for example Silver and Zabusky (1993), use octrees or other spatial data structures to achieve speedups. Since our algorithm selects, clusters and tracks features during a single iteration over the data set, there would be no benefit from the additional maintenance of a spatial data structure. The only exception would be if the available memory was very limited, such that the two additional three-dimensional arrays we use for storing the links to the candidate features of

the current time step, and to the features of the previous time step, could not be allocated. In such cases, it would be reasonable to replace these two arrays by an appropriate spatial data structure, in order to save memory at the expense of speed.

3.4. A climatology of upper-tropospheric jet streams and their events

The new segmentation algorithm has been implemented and tested for different types of atmospheric phenomena, for example ozone holes, jet streams, cyclones, and filaments of potential vorticity. In this section we present results from the first extensive application of our algorithm, which was the computation of a two-years climatology of three-dimensional upper-tropospheric jet streams and their merging and splitting events. Wind fields were taken from the operational ECMWF analyses for the years 2007 and 2008, available every six hours on 60 vertical levels interpolated to a regular longitude/latitude grid with a resolution of one degree.

For the detection of upper tropospheric jet streams, we choose the local selection criterion to be a height-dependent wind speed threshold. We accept all samples below 100 hPa (i.e., grid points with pressure larger than 100 hPa) with a horizontal wind speed exceeding $40\,\mathrm{m\,s^{-1}}$. This criterion is motivated by the wind speed threshold criterion used by Koch et al. (2006), who considered jet streams as two-dimensional features of the vertically integrated wind speed between 100 and 400 hPa. Strong winds in the stratosphere, that is at levels above 100 hPa, are excluded to focus on jet streams in the upper troposphere. Compared to Koch et al. (2006) we use a $10\,\mathrm{m\,s^{-1}}$ larger threshold, due to the extension of considering jet streams as fully 3D features instead of 2D features of the vertically averaged wind speed. As a side remark, note that recently also Schiemann et al. (2009) and Manney et al. (2011) introduced alternative jet identification schemes, which focus on the jet axis (i.e., the location in meridional vertical cross-sections where the horizontal wind speed is maximum) and therefore avoided the vertical averaging of the wind speed as performed by Koch et al. (2006). As a global selection criterion, we choose a threshold on the total lifetime of a four-dimensional segment. The idea behind this threshold is to separate real jet stream events from short-lived wind events that exist for less than 24 h. Due to the general homogeneity of the wind speed data from global analyses, we do not state any explicit homogeneity criterion.

Before presenting the climatological results, we start with a brief case study of an interesting episode of Rossby wave breaking and an associated jet stream merging event over the North Atlantic. The aim of this case study is to illustrate the relevance of such jet merging events for atmospheric dynamics.

3.4.1. A Rossby wave breaking event over the North Atlantic

A prominent chain of events including rapid cyclogenesis, an intense warm conveyor belt, the formation of a blocking anticyclone, a subsequent Rossby wave breaking, and eventually a jet stream merging event occurred during the time period 20–23 January 2007 over the North Atlantic. Isentropic charts of potential vorticity (PV) on 315 K at 12:00 UTC 20 January reveal a prominent trough with stratospheric PV (i.e., more than 2 pvu) over Eastern Canada

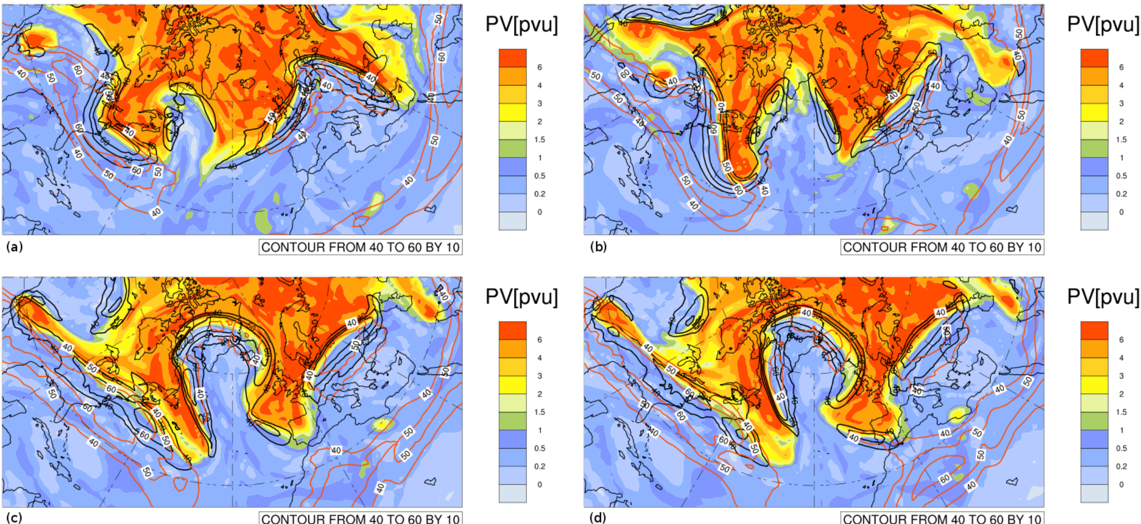


Figure 3.5.: Isentropic potential vorticity on 315 K (colors, in pvu) and wind speed on 315 K (black contours for 40, 50, and 60 $\mathrm{m\,s^{-1}}$) and on 350 K (orange contours for 40, 50, and 60 $\mathrm{m\,s^{-1}}$) at (a) 12 UTC 20 January 2007, (b) 12 UTC 21 January 2007, (c) 18 UTC 22 January 2007, and (d) 00 UTC 23 January 2007. The 2-pvu contour denotes the dynamical tropopause.

and an equally prominent ridge with tropospheric PV (i.e., less than 2 pvu) downstream, over the Western North Atlantic (Fig. 3.5a). Rapid surface cyclogenesis occurred during the previous day beneath the upper-level trough leading to a mature cyclone with a core pressure of less than 970 hPa situated over the Gulf of St. Lawrence. Trajectory calculations (not shown) indicate that the rapid cyclone evolution was associated with a prominent warm conveyor belt (Browning, 1990; Wernli and Davies, 1997), which ascends from the cyclone's warm sector almost to the 310-K isentrope and enlarges the upper level ridge during the following day (Fig. 3.5b). In parallel, the trough over the Western North Atlantic elongates into a filamentary "PV streamer" (Appenzeller and Davies, 1992) and a downstream trough evolves to the west of Europe. In between, the upper-level ridge develops into a persistent atmospheric blocking. At this time intense jet streams are present on 315 K (black contours) along both flanks of the PV streamer and the downstream trough. On the 350-K isentrope, jets exist over the US east coast, over Central Europe, and over Northern Africa. Whereas the first two of these jets are partially aligned with the jet systems on 315 K, the African jet is shallower and only present on the higher isentrope.

During the following 30 h the blocking becomes more prominent (reaching a maximum sea level pressure larger than 1045 hPa), the downstream trough protrudes to the Iberian Peninsula where it triggers the formation of a Mediterranean cyclone (not shown), and the anticyclonically curved jet stream to the north of the blocking on 315 K intensifies (Fig. 3.5c). Six hours later, at 00:00 UTC 23 January (Fig. 3.5d), the downstream trough reaches into the Western Mediterranean and its associated jet stream on 315 K becomes vertically aligned with the northern extension of the African jet on 350 K. A similar event has been described by Martius et al. (2010) (their Fig. 5) who emphasized the importance of such a jet merging event for a kinetic energy transfer from the extratropical to the subtropical waveguide. Recently, Martius and Wernli (2012) corroborated the relevance of these extratropical wave

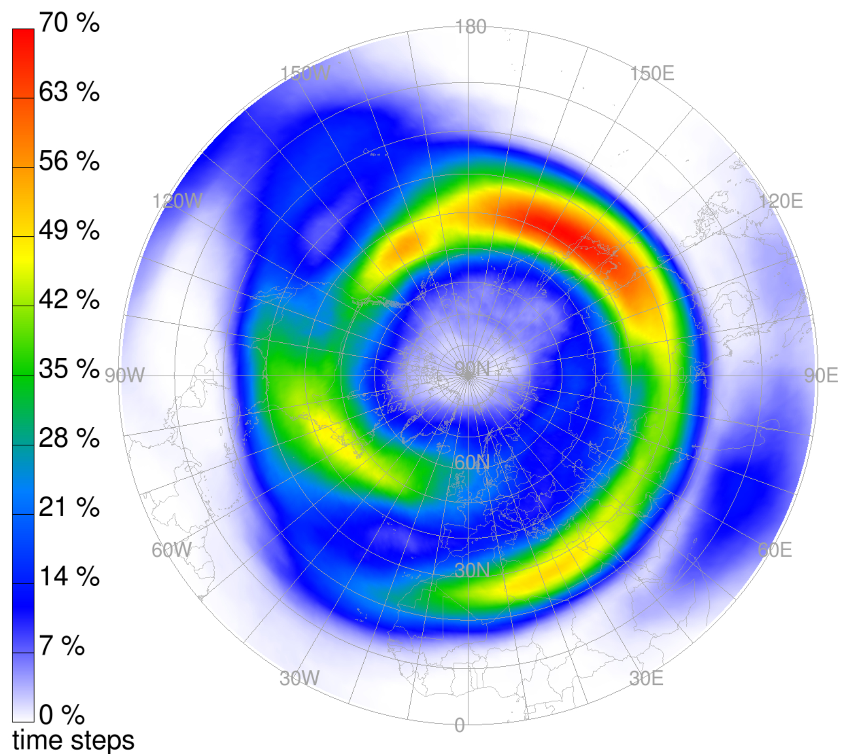


Figure 3.6.: Two-years climatology of the spatial distribution of jets identified with the new segmentation method. Values indicate the frequency of jet stream occurrences (in %).

breaking events (and associated jet mergings) for the intensification of the subtropical jet over Africa.

Figure 3.4 shows the three-dimensional structure of the jet streams associated with the jet stream merging event over Gibraltar between 18:00 UTC 22 January and 00:00 UTC 23 January, as identified with the new segmentation algorithm. The previously discussed jet streams to the north of the blocking and over Europe, and the elongated subtropical jet reaching from Northern Africa to the Western North Pacific are clearly visible. The red region in the second panel highlights the merging of the extratropical and subtropical jets, as discussed above. This brief case study illustrates that jet merging events can be associated with prominent events of extratropical Rossby wave breaking and an associated momentum transfer from a midlatitude to a subtropical jet stream. According to climatologies of Rossby wave breakings (e.g., Wernli and Sprenger, 2007) such events are most likely to occur over the Eastern North Atlantic/Western Mediterranean and Eastern North Pacific/Western North America. It will be therefore interesting to consider the climatological occurrence of jet streams and their merging (and splitting) events in the following subsections.

3.4.2. Frequency of jets, jet merging, and jet splitting

With the results of the segmentation, we are able to compile a climatology of the frequency of jet streams and their genesis, lysis, merging, and splitting events. Figure 3.6 depicts the

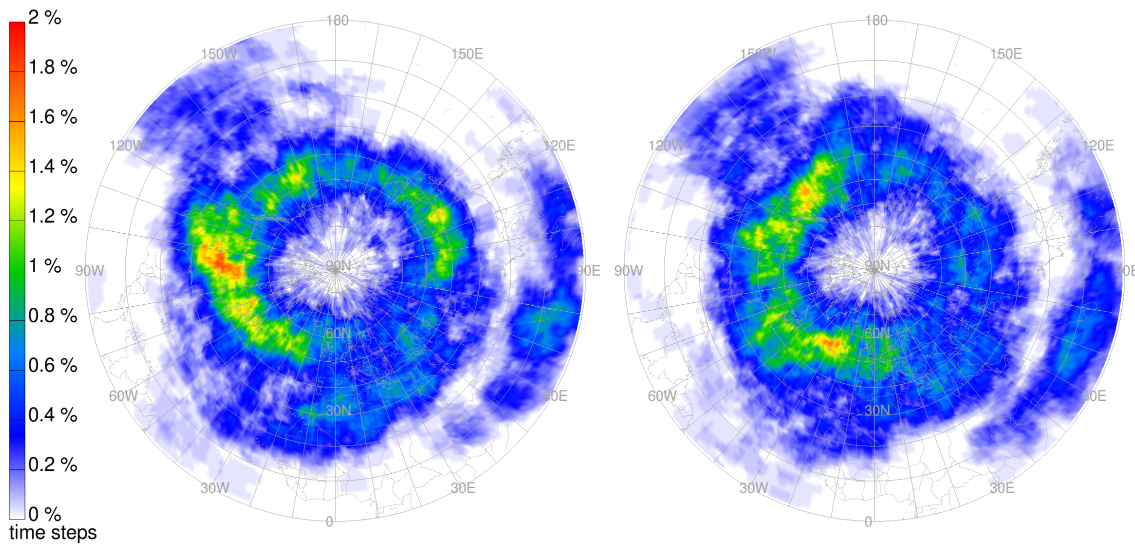


Figure 3.7.: Two-years climatology of the spatial distribution of (left) jet stream merging and (right) jet stream splitting events. Values indicate the event frequencies (in %).

spatial distribution of all detected jet streams during the two-year time period. In order to obtain a two-dimensional plot, we calculated the jet stream frequency at each horizontal position as the ratio between the number of time steps at which a jet stream was detected at any observed vertical level (i.e., levels below 100 hPa), and the total number of time steps.

Our results agree favorably with the previous climatology by Koch et al. (2006), despite the different time periods (2007–2008 vs. 1979–1993) and the slightly different jet stream definition. The overall frequency maximum is located over Japan (with values exceeding 70 %) and secondary maxima are located over Newfoundland and Libya/Egypt. The general spiral-like shape of the region with high jet stream frequencies is also well reproduced, corroborating the reliability of the new approach.

In addition, with the aid of our new method it is straightforward to compute climatological frequency distributions of merging and splitting events, as shown in Fig. 3.7. These patterns are obviously very different from the overall jet stream frequency distribution. Clear maxima of both mergings and splittings occur in the Western Northern Hemisphere, in particular over North America. Secondary maxima are found to the north of the Tibetan Plateau and over North Africa. This indicates on the one hand that the very high jet stream frequency over the Western North Pacific is associated with very stable jets that experience comparatively little merging and splitting events. On the other hand the much higher frequency of these events in regions mentioned above is qualitatively consistent with a high frequency of Rossby wave breaking (Wernli and Sprenger, 2007). Future studies will be required to better understand the global linkage between wave breaking and jet stream merging and splitting events.

3.4.3. Lifetime and stability of jet segments

The climatology indicates that jet merging and splitting is particularly frequent in one half of the Northern Hemisphere (from approximately 120° W to 60° E). We call this region the “North Atlantic region”, in contrast to the “North Pacific region” where the identified jet stream segments appear to be more persistent. In order to get statistical evidence for the differences in the stability of these segments in the two semi-hemispheres, we further investigate the size and lifetime of these jet segments.

As a direct consequence of the way we perform our feature tracking (see Sect. 3.3.2), major jet streams that are first separated, but merge at some later point in time, are associated with the same 4D segment. Here we are interested in obtaining the statistical distribution of the time span between consecutive merging and splitting events of the identified 4D segments. This allows differentiating between, for instance, very stable, long-lived and maybe quasi-stationary jet stream segments, and highly transient segments that break apart and re-merge at frequent intervals. In order to achieve this goal, it is useful to introduce *sub-segments*.

A sub-segment is a subset of a 4D segment without any *undesired* (depending on the specific application) merging, splitting, and continuation events. As indicated above, in case of the jet streams we want to discard all major merging and splitting events between large individual connected regions. If, for example, a large feature splits up into two separate large features, we would rather consider this the creation of a new jet stream with an individual lifespan, instead of directly associating both jet stream features with the same object. Several techniques for the extraction of sub-segments can be applied and are available as part of our implementation. In the following, we present a very strict method that can be applied for the analysis of the jet stream segments. In later sections of this chapter, we present further methods that were developed for the segmentation of cyclones (cf. Sect. 3.6.3).

The decision whether or not a connection between two 3D features of a segment is considered a connection between nodes of the same sub-segment is taken as follows: For each feature $F_{t,i}$, we pick the successor $F_{t+1,j}$ whose size is closest to the size of $F_{t,i}$ (if present). If now out of all of $F_{t+1,j}$'s predecessors $F_{t,i}$ is the one with the closest size to $F_{t+1,j}$, we register a valid sub-segment continuation between these two features. Out of these sub-segment continuations we are able to construct the complete sub-segments of each 4D segment.

Additionally, for every sub-segment, we compute its average center of mass, size and lifetime and use this information to attribute the sub-segments to either the North Atlantic or the North Pacific region. The size of the sub-segments is approximated by the number of respective grid points, weighted with the cosine of the latitude of the corresponding position. Figure 3.8 shows the statistical results of this analysis for the two semi-hemispheres. There are many relatively small and short-lived segments in both regions. Large segments are less frequent. Very large segments, which contain about 15 000 grid points or more, have a lifetime of only up to 400 h in the North Atlantic. In contrast, segments of this size (or larger) all have a lifetime of about 400–3500 h in the North Pacific. Clearly, these huge and long-lived segments dominate the jet stream pattern in the North Pacific area and contribute essentially to the maximum jet stream frequency in this region. This finding is also consistent with the previously discussed results on the frequency of jet stream splitting and merging events over the North Atlantic and North Pacific, respectively.

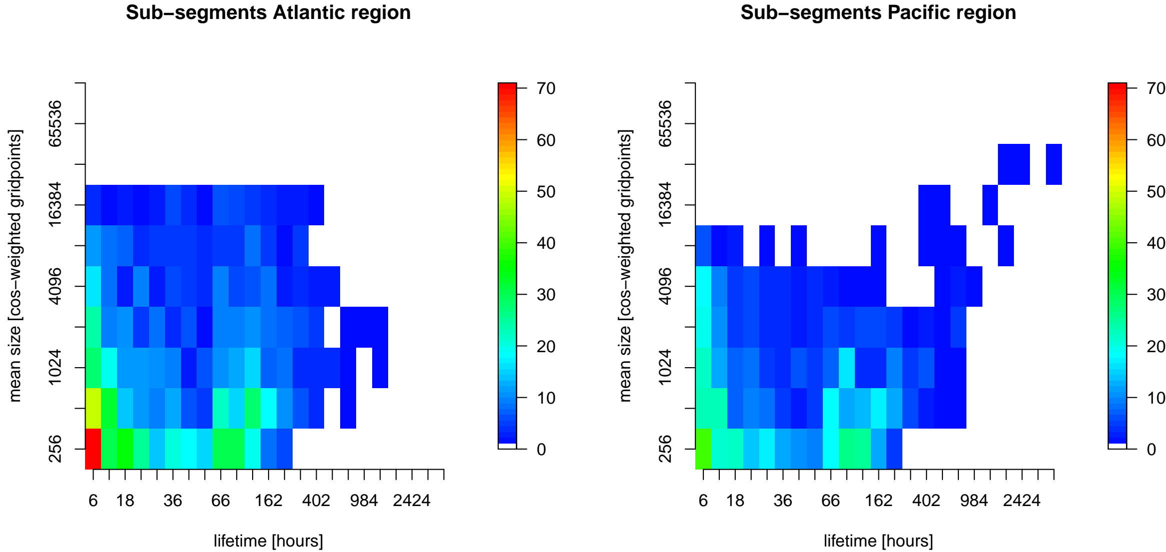


Figure 3.8.: Histograms relating the size to the lifetime of jet stream sub-segments in the North Atlantic (left) and North Pacific (right) region, respectively. The colors represent the number of sub-segments of the years 2007 and 2008. The size of each sub-segment is approximated by the sum of the cosines of the latitudes of the corresponding grid points.

3.5. Extensions of the algorithm

Besides this initial application of the algorithm for the identification and tracking of jet streams, we also experimented with the segmentation of additional atmospheric features. These experiments were partially performed in collaboration with students from the Johannes Gutenberg University of Mainz, the University of Bern, and from the ETH Zurich, in the contexts of their individual work. These additional applications enabled us to get an overview of the strengths of the algorithm, for example the fast creation of climatologies from large data sets, but they also raised our awareness of the common pitfalls and weaknesses.

A general problem of many segmentation algorithms is the *over-* and *under-segmentation* of the data, i.e., the detection of too few or too many individual segments. In our case, as we perform the segmentation on time series of three-dimensional data, many seemingly distinct 3D features may share the same connected component of the complex 4D event graph. Whether or not this is recognized as a under-segmentation often depends on the actual application. An over-segmentation can typically be observed when the temporal resolution is not sufficient for the tracking of fast moving small objects. Both problems have been addressed in different ways since the initial publication of Limbach et al. (2012). In the following parts of this section, we discuss several of these extensions of our segmentation algorithm.

In general, the under-segmentation can be addressed at two different stages of the segmentation algorithm: Either by improving the three feature detection criteria, such that no connection occurs in the event graph, or by a post-processing of the event graph, using the concept of sub-segments as demonstrated above in Sect. 3.4.3. It is not always a trivial task to choose the correct way of addressing an under-segmentation of the data. In practice, find-

ing suitable detection criteria that cover all cases of interest, allow a tracking through spatial overlaps, and avoid an under-segmentation is not always possible. Sometimes, it may even be the case that connections between features in the event graph are of interest and should be preserved, but at the same time a more detailed subdivision of the segments is required for a statistical analysis of other aspects of the phenomenon. The current implementation of the algorithm contains an extension that allows to apply *double thresholding* (see, for example, Jain et al., 1995, page 85f.) for the prevention of under-segmentation at the stage of feature detection, without the need to reduce the final feature sizes. This technique is discussed in detail in Sect. 3.5.1. We also increased the number of available sub-segment detection techniques. The new methods are described in Sect. 3.5.2. The practical applications showed that a combination of improved detection criteria and an additional sub-segment detection often leads to useful and valuable results. An example for such a combined approach can be found in Sect. 3.6, where the segmentation of cyclones is discussed in detail.

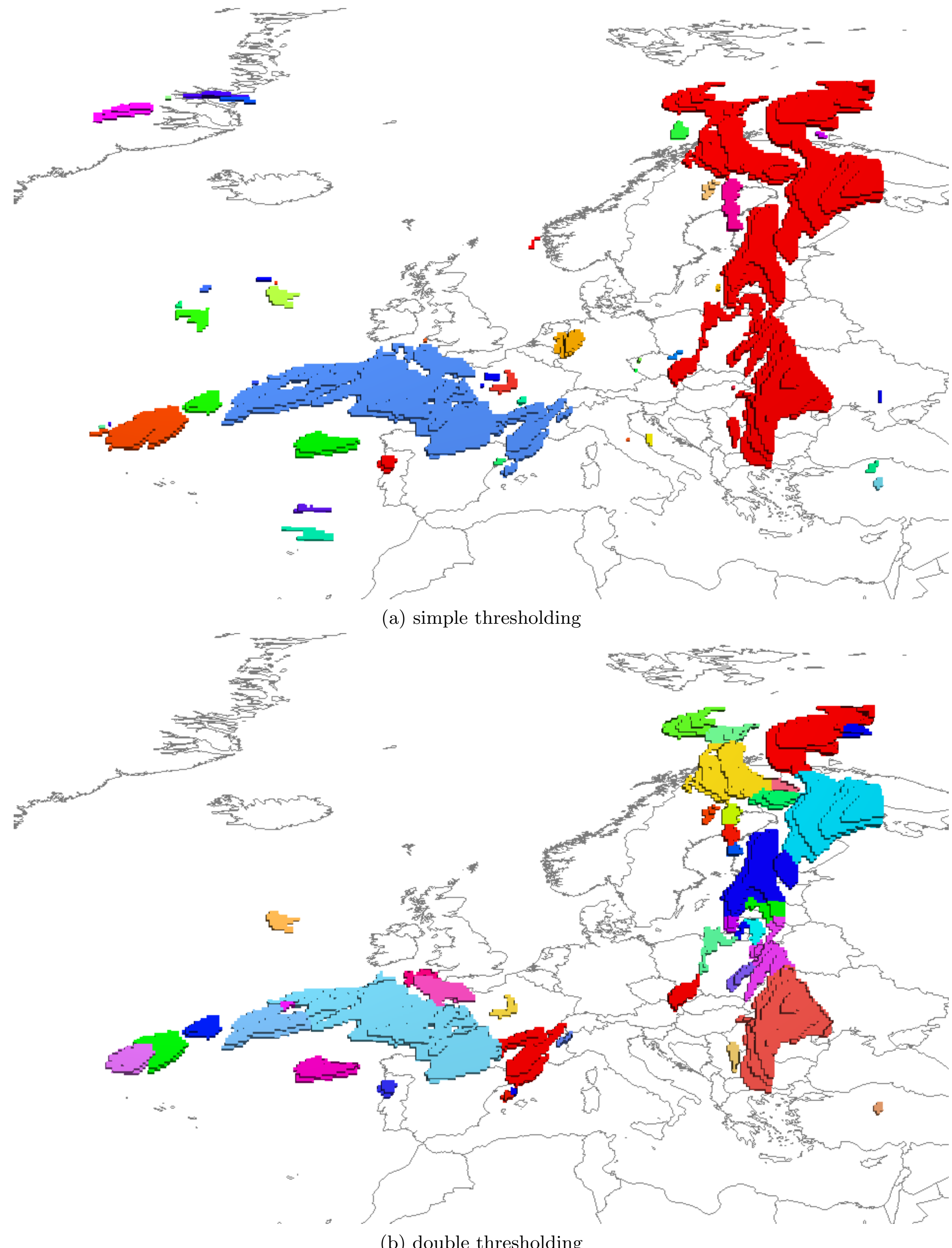
As mentioned above, an over-segmentation is commonly observed for fast moving, small objects. A first implemented technique for an improved feature tracking based on the dilation of the features can be found in Sect. 3.5.3. We propose several further ideas for improving the tracking mechanism in order to avoid a temporal over-segmentation in the final Sect. 3.7.

3.5.1. Double thresholding

Double thresholding is a segmentation technique that extends the basic idea of simple thresholding with a single threshold T. In our implementation, simple thresholding is available as a local selection criterion. In practice it is often hard to find a fixed value T such that the resulting features are large enough but separate phenomena are still recognized as individual features. The problem may persist even if the threshold T is realized as a function of the location and of the time step of the local grid point. The region growing approach of our algorithm connects 3D features as soon as a single pair of adjacent grid points is found (and their local homogeneity criterion is fulfilled). Some phenomena, for example the ice-water clouds that are subject of the research of Patrick Neis (University of Mainz), may touch at some point but would still be recognized by the researcher as two individual objects. An example of such an under-segmentation of ice-water clouds can be found in Fig. 3.9a. Especially the large red feature in the right part of this figure is a clear example of an under-segmentation.

Double thresholding tries to avoid such an under-segmentation by using two different thresholds: A tighter threshold T_1 is used for the identification of seed regions, each of which is associated with a distinct feature. In a second step, these seed regions are grown into larger features by iteratively adding adjacent grid points, taking into account a weaker threshold T_2. In our implementation, T_2 is controlled by a linear factor τ, such that $T_2 := \tau \cdot T_1$. The results of an example application of double thresholding can be found in Fig. 3.9b. The large feature dominating the right part of Fig. 3.9a is now split into several smaller features. Since we implemented the new method in order to avoid an under-segmentation of the data, we perform feature tracking only on the seed regions defined through the tighter threshold T_1, and not on the enlarged features resulting from the additional growing phase.

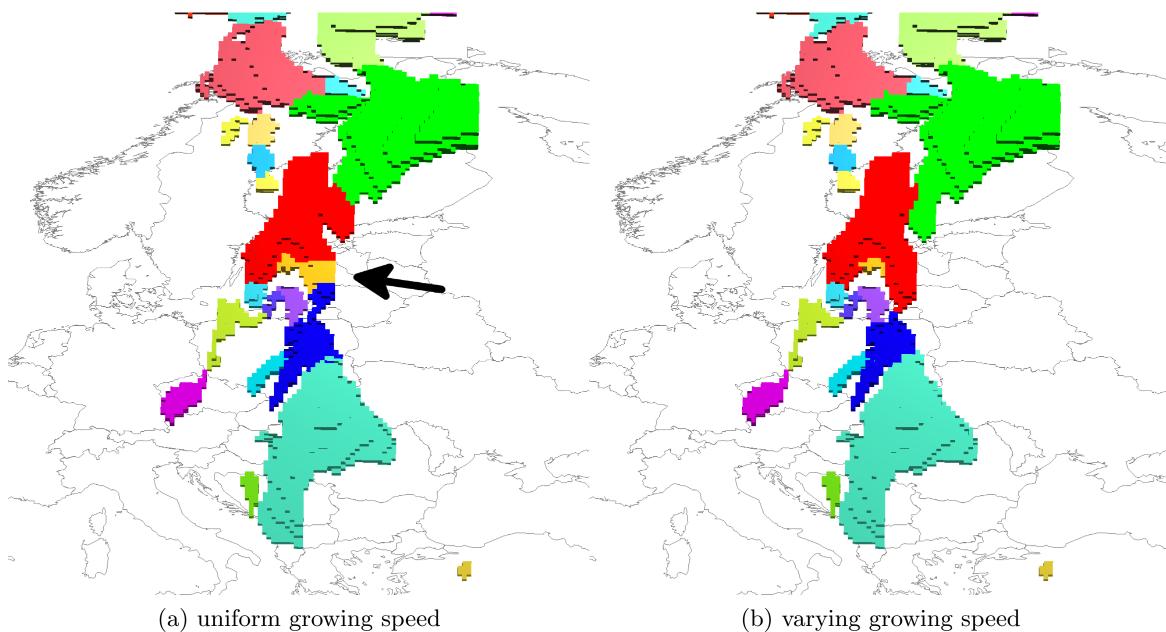
The results of the double thresholding not only depend on the selection of the two thresholds



(a) simple thresholding

(b) double thresholding

Figure 3.9.: Example segmentation with and without double thresholding. Both plots depict single features representing clouds, each feature has its individual color. The variable used for segmentation is RHI (relative humidity over ice). In (a), the threshold is 110 %, in (b), $T_1 = 120\,\%$ and $T_2 = 110\,\%$.



(a) uniform growing speed

(b) varying growing speed

Figure 3.10.: Comparison between uniform and non-uniform growing speed of the seed areas. In (b), the neighbouring grid cells are added to a region with a certain probability only. The probability at each grid cell is defined by the position of the variable value in the interval $[T_1, T_2]$. The arrow indicates an area were a uniform growing speed performs poor in comparison to a non-uniform growing speed.

T_1 and T_2 (or T_1 and τ, respectively), but also on the way in which the growing of the seed features is realized. In the previous example from Fig. 3.9, the area of the features was iteratively increased by adding all directly adjacent grid points in the same way the region growing was realized for the detection of the event positions (cf. Sect. 3.3.3). With this uniform growing speed, the location of the borders between the 3D features is not always optimal. For example, if we look at the area marked by the arrow in Fig. 3.10a, we clearly see that the yellow and blue-colored features are not distributed in an intuitive way. One problem here is the blue feature spreading across the thin passage of grid points northwards into the wider region where the yellow and red segments are located. In a similar way, the yellow feature spreads eastwards into a wider area where unnatural looking straight borders with the neighbouring red and blue features are generated. Intuitively, we would rather locate the borders between these features at the thinnest point of their connection. In order to improve the subdivision in such a way, we additionally implemented a growing scheme that varies the growing speed of the seed regions, making it harder for a feature to expand across thin regions.

Two different tasks had to be addressed for the implementation of this improved growing scheme: The formulation of a method for the determination of the growing speed at each grid point, and the extension of the growing algorithm for the realization of these non-uniform growing speeds on a discrete grid. The initial idea was to define the growing speed as a function of the thickness of the detected features at each grid point. For our relatively homogeneous atmospheric data sets, the thickness of a feature can usually be approximated

by the relative position of the data value within the interval $[T_1, T_2]$ at each grid point. Thick areas of a feature usually have values with a higher distance to the weaker threshold T_2, in contrast to grid points of thin areas that have usually values closer to T_2. We can therefore define the growing speed as

$$\gamma(i, j, k, t) := \frac{X_{i,j,k,t} - T_2}{T_1 - T_2}$$

where X is the set of samples as introduced in Definition 2.

For the realization of these non-uniform growing speeds on a discrete grid, we implemented a probability-based method[2]. In the process of region growing, we only add an adjacent grid point with index (i, j, k, t) to any region with the probability of $\gamma(i, j, k, t)$. In our implementation, we use a discrete approximation of the probability γ with an enforced fixed minimum probability in order to ensure the termination of the growing process. The computational overhead of the growing phase remains low in general, since the growing phase has linear complexity in the number of candidate grid cells, which is typically much smaller than the total number of grid cells. An example result demonstrating the difference between uniform and non-uniform growing speed can be found in Fig. 3.10. The problems visible in Fig. 3.10a that we discussed above are avoided by applying this technique.

Inspired by these promising results, we plan to apply a modified version of the non-uniform growing method for an improved detection of event locations (cf. Sect. 3.3.3) in the future. For this, the real geographic distances between the centers of each grid point and the local wind speed could be considered for determining the growing speed at the transition between two neighbouring cells.

3.5.2. Extended sub-segment detection

The attribution of a segmentation as an under-segmentation can be very subjective. For some applications, the temporal connections between distinct strong peaks (usually a short series of directly connected large and pronounced 3D features) of a phenomena can provide valuable insights. For other applications, it can be useful to further subdivide the event graph of a 4D segment into smaller subgraphs that can be associated with these distinct peaks. For the realization of such a subdivision, the concept of the detection of sub-segments was introduced.

In case of the sub-segment computations for jet streams, we applied a very strict criterion that was based on a symmetrical search for features of similar size (cf. Sect. 3.4.3). This method had the side-effect that the sub-segments contained no merging and splitting events at all. This was sufficient for the creation of the statistics on the jet stream life spans, but for the analysis of different atmospheric phenomena, merging and splitting events can become relevant.

Therefore, we developed and implemented three additional methods for the classification of sub-segments. These methods were developed primarily in the context of the cyclone segmentation (cf. Sect. 3.6) in close cooperation with Lukas Papritz (ETH Zurich). The first

[2]The initial idea for this came from a discussion with Christian Hundt (Johannes Gutenberg University of Mainz).

method is very simple: A sub-segment continuation is detected between a feature and its successor of the next time step with the most similar size. This has the effect that simple merging events of features with a single successor, as well as mergings with similar sized features are kept, but the connections to small features splitting off of a continuously large feature are removed. In fact, since each feature only has a single successor in a sub-segment, splittings are always replaced by simple continuations.

The second method keeps all connections between two features if the ratio between the size of the smaller feature and the size of the larger feature stays above a user-specified threshold. In contrast to the first method, this method prevents mergings of features with significantly different sizes. In addition, it preserves certain splittings, as long as the sizes of the involved features stay in the defined range.

In many applications, we want to obtain sub-segments for which we can guarantee that all considerably large features have a direct connection by a single edge in the event graph of a sub-segment. None of these two methods presented above, however, is able to separate two or more unrelated large peaks of a phenomenon reliably. A large feature of a sub-segment may shrink significantly over some time steps first, but grow into another major feature after a certain amount of time. Mergings (and splittings, in case of the second approach) between small features can also be part of a sub-segment.

These drawbacks led to the development of an additional method for the sub-segment detection. The first step of this method is the division of the 3D features of the event graph into *major features* and *minor features* based on an additional fixed threshold on the feature size. All connections between major features are kept, allowing direct interactions of important occurrences of the phenomena to be part of a sub-segment. The direct connections between minor features are, in contrast, selected using the original sub-segments criterion based on a two-ways size comparison. Since this original criterion does not include any mergings and splittings, we obtain single "strings" of minor features. Each of these strings corresponds to a path in the original event graph that could have connections to a single or to multiple major feature both at its beginning and at its end. In both cases, we first pick the major feature with the most similar size compared to the directly connected minor feature of the string, if it exists, such that we have either one or no major feature at each end of the string. If one end of the string is empty, and the other end is connected to a major feature, we associate the whole string with the sub-segment of this major feature. If there is no major feature at any end of the string, we create a new sub-segment and associate the whole string with it.

In the special case of two major features being present at the beginning and at the end of a string of minor features, we made the rather arbitrary decision to associate the whole string with the feature at its end. By doing this, we interpret the splitting off of a small feature from a major feature as the genesis of a new sub-segment, in contrast to the interpretation of a decaying chain of features leading to a lysis event, when the small feature merges into the next major feature. Another possible way of dealing with this special case would be to break the chain of minor features up at some reasonable point, for example in the middle or at the continuation with the highest size difference between the connected minor features.

This technique for sub-segment detection ensures that no two major features that are only connected by means of minor features are associated with the same sub-segment. We applied this third method as part of our 3D cyclone segmentation (cf. Sect. 3.6) and achieved

promising results. For the future, we plan to further improve the technique of classifying the 3D features as major and minor features. At the moment, our method can lead to an over-segmentation in cases where the sizes of connected features are closely oscillating around the fixed threshold.

3.5.3. Feature dilation

Our new segmentation algorithm performs a feature tracking based primarily on spatial overlaps. As argumented above, assuming spatial overlaps between samples of the same phenomenon at two consecutive time steps is valid, as long as the temporal sampling resolution is high enough with respect to the speed of the objects we want to track (cf. Samtaney et al., 1994).

In practice, however, we already dealt with cases where the temporal resolution is not sufficient for the tracking of all features of interest, leading to an over-segmentation of the data. Sometimes, a first work-around is the choosing of a weaker threshold, in order to enlarge the features and to enforce spatial overlaps. However, this is often not practicable, as the final 3D features should reflect the actual, smaller objects of interest. There are also cases where the feature selection was already applied externally, such that the input data we want to perform the tracking on is only available as bit fields. This was, for example, the case for the studies of Christine Aebi (University of Bern, Aebi, 2012).

For such cases, we implemented a technique known from image processing called *dilation* (see, e.g., Serra, 1982; van Herk, 1992). Van Walsum (1995) proposed the usage of dilation after feature detection for his own segmentation approach. We apply a very basic version of dilation to the feature data during the feature extraction phase: We define a 3D cubical kernel K with a fixed edge length, given as number of grid points, and replace the original local selection criterion l by l' defined as follows:

$$l'(i,j,k,t) := \bigvee_{(i',j',k') \in K} l(i+i', j+j', k+k', t)$$

Special care has to be taken at the grid boundaries, as usual. This formulation results in 3D features containing additional grid points at the original boundaries induced by l. Our application contains an option to exclude the additional grid points from the final features, and use them only for the detection of overlaps during feature tracking.

3.6. Identification and tracking of cyclones

In this chapter, we present a final example application of our feature detection and tracking algorithm. The example demonstrates the utilization of many of the previously introduced extensions to our algorithm. In addition, the feature extraction is based on more complex selection criteria than it was the case in previous applications.

The next task we focused on after having finished our first attempts regarding the segmentation of jet streams, was the identification and tracking of cyclones as full, three-dimensional objects developing over time. In a joint work with Heini Wernli and Lukas Papritz from the

ETH Zurich we formulated, tested, and fine-tuned a complex three-dimensional criterion for the detection of the cyclones. This criterion is presented in Sect. 3.6.1. In order to avoid problems with under-segmentation, which occurred when we applied the algorithm on data sets covering a long period of time, we applied one of the new methods of sub-segment detection. This method was developed specifically for the application on cyclones and is discussed in Sect. 3.6.2. The algorithm was tested on several case studies, and was utilized for the computation of a five-years climatology of cyclones on multiple height levels. In Sect. 3.6.3, we present some first results of this climatology, as well as an example case study covering the episode of storm *Kyrill* from 2007.

3.6.1. 3D segmentation criterion

Cyclones are complex, three-dimensional atmospheric features. In contrast to existing methods operating on 2D fields (e.g., Raible et al., 2008, and studies mentioned in Sect. 3.1) we detect and track the cyclones as full 3D objects. We tested several different selection criteria for the detection of cyclones. For this, we had to take into account time series of three-dimensional atmospheric variables, and could not, for example, use two-dimensional fields such as sea level pressure (SLP) (e.g. used by Wernli and Schwierz, 2006), vorticity derived from SLP (Pinto et al., 2005), or geopotential height at a fixed pressure level (Blender and Schubert, 2000). In our first attempts, we focused on the cyclonic circulation by imposing lower limits on the total vorticity and on the curvature vorticity of the underlying wind field (cf. Bell and Keyser, 1993). We matched the detected features with the two-dimensional SLP field and compared the features with known cyclones from several case studies. These comparisons showed that neither type of vorticity is a suitable criterion for the feature extraction.

Heini Wernli proposed the application of a threshold profile based on the ten-days running mean of the 3D geopotential height Z. We precompute the deviation of the Z values from this running mean at each position of the data set as follows:

$$Z_{anom}(x, y, z, t_0) := \frac{\sum_{t \in \Theta} Z(x, y, z, t)}{|\Theta|} - Z(x, y, z, t_0)$$

where $\Theta := \{t_0 - \text{five days}, \ldots, t_0, \ldots, t_0 + \text{five days}\}$. In a first attempt, we computed a vertical threshold profile $p(z)$ based on the spatial and temporal mean negative Z_{anom} values for each of the lowest model levels from the ground up to $\approx 200\,\text{hPa}$ over the time period of interest. This is reasonable since the variability of Z_{anom} varies with the height. The selection criterion can then be formulated as follows:

$$Z_{anom}(x, y, z, t) < \alpha \cdot \min\left(p(z), Z_{max_anom}\right)$$

The factor α controls the extension of the extracted features, and the fixed value $Z_{max_anom} < 0$ is an upper limit for the negative values of $p(z)$ in order to avoid noise at height levels (or in certain zonal regions, see below) with a mean negative Z_{anom} value that is close to zero. For the investigated storms Renate, Kyrill, and Klaus, we experimented with this setup and varying values for α between 3.0 and 5.0, as well as with a Z_{max_anom} value of -10, -15, and -20.

Our experiments showed that this setup leads to segments that fit the expected locations and extents of the storms much better. Nevertheless, we still observed a lot of small, short-living features that we identified as false positives. In order to lower their number and to focus on the most important and severe cyclones of the investigated episodes, we further refined the extraction by using a global selection criterion taking into account a second factor $\alpha_{GSC} > \alpha$. Segments whose global minimum Z_{anom} value over all samples of all time steps stays above $\alpha_{GSC} \cdot \min(p(z), Z_{max_anom})$ are filtered out by this global criterion. In addition, we replaced the vertical threshold profile $p(z)$ by an extended threshold profile $\tilde{p}$ taking into account both the seasonal and the zonal variation of the Z_{anom} values. Lukas Papritz computed a new data set with values $\tilde{p}(\varphi, z, t_{month})$ using a monthly mean and 181 zonal sections on 37 vertical levels based on geopotential height data from the years 1979–2011. We applied a temporal linear interpolation between the profiles of the two closest months. For this, the profiles for the interpolation at a time step t are weighted according to the distance of t to the middles (day 15) of these two months. The computation of zonal mean values resulted in some values that are close to zero, especially in tropical and subtropical regions. In order to avoid noisy results in these regions, a reasonable value of Z_{max_anom} has to be chosen.

By combining these improved feature extraction criteria we just presented, the occurrences of cyclones in the atmosphere can be detected with high accuracy. We were able to verify the detection method and to discover new aspects in the three-dimensional development of certain cyclones on different model levels by applying the algorithm on analysis data covering some known storm events, e.g. the already mentioned storms Renate, Kyrill, and Klaus. Lukas Papritz additionally computed a five-years climatology of cyclone frequencies on different height levels (see Sect. 3.6.3 for examples of these results).

3.6.2. Improved sub-segment assignment

Applying the segmentation algorithm to a long-term data set showed that the tracking criterion based on spatial overlaps can lead to a clustering of cyclone features into a small number of very long living 4D segments. The reason for this clustering are often connections induced by small features that split off from large cyclones and persist for several time steps, in which they may interact with additional small features, until they finally merge into other large cyclones. While the detected connections can show interesting relations between these 3D features in many cases, for the analysis of the lifespans of single cyclones a more detailed clustering is required. In case of the jet stream segmentation, we experienced similar problems. Our solution was the introduction of sub-segments as presented in Sect. 3.4.3. These sub-segments enabled us to perform a statistical analysis of the lifespan and stability of the jet streams.

Unfortunately, the sub-segment detection method for the jet streams cannot be directly carried over to the cyclone segmentation. A typical event in the development of a strong cyclone is a merging between an upper-level and a lower-level feature. Such events should be preserved, while minor events should not be considered as part of the corresponding sub-segments. However, the sub-segment detection introduced for the jet streams creates sub-segments without any merging and splitting events. This was the reason behind the development of the three additional methods of sub-segment detection presented in Sect. 3.5.2.

The first two additional methods are capable of preserving some of the merging and splitting

events of the features, but not all indirect connections between distinct large cyclones are discarded. The third additional method solves this problem by grouping the features into two classes: connected sets of major features and simple chains of minor features. This technique allows us to uniquely identify distinct cyclone episodes containing only a single connected set of major 3D features. The created sub-segments proved to be reasonable in our particular case studies, as well as in additional test-cases where we singled out specific events that should be preserved, for example a significant merging between an upper-level and a low-level feature.

3.6.3. Results

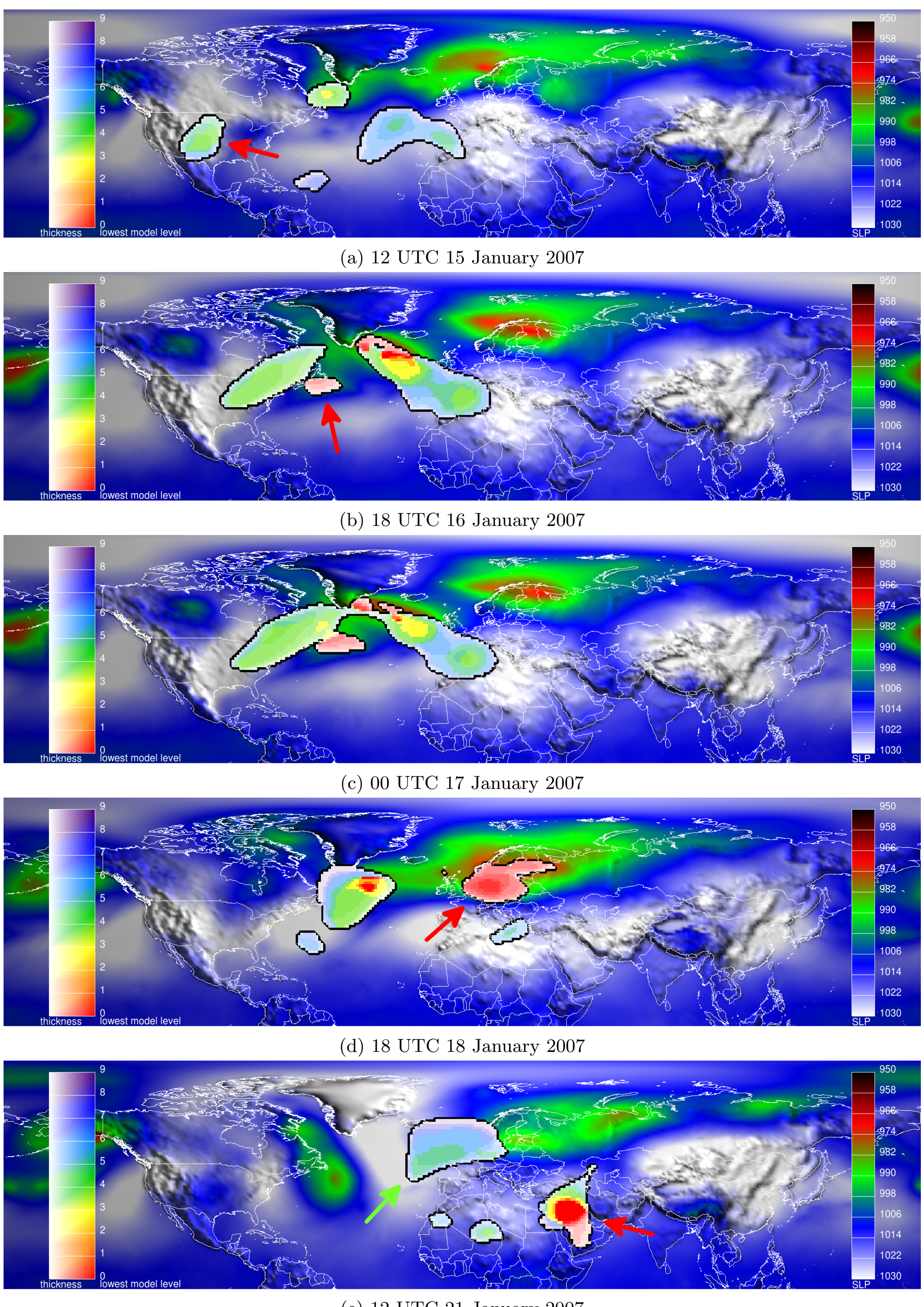
In this section, we present two early results of our new segmentation technique for the detection and tracking of three-dimensional cyclones. Both results are joint work with Heini Wernli and Lukas Papritz: A brief case study of the storm Kyrill, as well as a five-years climatology of cyclones on different height levels.

In the process of developing our 3D segmentation criterion, we successively applied different variants of our method to data of several different storm episodes. We already achieved decent results even with simple vertical threshold profiles $p(z)$, at the cost of a manual fine-tuning of the deviation factor α, as well as other parameters and extensions of our algorithm. For example, in case of the segmentation of storm Klaus, we additionally enabled the application of dilation, which was introduced in Sect. 3.5.3. The reason for this was the relatively high factor α we had to apply in order to obtain a good representation of the storm. This had the consequence that the distinct 3D features got smaller and the quality of the tracking was reduced. In order to preserve the history of the formation of Klaus, we applied the additional dilation of the detected features.

The introduction of the three-dimensional profile $\tilde{p}(\varphi, z, t_{month})$ and the bound on the profile values Z_{max_anom} reduced the amount of required manual customization. This allowed us to compute the climatology of cyclones presented below. However, it is still an open goal to find a universal criterion without the need of any custom parameters for the detailed analysis of distinct single case studies.

Storm Kyrill

For the setup of the cyclone segmentation of the storm Kyrill, we used the three-dimensional profile $\tilde{p}$ with the parameter settings $\alpha = 4.5$, $\alpha_{GSC} = 12$, and $Z_{max_anom} = -15$. We applied no dilation on the resulting features of this case study. The basis of all input data was the ERA-Interim reanalysis data set from the ECMWF (Dee et al., 2011). Several snapshots depicting the development of the segment containing the storm Kyrill can be found in Fig. 3.11. We visualized the 3D features in form of two-dimensional plots using a special coloring scheme. At each horizontal grid position, we indicate the lowest model level covered by a feature through different colors, and the *thickness* of the feature (i.e., the number of occupied grid cells in the vertical column) via different saturations of the respective colors. For example, a light-red color indicates a vertical column of grid cells in which the feature reaches the lowest model level “0”, but is rather thin, whereas a saturated



(a) 12 UTC 15 January 2007

(b) 18 UTC 16 January 2007

(c) 00 UTC 17 January 2007

(d) 18 UTC 18 January 2007

(e) 12 UTC 21 January 2007

Figure 3.11.: Snapshots of the 3D cyclone segmentation of Kyrill. The features are enclosed by black outlines, the colors in the background indicate pressure at sea level.

red color means both a reaching of the lowest model level and a higher thickness of the feature at the particular horizontal position.

The plots of Figure 3.11 show snapshots of the development, peak impact, and aftermath of the storm Kyrill, involving all considered model levels. In Fig. 3.11a, several small, disconnected high-level features are visible. The high-level trough over North America (indicated by the red arrow) is of special importance for the later formation of Kyrill. This trough propagates eastwards, where it merges with the newly forming low-level cyclone Kyrill. In Fig. 3.11b, the propagated trough, as well as the newly formed feature of Kyrill over Newfoundland (red arrow) are clearly visible and almost have contact. One time step later, depicted in Fig. 3.11c, the trough merges with Kyrill and with a pre-existing Icelandic low. The storm intensifies and the large cyclone splits into the core feature of Kyrill, further moving eastwards, and another North Atlantic cyclone. Kyrill had its peak impact on Germany on the 18th of January 2007 (red arrow in Fig. 3.11d). In the aftermath, the core feature of Kyrill persists for several additional time steps as a still intense cyclone over the Eastern Mediterranean (red arrow in Fig. 3.11e), where it further shrinks and finally vanishes. The prominent upper level trough over North-Western Europe (green arrow) leads to the formation of another major cyclone in the course of the next days (not shown).

This case study demonstrates that the detection and tracking of cyclones as full four-dimensional segments is reasonable. An analysis only at the surface level cannot provide the same information about the interactions with upper-level features, and existing connections between distinct surface-level cyclones through features at higher levels are not considered. A full four-dimensional segmentation, in contrast, leads to interesting new insights into the development and the impact of cyclones at multiple height levels.

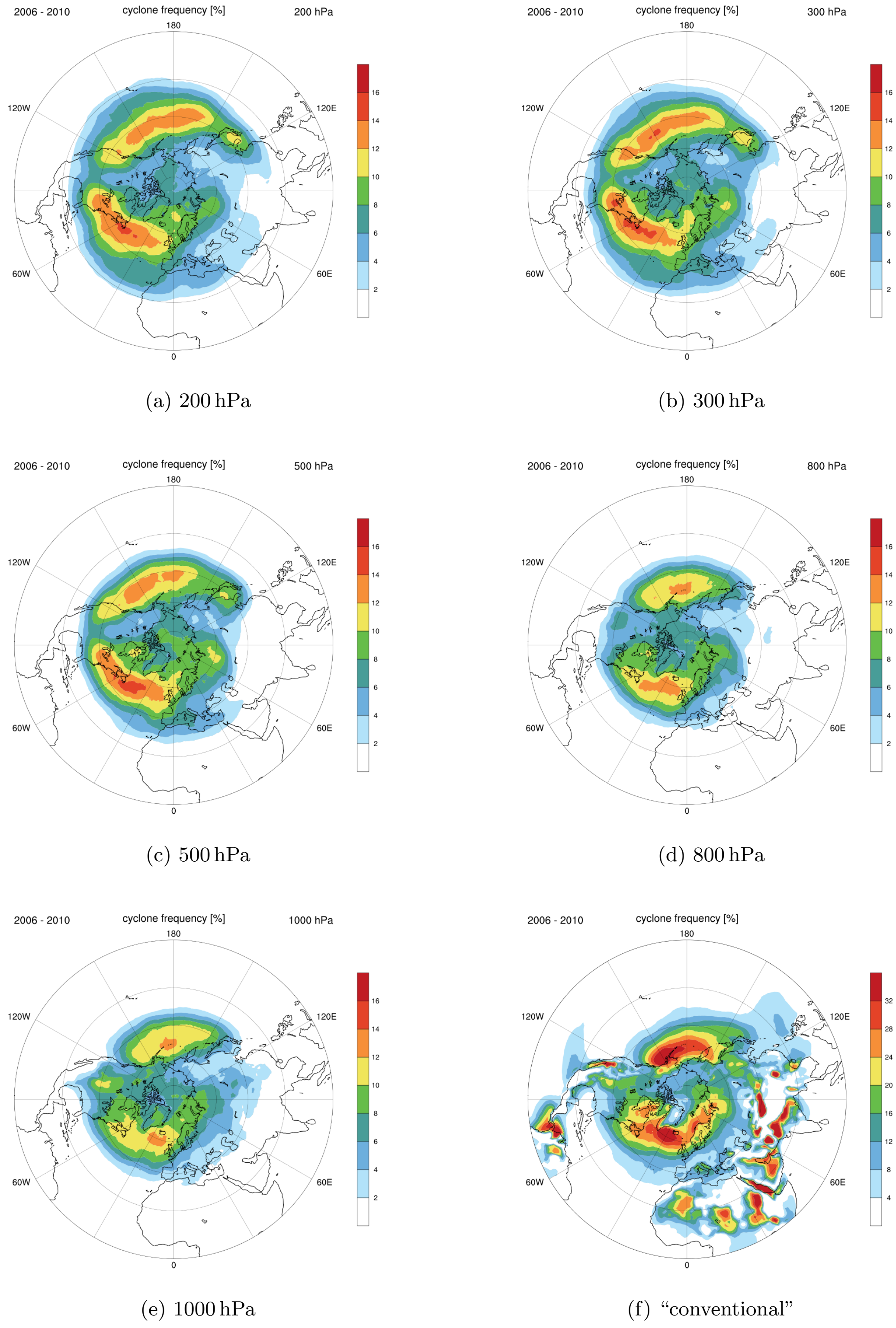
Multi-level cyclone climatology

Finally, we present the results of a five-years climatology of cyclones on different height levels created with our new segmentation algorithm. We briefly discuss the results and compare them to a 2D climatology constructed with the method from Wernli and Schwierz (2006).

For the multi-level climatology, the segmentation algorithm was applied with the three-dimensional profile $\tilde{p}$ and the parameter settings $\alpha = 3.5$, $\alpha_{GSC} = 9$, and $Z_{max_anom} = -20$. The detection in Wernli and Schwierz (2006) is performed by searching the SLP field for local minima and considering the outermost closed SLP contour containing only a single SLP minimum as the area associated with each cyclone.

Figure 3.12 shows selected plots of the results of both methods. In contrast to the ERA-40 input data originally used by Wernli and Schwierz (2006), Papritz applied both methods to the years 2006–2010 of the ERA-Interim data set, for better comparison. In Fig. 3.12a–e, the cyclone frequency of our method on different height levels is shown. Figure 3.12f shows the frequency of the results of the two-dimensional cyclone climatology.

A direct comparison of our new results on the lowest depicted level (1000 hPa) with the conventional cyclone detection method shows consistency in the relative cyclone frequency in regions from 40° N on northwards, whereas the conventional method detects significantly more cyclones in the regions closer to the equator. This discrepancy in the results could



(a) 200 hPa

(b) 300 hPa

(c) 500 hPa

(d) 800 hPa

(e) 1000 hPa

(f) "conventional"

Figure 3.12.: (a)–(e): Climatology of cyclones on different height levels, (f): Conventional, SLP-based cyclones climatology. Images by courtesy of Lukas Papritz (ETH Zurich).

be caused by the detection of heat lows with the conventional method, which are frequent in subtropical regions. These heat lows, however, do not correspond to the extratropical cyclones considered by the new climatology. In general, the method of Wernli and Schwierz (2006) seems to be more sensitive in the detection of cyclones, which could be the reason for the globally increased cyclone frequency. In their method, a local pressure minimum is assumed if a SLP value is minimal in a 3×3 neighborhood of grid cells, which is, according to the authors, a relatively weak criterion leading to a high number of detected cyclones. Taking these differences into account, both climatologies show comparable results in the remaining parts of the plots. Even at higher levels of the atmosphere, the general structure of the climatology is maintained, although the covered area grows and the detected cyclone frequency increases. Further investigation and work is required for a full analysis of these new results.

3.7. Outlook

The results presented in this chapter show that practical applications of the new segmentation algorithm have already led to interesting new insights into the development of different atmospheric phenomena. Nevertheless, we have several ideas for improving both the feature detection and the feature tracking in the future. For improving the detection of 3D features it may be helpful to perform an analysis of the topology of the underlying 3D wind field, similar to the methods of Mahrous et al. (2004). Fuchs et al. (2008) proposed a different method for performing combined extraction and tracking of vortices from time-dependent flow data. More complex feature detection and segmentation techniques, for example spectral clustering techniques as proposed by Ng et al. (2001), also seem promising for feature detection. However, in order to achieve acceptable running times, at least with an unoptimized implementation of spectral clustering, the dimensions of the input data have to be drastically reduced. The detection of the event locations could be improved by applying the same probability-controlled growing technique as used for double-thresholding. For this, a factor proportional to the real distance between the centers of the adjacent grid cells could be used as probability. The localization of the merging and splitting events for non-overlapping features is an additional issue which has to be addressed. For this, we need to allow event locations outside of the actual 3D features, such that we could, for example, use dilated features during the event localization phase.

We already applied and tested our method on smaller and faster moving objects than the jet streams and cyclones, for example ice-water clouds, PV streamers, and warm conveyor belts. These features were given as data sets with a sampling rate comparable to the jet stream data sets. We already implemented an additional dilation of the samples after feature detection, as proposed by van Walsum (1995). However, some additional improvements and extensions of our feature tracking method could be applied in the future. The dilation of selected samples could possibly be refined by taking into account additional knowledge about the expected direction of movement of the features. We could additionally integrate more involved techniques into our approach that make use of the features' attributes, as proposed for example by Reinders (2001), or that rely on prediction-correction methods as used by Muelder and Ma (2009). Another possibility for improving the tracking could involve the assimilation of additional data with a higher temporal resolution, for example satellite data, in order to predict the development of the features between two original time steps.

3.8. Conclusions

In this chapter, we presented a new segmentation method with per-grid-point localization of merging and splitting events. The method is capable of efficiently detecting three-dimensional atmospheric features and their development over time. We adopted ideas from several methods from the field of flow visualizations, as described in Sect. 3.1. However, our method is unique in that it is capable of estimating the locations of merging and splitting events in grid space. In addition, we could simplify the implementation to the point that we only need a single iteration over the data set. By selecting and clustering the data during this iteration, we avoid the construction of additional spatial data structures, as for example done by Silver and Zabusky (1993). Muelder and Ma (2009) track single features very efficiently by operating on the boundary of the feature only. However, if new features (or holes inside existing features) should be detected, a full iteration over the data set (except for the already detected boundaries) is unavoidable.

We showed in a case study how the detection of event locations can be relevant in studies of atmospheric dynamics. We applied our method to analyze upper-tropospheric jet streams in the Northern Hemisphere and their stability in different regions. A method for a more detailed analysis of the event graphs of the segments was presented. The resulting sub-segments were the basis for the stability analysis of the jet streams.

Further extensions of the basic algorithm were presented, including a double thresholding approach for avoiding under-segmentation, a dilation technique for preventing over-segmentation, as well as additional sub-segment detection techniques for an improved analysis of the resulting event graphs of the segmentation. The motivation behind these extensions came from additional applications of our algorithm for the detection and tracking of further atmospheric phenomena.

At the end, we presented some first results of the detection and tracking of three-dimensional cyclones. For this application of our algorithm, more complex selection criteria were formulated and the new sub-segment detection techniques, as well as dilation was applied.

So far our experience with the new segmentation method in terms of computational costs is very positive. The analysis of the two-years wind field data set used in the jet streams study was performed with an early version of our software in about five hours on a standard Linux PC. The computation of the Kyrill case study, containing 44 time steps covering a period of eleven days, took about 30 seconds on a standard Linux laptop. Due to the simple structure of the algorithm consisting mainly of one iteration over the data set, and due to the way the input data is accessed, the algorithm seems well-suited for a future extension towards parallel processing. All this indicates that the method is also capable of analyzing very large data sets, for instance output of long-term climate simulations and/or from ensemble simulations. This will offer novel pathways for an in-depth analysis of flow features in very large climate data sets.

4. Interactive Weather Analysis Laboratory (IWAL)

4.1. Introduction

The goal of the IWAL (Interactive Weather Analysis Laboratory) software project was the development of a tool for an easy, fast, and interactive access to large meteorological data sets. The main target audience are students with no or just little experience in the handling and visualization of such data. The provided interactivity and advanced features, such as the interactive computation of trajectories, can also be of interest for a wider, more experienced target audience.

Students of atmospheric sciences commonly face a set of problems in producing appropriate plots during their first semesters of study. One main problem is the amount of knowledge about infrastructure and software tools that is necessary for the creation of the visualizations they are interested in. So instead of spending the majority of their available time thinking about the weather phenomena they are originally interested in, some students end up dealing with unknown scripting languages, unintuitive software tools, and with learning to handle a Linux shell. These problems were the main motivation behind the development of IWAL. However, the IWAL software tool is not intended to be a replacement of existing visualization tools, but to provide the possibilities to get a first impression and an initial understanding of the data sets, to allow interactive searching for interesting data that may later be visualized by means of other tools, and to share these discoveries with other students or teachers in an easy and convenient way.

IWAL was planned from the beginning as a web application (see Fig. 4.1 for an example screenshot). This way, a lot of pitfalls for the users are avoided. As a web application, the software is platform independent, no installation is required, and the input data does not have to be directly accessible on the local machine. The acquisition of new data, for example up-to-date forecast data, can be organized at a central place, the IWAL server. However, a web application also has a lot of drawbacks in comparison to native applications. Examples are the increased effort that has to be put into securing the application against attacks, and the limits on the execution speed imposed by the available band-widths on both the user side, as well as on the server-side. One permanent challenge during the development of IWAL was to find a working balance between execution speed and available visualization options.

The design and development of IWAL was, of course, influenced by Insight. Both software tools have a lot in common, for example some visualization techniques and the handling of properties as individual objects. The main differences are the lack of three-dimensional visualization and the handling and caching of plots in form of image tiles in IWAL. As it is the case for Insight, IWAL is implemented in a modular fashion. Additional modules for the

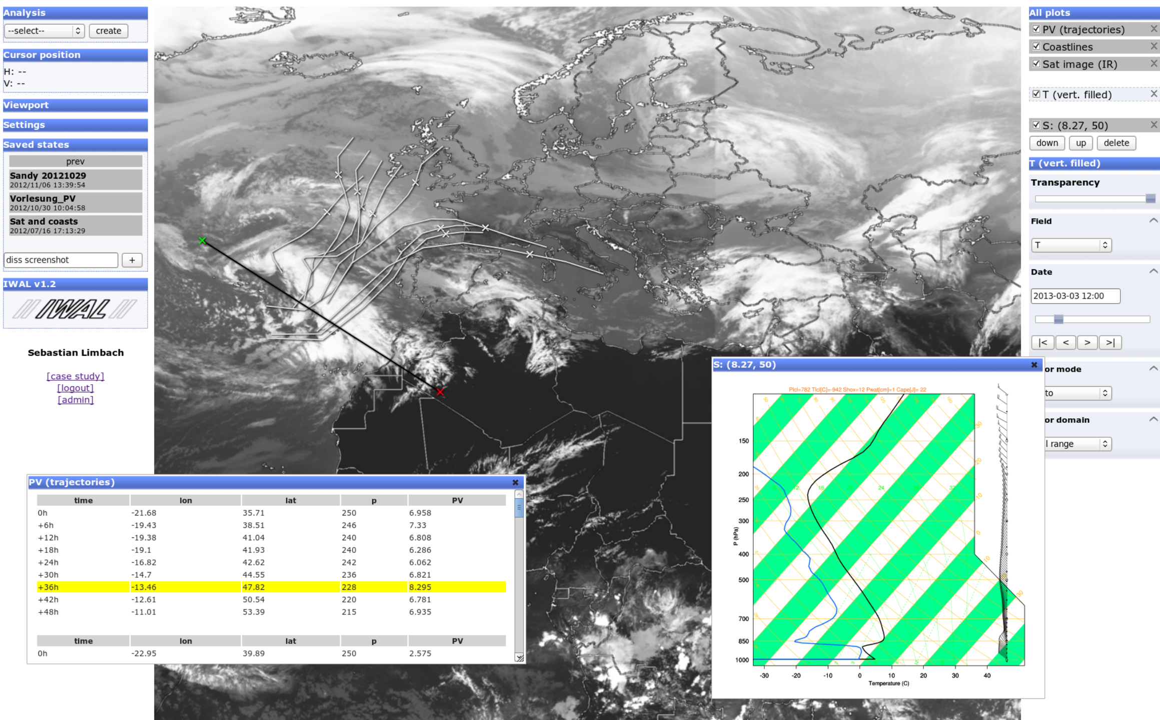


Figure 4.1.: Screenshot of a typical IWAL session. The image shows the user interface, a satellite image of Europe and Africa, trajectories and PV data traced along the trajectories, as well as a skew-T/log-P-plot.

creation of new plots or diagrams, as well as new algorithms for the computation of derived data sets (e.g. trajectories) can easily be added to the existing IWAL infrastructure.

The IWAL software project was initiated at the ETH Zurich by Michael Sprenger and Heini Wernli in 2010. It was funded as a "Innovedum" project. Innovedum is a fund for innovative projects which aim at new methods of learning and the improvement of teaching[1]. The implementation officially started in October 2011 and lasted until end of September 2012. IWAL contains several Python scripts for the creation of two-dimensional plots that are contributions by Michael Sprenger.

4.1.1. Outline

In this introduction, a general overview of the ideas and concepts behind the IWAL web application is given. In addition, we introduce related software projects that influenced the development of IWAL, directly and indirectly.

The following sections focus more on the technical realization of IWAL. In Sect. 4.2, all third-party libraries and tools that were used for the creation and the execution of IWAL are introduced. Thereafter, the software architecture and technical realization of the IWAL server is presented in Sect. 4.3, followed by a detailed description of the implementation of the IWAL client in Sect. 4.4. Section 4.5 summarizes the achieved goals of IWAL by

[1] cf. `http://www.innovedum.ethz.ch`, last accessed: 19-September-2012

presenting the results of a series of benchmarks, as well as the results of a user survey that was carried out with the first students using IWAL in class. Finally, Sect. 4.6 contains an outlook on possible future features and enhancements of the IWAL software.

4.1.2. Features

This section focuses on the most important features of the IWAL software. Some of these features are project goals that were set prior to the start of the implementation. Other features evolved out of ideas or needs which came up during the development process. The first group of students which worked with IWAL during the spring semester of 2012 also provided valuable input and ideas for features and enhancements that are now part of IWAL.

Below, the most important features of IWAL are presented in order to provide an overview of the capabilities of the final software.

Interactive data visualization

The core feature of IWAL is the interactive visualization of atmospheric data. Different types of visualizations are realized in form of distinct IWAL *modules*, from which the user can choose. Each module provides a set of plot properties such as the time step, the variable, the height, and the color bar for the visualization. The user selects the desired parameters for these properties and finally invokes the creation of a corresponding *plot*.

The two concepts of *modules* and *plots* are closely related. An IWAL module is the representation of a certain plot type and its available options. For each type of visualization, exactly one module is responsible. A plot, on the other hand, represents a concrete set of options for the properties of one module and the corresponding plot data. For each module, a set of multiple plots can exist. The user is capable of changing the properties and the covered region of the plots in order interactively explore the sets of atmospheric data. For the visualization on the client-side, each plot contains some additional data (e.g. an individual name, a transparency value, and a visibility flag).

Viewports

IWAL provides different areas (*viewports*) for the visualization of the data. Each module is associated with one of these viewports, all plots produced on basis of a module are typically visualized in the corresponding viewport area. The available viewports are:

- the *horizontal* viewport, providing a shared area for displaying plots using a cylindrical projected coordinate system,
- the *vertical* viewport, providing a shared area for plots lying on a common vertical cross-section with a logarithmically scaled z-coordinate,
- the *separate* viewport, providing individual non-modal dialogs for each plot, such that each plot can display the information in its own reference frame.

The vertical and separate viewport are only capable of visualizing plot data that has the form of raster graphics images. These raster images are generated by the modules on the

server-side. The horizontal viewport offers the additional possibility to visualize the plot data individually on the client-side. For this, the corresponding plots, which are called *data plots*, store the data values for the visualization internally and provide a drawing function that is called in order to visualize this data. These plots are not limited in their drawing options and are capable of displaying data or information outside of the actual viewport area. Currently, only the trajectory module creates data plots. The drawing code uses JavaScript's canvas functions for drawing the actual trajectories. In addition, it manages a separate view on the trajectory data in form of a table shown in its own, non-modal dialog window.

Image export

IWAL can export the content of the horizontal and vertical viewport in form of image files. The file format offered by IWAL is PNG (portable network graphics)[2].

Trajectory calculation

As mentioned above, IWAL offers a module for the calculation and display of trajectory data. This module differs from the other IWAL modules in that it does not create raster graphics images out of the existing data sets. Instead, the trajectory data is transmitted in JSON-encoded form (cf. Sect. 4.2.6) from the server to the plot instance of the client. Internally, *Lagranto* (Lagrangian analysis tool) is used for the creation of the trajectory data (Wernli and Davies, 1997). The client visualizes the data as line strips by using corresponding canvas drawing functions.

Different user modes

One drawback of the client/server architecture of IWAL is the amount of workload the server may experience whenever multiple clients try to create many plots at the same time. Such a scenario is very likely, since IWAL is intended to be used primarily in class, with multiple students and teachers accessing it at the same time. The calculation of the image of a plot can take a considerably amount of time, and early benchmarks showed that the plot images need to be cached in order to handle a reasonable number of queries. Such caching, however, is only feasible when the options an IWAL module offers to the user are heavily restricted. For details on these restrictions and for a discussion on the performance of IWAL, see Sect. 4.3.2 and Sect. 4.5.1.

To give teachers or PhD students the chance to work with IWAL in a more flexible way, IWAL offers two different user modes. One mode is for students, and the other is intended for privileged users, giving them more plot options. Internally, the privileged user mode is realized by offering different modules to the privileged users, having a different set of properties with generally less restrictions. In addition, some restrictions on the viewport properties are lifted as well, for example the maximum allowed zoom level of the horizontal viewport.

[2] see `http://www.libpng.org/pub/png/`, last accessed: 03-January-2013

User, courses, and case studies

In IWAL, the individual set of modules available to a user is determined in the following way: Each *user* is enrolled to a set of *courses* which determine the set of available *case studies* from which the user can select an active one. After having selected a case study and the desired user mode (basic or privileged), the user can create plots from the set of available basic or advanced *modules* provided by the case study.

In IWAL, sets of available modules for the creation of plots are grouped into different *case studies*. Each case study offers a basic set of modules, as well as an optional set of advanced modules that replace the basic set of modules in the privileged user mode. In addition, each case study determines the location on disk and the range of the input data that can be used by the modules.

Case studies themselves are grouped into different *courses*. Each course consists of a name, a description, and the set of available case studies. In addition, a course can offer a password that can be used by a *user* to enroll into the course. All case studies from all enrolled courses are available to the user. A new user can be registered through the IWAL web interface. Initially, the new user is not associated with any course.

IWAL associates additional data with each user for controlling the user's privileges. This data contains, for example, a flag indicating whether the user represents a single person (*single user*) or a group of students (*group user*). In addition, the information about whether or not a user is allowed to enter the privileged user mode and the administration interface of IWAL is stored.

Storing and restoring of program states

Each *single user* can save the current program state (the current case study and the set of plots, as well as the viewport settings) using an individual name. The saved user state is visible and can be restored by any user with the same active case study. This way, teachers and individual students can share interesting plots. The stored state can be removed again by the same user who created it, or by any user with access to the administration interface.

Administration interface

IWAL provides an administration web interface for direct manipulation of the database background. This administration interface is provided directly by the Django framework (see Sect. 4.2.1) used by IWAL. The interface can be used to perform tasks such as the deletion of users, and the creation and configuration of new modules, case studies, and courses.

4.1.3. Related work

The development of IWAL was influenced by several existing programs and web applications, and uses internally a diverse set of multiple technologies, protocols, and software tools. In

this section, related software projects are presented. An overview of the internally used technologies for the development and installation of IWAL can be found in Sect. 4.2.

The general idea of a tile-based plot visualization and the basic mechanics used for the translation and zooming of the viewports can be found in many existing interactive web applications for the display of maps, foremost in Google Maps[3]. A similar tile-based maps visualization is provided by the standalone 3D application Google Earth[4]. In addition, Google Earth works as a WMS (Web Map Service, cf. de la Beaujardiere, 2002, 2006) client. IWAL itself provides most of its plots through the WMS protocol, implementing a full WMS server (see Sect. 4.2.7), such that Google Earth could be utilized as an early testing environment for the first IWAL visualizations.

The basic idea of using WMS came from the flight planning tool "Mission Support System" (MSS) by Marc Rautenhaus (TUM), see Rautenhaus et al. (2012). Several other WMS client tools exist, see for example the list at `http://www.ogcnetwork.net/node/175` (last accessed: 03-January-2012). Some clients use WMS even in nonstandard ways in order to produce special plots and visualizations, for example vertical cross-section plots or time series. One example tool is IBL Visual Weather[5]. One of its developers proposes in Matula (2009) some ways to utilize WMS for the creation of time series and skew-t/log-p-plots. The European Centre for Medium-Range Weather Forecasts (ECMWF)[6] is currently developing several different web services and applications under the project name *ecCharts* as a result from their "Web Re-engineering Project" (WREP)[7]. One of these applications, ecCharts/-forecaster provides a web-based interface to interactive visualizations of forecast data (Raoult et al., 2011).

IWAL provides a pure web-based WMS client front end for different ways of visualizing meteorological data. In contrast to other WMS front ends presented here, IWAL provides a web-based user interface for the manipulation of the parameters of the visualized maps. In addition, it provides the infrastructure for the user-management, for setting up different courses and case studies, and for the exchange of interesting plot settings between users, as presented in the previous parts of this section. In contrast to presented web applications which are dealing mostly with two-dimensional data sets, the data basis for most of the plots of IWAL are four-dimensional. Furthermore, the IWAL plots offer a number of user options. These additional options and dimensions made the development of new strategies for the precomputation and the caching of the plots necessary in order to provide an interactive experience to a reasonable number of simultaneous users.

4.2. Technical building blocks

The IWAL implementation uses several software tools, libraries, and protocols, both on the server and the client-side. Already at an early planning stage, we decided, together with our colleagues at the ETH Zurich, to use the Python programming language with the PyNGL and PyNIO libraries for the creation of the data plots. Based on this decision,

[3] `https://maps.google.com/`, last accessed: 03-January-2013
[4] `http://www.google.com/earth/index.html`, last accessed: 18-June-2013
[5] `http://www.iblsoft.com/products/visualweather`, last accessed: 04-January-2013
[6] `http://www.ecmwf.int/`, last accessed: 03-January-2012
[7] `http://www.ecmwf.int/eccharts`, last accessed: 20-September-2012

we evaluated several different Python web frameworks (cf. http://wiki.python.org/moin/WebFrameworks, last accessed: 11-June-2013) and finally decided to use Django, which offers all required features, is well documented, successfully used by many different websites, and which we were already familiar with. As a web application, IWAL depends on additional software running on the server, namely a database, a caching back end, and the web server itself. We use standard software and technologies for these tasks that have already proven their suitability in the context of numerous successful software projects. In this section, the most important third-party components of IWAL are presented in detail.

4.2.1. Django and South

On the server-side, IWAL is implemented in form of *Django*[8] (see, for example, Holovaty and Kaplan-Moss, 2009) applications. Django is a Python-based, high-level web framework whose development is supported by the non-profit Django Software Foundation[9]. Django consists of several components supporting multiple different aspects of the development of a web application. The provided features range from an object-relational mapper for the description and manipulation of the objects to be stored in the database, support of user management, the provision of an administration interface, session management, support of different caching solutions, an HTML-template engine, up to security features for the prevention of cross-side scripting and other forms of attacks. IWAL was developed and deployed using Django version 1.2.3.

IWAL consists of two cooperating Django applications. The `wms` application contains the implementation of the WMS server, including all IWAL modules. The `wmsclient` application contains the server-side implementation of the WMS client (database access, session management, provision of the HTML5 pages through Django's template engine).

For the management of migrations of the database schemas during the development, we used *South*[10]. South is a software tool specifically designed for Django projects, since Django itself offers no support for schema migrations.

4.2.2. PostgreSQL

Django offers support for multiple different databases (PostgreSQL, MySQL, Oracle, and SQLite). For IWAL, we use the object-relational *PostgreSQL*[11] database for storing information about the users, courses, case studies, program states, as well as the available modules. An overview of the database design can be found in Sect. 4.3.4.

PostgreSQL is open source software developed by the PostgreSQL Global Development Group, with contributions from developers coming from different countries and enterprises[12]. It originated from the POSTGRES project, whose development started in 1986 at the UC

[8] https://www.djangoproject.com/, last accessed: 03-January-2013
[9] https://www.djangoproject.com/foundation/, last accessed: 14-March-2013
[10] http://south.aeracode.org/, last accessed: 04-January-2013
[11] http://www.postgresql.org/, last accessed: 04-January-2013
[12] see http://www.postgresql.org/community/contributors/, last accessed: 04-January-2013, for a list of contributors

Berkeley (Stonebraker and Rowe, 1986). Later on it was extended with an SQL interpreter and renamed into Postgres95 in 1994, and finally to PostgreSQL in 1996[13].

4.2.3. PyNGL and PyNIO

For the input and visualization of data stored in netCDF files (cf. Sect. 2.4.3), two Python packages for scientific visualization are used: *PyNGL* and *PyNIO*[14]. These packages are developed by the Computational and Information Systems Laboratory (CISL) at NCAR (National Center for Atmospheric Research)[15]. The NCAR is a program of the National Science Foundation (NSF) and is managed by UCAR, the University Cooperation for Atmospheric Research, a cooperation of 70 North American universities[16].

Both PyNIO and PyNGL provide a Python interface to *NCL*, the NCAR Command Language[17]. PyNIO covers file input/output operations and is used by many IWAL modules for reading in netCDF files. PyNGL provides various functions for the creation of two-dimensional scientific visualizations, it is used in IWAL for creating contour plots, solid, color-filled plots, as well as for the creation of skew-t/log-p diagrams.

4.2.4. Apache and mod_wsgi

For the deployment of the Django application, we use an *Apache web server*[18] running *mod_wsgi*[19]. The development of the Apache web server is carried out by the Apache Software Foundation[20]. It is an open-source HTTP server that is, according to the project homepage, "the most popular web server on the Internet since April 1996".

The basic functionality of the server can be extended through modules. Several ways for the deployment of a Django program exist, one of them is through the Apache module mod_wsgi. *WSGI*[21], the Web Server Gateway Interface, is an interface for the communication between web servers and Python applications. The mod_wsgi module is developed by Graham Dumpleton as open source software for the execution of WSGI-based Python programs on an Apache web server.

The main server running IWAL at the ETH Zurich can be reached at `http://iwal.ethz.ch`. For development, we used the local web server provided by Django, as well as a second, non-public Apache web server.

[13] cf. `http://www.postgresql.org/docs/9.2/static/history.html`, last accessed: 04-January-2013
[14] `http://www.pyngl.ucar.edu/index.shtml`, last accessed: 04-January-2013
[15] `http://ncar.ucar.edu/about-ncar`, last accessed: 04-January-2013
[16] cf. `http://www.ucar.edu/governance/members/institutions.shtml`, last accessed: 30-August-2012
[17] see `http://www.ncl.ucar.edu/`, last accessed: 04-January-2013
[18] `http://httpd.apache.org/`, last accessed: 04-January-2013
[19] `http://code.google.com/p/modwsgi/`, last accessed: 04-January-2013
[20] `http://www.apache.org/`, last accessed: 04-January-2013
[21] see `http://wsgi.readthedocs.org/`, last accessed: 14-March-2013

4.2.5. HTML5, JavaScript, and JQuery

IWAL is a web application that runs on the client-side in a web browser, for example Firefox or Chrome. It uses standard *HTML5* with embedded *JavaScript* for the provision of the web page content and all its interactive elements. The HTML5 pages are generated on the server by the corresponding Python code of the `wmsclient` Django application. Besides database access and session management, the application makes use of Django's template engine for the creation of the HTML5-code containing all required settings.

HTML5 (HTML stands for *hypertext markup language*, the markup-language of the world wide web) is being developed by the WHATWG[22] (Web Hypertext Application Technology Working Group) and the W3C[23] (World Wide Web Consortium). It is aimed to be completed and recommended by 2014[24]. The original HTML was developed by Tim Berners-Lee at CERN, see Berners-Lee and Connolly (1993). HTML5 contains many enhancements compared to its predecessor, HTML4, for example the introduction of the new elements `<audio>`, `<video>`, and `<canvas>` for the provision of multimedia content[25].

JavaScript is a scripting language developed in 1995 by Brendan Eich of Netscape (cf. Severance, 2012). It was originally named LiveScript, and it was intended to be a simple and lightweight language for being embedded into webpages. JavaScript is used by IWAL for the creation of most of the interactive elements of the web application. We use the *JQuery*[26] JavaScript library for an easier manipulation of the DOM (document object model) of the page, and *JQueryUI*[27] for the rendering of some advanced user interface elements (dialog windows and sliders). The display of the content of the viewports, either in form of bitmap tiles or in form of vector graphics created on the client-side, is realized via HTML5 canvas elements and corresponding JavaScript visualization code.

4.2.6. JSON

JSON[28] (the JavaScript Object Notation) is a compact, language independent format for the description and exchange of simple data objects. The term "JSON" was coined by Douglas Crockford, who acquired the domain `json.org` in 2002[29]. JSON is based on the syntax of object literals in JavaScript. The standard is described in RFC 4627[30].

Libraries for the transformation of data objects into a JSON representation and vice versa are available for many different programming languages. In IWAL, JSON is used in the Python code on the server-side through the `simplejson` library, which is the *"externally maintained version of the `json` library contained in Python 2.6"*[31]. On the client-side, the conversion of JavaScript objects into JSON objects is realized directly through the `JSON.stringify()`

[22] `http://www.whatwg.org/`, last accessed: 07-January-2013
[23] `http://www.w3.org/` and `http://www.w3.org/html/`, last accessed: 07-January-2013
[24] see `http://www.w3.org/html/wiki/FAQs#When_will_HTML5_be_done.3F`, last accessed: 07-January-2013
[25] cf. `http://dev.w3.org/html5/html4-differences/`, last accessed: 07-January-2013
[26] `http://jquery.com/`, last accessed: 14-March-2013
[27] `http://jqueryui.com/`, last accessed: 14-March-2013
[28] `http://www.json.org/`, last accessed: 10-January-2013
[29] see `http://www.json.org/fatfree.html`, last accessed: 27-February-2013
[30] `http://tools.ietf.org/html/rfc4627`, last accessed: 27-February-2013
[31] `http://simplejson.readthedocs.org`, last accessed: 28-February-2013

JavaScript function. For the other way around, either the `JSON.parse()` or the `eval()` function can be used.

4.2.7. WMS

The Web Map Service (WMS) is a protocol for the exchange of two-dimensional, georeferenced maps (de la Beaujardiere, 2006). A "map" is a visual representation of the georeferenced data, not the data itself. The protocol was developed by the Open Geospatial Consortium[32] (OGC). WMS describes a set of operations in form of HTTP GET-requests with a fixed set of parameters[33]. Early IWAL versions were based upon WMS version 1.1.1 (de la Beaujardiere, 2002) (which was required for testing with Google Earth), the current IWAL version supports version 1.3.0 of the protocol.

In de la Beaujardiere (2002), three operations for the server are defined:

> ***GetCapabilities*** *(required): Obtain service-level metadata, which is a machine-readable (and human-readable) description of the WMS's information content and acceptable request parameters.*
>
> ***GetMap*** *(required): Obtain a map image whose geospatial and dimensional parameters are well-defined.*
>
> ***GetFeatureInfo*** *(optional): Ask for information about particular features shown on a map.*

IWAL supports the two mandatory commands *GetCapabilities* and *GetMap*. In Sect. 4.3.2, the implementation of IWAL's WMS server is presented, containing a detailed description of the implementation of these two WMS operations. In addition, we discuss the extension of the WMS operations by our own operation *GetData*, which is used for requesting JSON-encoded data (e.g. the trajectories data) from the corresponding IWAL module. For the definition of the location of the vertical cross section we also use a nonstandard extension of the GET-parameters of the WMS request.

4.3. Server architecture

This section covers the server-side implementation if IWAL. It contains a discussion of the two Django applications `wms` and `wmsclient`. The first one provides the server-side source code of all IWAL modules, each of which can be accessed through WMS requests. In response to a WMS request, plot data is send back to the requester, which is either produced by the corresponding module or retrieved from the cache. The data itself is typically a map image, or a JSON-encoded data set in case of the trajectories module.

The `wmsclient` application is responsible for the provision of all web pages the IWAL client consists of. Most of these web pages are created dynamically by the server. For this, the `wmsclient` application uses Django's template engine and the object-relational mapper for

[32] `http://www.opengeospatial.org`, last accessed: 07-January-2013
[33] Version 1.3.0 of the protocol also supports POST-requests.

the creation of the corresponding HTML5 and JavaScript code. The application is also responsible for handling the user privileges and for the session management.

Besides the implementation details of these two applications, additional topics covered in this section are the extension of the WMS specification for querying vertical plots and data plots, the implemented caching strategies for the provision of the plot data, and the design of the database of IWAL.

4.3.1. Django basics

Each Django project is divided into a set of applications. An application is a closed unit containing several standardized Python files. These files define, for example, the database models of the application (in the file `models.py`), a set of automated tests (`tests.py`), and most importantly a set of views (`views.py`). A view is a function which is called in response to a specific HTTP-request of a user. The function is responsible for the creation and the returning of the corresponding HTTP-response (e.g. an HTML document or an image file).

For the mapping of HTTP-requests to the specific views of the applications, the global project file `urls.py` is used. It contains a set of tuples for the mapping of regular expressions describing the format of an URL to the corresponding view-functions. In addition, the project file `settings.py` contains variables with information about global settings, such as the database connection, the caching back end to be used, the Django applications to be included into the project, directory paths for the serving of static files, as well as some IWAL specific settings that were added as part of our own implementation.

4.3.2. The `wms` application

The `wms` application is responsible for the handling of all IWAL modules and for the provision of all WMS operations. The modules are realized through the `wms.modules.IWALModule` class, the WMS operations are implemented in form of a single Django view, namely `wms.views.wms_request`.

The `wms` applications contains no database models itself. In the `wms_request` view, however, it queries the database for information about the respective user profile and the selected case study through the database models of the `wmsclient` application.

The `IWALModule` class

At the core of the `wms` application is the `IWALModule` Python class. This class is the base class for all individual IWAL module implementations. Its interface provides access to meta information, the properties, and to the functions for the production of the plot data of an IWAL module. Each module is implemented as a subclass of `IWALModule`, some modules are represented by instances of the same subclass with differing internal settings.

Each `IWALModule` object stores at least the following information:

- its unique name as a string,

- the WMS server name which is part of the URL for the corresponding WMS query,
- the names of the available WMS layers (currently, only two layers are supported by IWAL, one for the plot data, and one for the legend of the plot),
- an array of `Property` objects representing the different plot properties,
- the type of the associated viewport,
- a flag indicating whether or not the plot data is intended to be cached,
- and additional information about the input data (see the section below).

The interface of the `IWALModule` class offers functions for:

- the update of all properties from the parameters of a given HTTP request (`update_properties_from_http_request`),
- the setup of the input data of a given case study (`set_case_study_input_data`),
- the serialization of the meta information (`server_infos`-property) and the module's plot properties (`properties`-property) via JSON, in order to be able to pass this information on to the JavaScript code of the client (see Sect. 4.2.6),
- the actual plotting of the data (`create_image`) and the legend (`create_legend`),
- as well as an optional function for the computation of the data of any data module (`get_data`).

All functionality for the creation of plot data is contained in subclasses of the `IWALModule` class. Therefore, the set of implemented subclasses defines the range of plot types, possible visualization and data processing techniques available through IWAL. If IWAL is to be extended by a new type of plot, a new subclass of `IWALModule` needs to be implemented, or some existing subclass needs to be adapted. For the first applications of IWAL, a variety of modules is already available in IWAL. Table 4.1 provides an overview of all currently implemented subclasses of `IWALModule`.

Input data handling

On startup of the `wms` application, two sets of `IWALModule` objects are created. One set represents the default, initial settings and properties of each IWAL module per case study. The other set contains `IWALModule` objects for the actual plotting. The modules of this latter set get modified by the application whenever new plot data needs to be produced. The application is responsible for updating the information about the input data, if necessary, and for setting all properties of the corresponding module to the requested plot options.

The reason for the instantiation of individual `IWALModule` objects for every case study is that the available plot options and the input data may differ significantly for different case studies. One main motivation behind the introduction of case studies in IWAL was to give the user a way of focusing on a limited time slice of all available input data. In addition, each case study defines its own set of input variables (or fields) which also allows for focussing on the relevant variables of interest.

Coastlines	This module generates images of the coastlines and country borders of the earth.
HorizontalCrossSection **HorizontalCrossSectionEnhanced**	These modules provide horizontal cross sections through data sets at fixed pressure levels. A coloring scheme is applied in order to map data values from different intervals to different colors. The standard version provides fixed coloring schemes depending on the selected variable and the height of the horizontal cross section. The enhanced version is intended for the privileged user mode, since it allows for a free selection of the height of the cross section, as well as for individual configuration of the coloring scheme.
HorizontalCrossSectionContour	This module also provides a horizontal cross section, but visualizes the data in form of closed contour lines.
SatImage	Provides a satellite image of the earth's surface. Two channels are available, namely infrared and water vapor.
SkewTLogP	This module produces skew-T/log-P diagrams of any single point on earth. The resulting diagrams are visualized by the IWAL WMS client in individual non-modal dialog windows.
Trajectories	This module computes trajectories and traces an arbitrary set of variables along these trajectories. A JSON-representation of the trajectory data is returned, which is then visualized in form of line strips and in tabular form by the IWAL WMS client.
VerticalCrossSection **VerticalCrossSectionEnhanced**	These two modules provide vertical cross sections of a data set, where the data values are colored as it is the case for the **HorizontalCrossSection** modules. The enhanced version provides a different set of properties for the manual selection of the parameters of the coloring scheme.
VerticalCrossSectionContour	This module produces the same plots as the **VerticalCrossSection** module, but uses contour lines instead of solid filled areas for the display of the data.

Table 4.1.: The available subclasses of `IWALModule`.

Basically, each IWAL module has completely freedom on how it handles its own input data. However, some automated methods are provided by IWAL in order to simplify the input data handling. Each IWAL module can define a set of variable selection properties and time selection properties which are automatically adapted by the `set_case_study_input_data` function, based on the settings of the case study from the database. Additionally, the two private member variables `_data_dir` and `_sat_image_dir` are updated accordingly.

The following configurations for the automated update of the input data information are supported by IWAL:

1. In the most simple case, the case study provides a single directory where all input data can be found, together with a start date, a total number of time steps, and the time difference in hours between single time steps.
2. Some case studies operate with automatically updated sets of input data (e.g. recent analysis data) which is updated externally whenever new data is available. For these cases, IWAL allows the definition of a wildcard-expression for obtaining all input files from a given input directory.
3. Other case studies work on different sets of input data that may overlap in time, for example forecast data produced at different days. For this, a comma-separated list of subdirectories can be specified. IWAL automatically adds a selection property with the name "Data set" to the module, from which the user can select the data set to be visualized.

The `Property` class

The design of the `wms.modules.properties.Property` class of the `wms` application is closely related to the `Property` classes of Insight (see Sect. 2.5.5). Each `IWALModule` object maintains a list of `Property` objects. These properties represent the plot options of the module, for example the time step or the variable to be visualized. Not included in a module's list of properties are the viewport-related plot options, for example the visible region of the horizontal viewport. These settings are handled separately and are directly passed to the module's corresponding member function for the production of the plot data.

Each property has a name, a type (identified through a string), and the following three flags:

- The `fixed` flag indicates properties whose value is set initially by the user, but cannot be changed once a plot is created.
- The `visible` flag indicates whether or not the property should be initially shown by the client in the list of all properties of the module or a plot.
- The `client_side` flag is set for properties whose value does not affect the computations on the server-side. Because of this, no update of the plots is required whenever the value of such a property changes. The only current example of such a property is the "Highlight timestep" property of the `Trajectories` module, which indicates the point in time for which the positions of the trajectories should be highlighted by the client when they are drawn.

`BoolProperty`	This property represents a boolean value.
`ColorProperty`	This property represents an RGB color value.
`ValueProperty`	This property represents a numerical value together with a set of possible units the user can select from. This property is also used for the representation of dates in form of time steps. A set of member functions is provided for the conversion of the time step numbers into formatted date strings.
`ValueRangeProperty`	This property is similar to the `ValueProperty`, but instead of representing only one value, it represents two values forming an interval.
`PositionProperty`	This property represents a two-dimensional geographic position given as a pair of longitude/latitude coordinates.
`SelectionProperty`	This property represents a set of options from which one option can be selected at a time. The options and the selection are represented by strings.
`VertCrossProperty`	This special property contains the horizontal start and end positions, as well as the type of a vertical cross section. Having an individual property object instead of a combination of a `SelectionProperty` and two `PositionProperty` objects allows for a direct interaction between these properties. For example, if the cross section type is set to linear interpolation in north/south direction, the end position can be constrained to positions on the same longitude as the start position by the client interface.

Table 4.2.: The current subclasses of the `Property` class.

The actual settings represented by each property are defined in corresponding subclasses of the `Property` base class. An overview of all available subclasses can be found in Table 4.2.

The representation of the state of a property needs to be transported both from the server to the client and vice versa. For the first direction, the properties can be serialized using JSON, but for the other direction, we want the properties to be part of the corresponding WMS request. For this, we use the possibility of WMS to add *sample dimensions* to a query. These sample dimensions are realized as a key/value pair that is added to the request's GET-parameters, where all keys have the prefix `DIM_`. The different types of properties use one or multiple sample dimensions for the representation of their particular values. To avoid conflicts, for example if multiple properties of the same type are part of one module, the individual name of the property is added as a prefix to the dimension name. The value of a property named "Color" is, for example, represented by the three GET-parameters `DIM_COLOR_RED=...&DIM_COLOR_GREEN=...&DIM_COLOR_BLUE=...` in a WMS query. IWAL

uses these additional sample dimensions for updating the properties of a module prior to the creation of the plot data.

Handling of WMS requests

Each IWAL module has its own URL, to which the IWAL client (or any other WMS client) can send WMS requests. Information about the respective case study is managed on the server-side as part of the user session data that is associated with a single client. If there is no user session data available (e.g. because the client has not logged in), a default case study is used. All WMS operations are handled by the `wms_request` view of the `wms` Django application. The server path associated with the `wms_request` view is `/m/` followed by the server name of the module. Django allows parts of the URL to be passed on as a function parameter to the view functions. So if the server has the URL `http://iwal.ethz.ch/iwal`, the `wms_request` view function can be invoked by a request to, for example, `http://iwal.ethz.ch/iwal/m/horizontalcs`. Each IWAL module provides its own server name that is used as part of these URLs.

The `wms_request` view expects a set of GET-parameters that form either a valid WMS request, or an extended request using the nonstandard parameters of IWAL. Table 4.3 provides an overview of the GET-parameters recognized by the `wms_request` view.

The WMS specifications define the two mandatory operations *GetCapabilities* and *GetMap*. In addition, the IWAL WMS server implements a third operation: *GetData*. If a *GetCapabilities* command was issued, IWAL returns an XML file with metadata describing the available layers, styles, dimensions, etc., of the current case study associated with the user session. For the *GetMap* and *GetData* commands, the `wms_view` function is responsible for obtaining and returning the corresponding plot data to the client.

If the plot data was cached, it is directly returned. If it has yet to be created, the `wms_view` function selects the module object according to the current case study and the given module server name. It parses the WMS-bounding box constraints, the parameters of the vertical bounding box (if available), the map layer, as well as the image resolution from the HTTP GET-parameters and calls the `update_properties_from_http_request`-function of the IWAL module for an update of the input data information. Finally, the `create_image`, `create_legend`, or the `get_data` function of the IWAL module object is called and the result is optionally cached and finally passed on to the client.

Creation of the plots

The code for the creation of plot data in the form of raster graphics images is located in the `create_image` function of the `IWALModule` subclasses. All current modules, except for the `Trajectories` module, implement this member function. While the `SatImage` and the `Coastlines` modules use the *ImageMagick*[34] command line tool for cropping existing image files in order to match the given bounding box, all other modules call external scripts that use the Python libraries PyNGL and PyNIO for the creation of the plots. For some of

[34] `http://www.imagemagick.org/script/index.php`, last accessed: 10-January-2013

Parameter	Description
REQUEST	The name of the (WMS) operation to be performed. Valid values are "GetCapabilities", "GetMap", and "GetData".
WIDTH and HEIGHT	The resolution of the resulting image.
BBOX	The horizontal bounds of the plot, or the coordinates of the start position and the end position of a vertical cross section.
LAYERS	If the module supports multiple layers, the name of the layer is given by this parameter. The default layer is `map`. Some modules provide an additional `legend` layer for the creation of an image representing the respective color scheme of the plot.
CSTYPE, VERTLOW, and VERTHIGH	These parameters are not contained in the official WMS standard. They are nonstandard parameters introduced by IWAL for referencing different types and positions of vertical cross sections. The CSTYPE parameter specifies the type of the cross section (possible types are free linear interpolation or linear interpolation in latitudinal or longitudinal direction, as well as a cross section along a great circle on the earth). While the start and end point are given by the BBOX parameter, the vertical range is specified through the VERTLOW and VERTHIGH parameters, providing vertical positions given as pressure values.
CRS	The coordinate reference system of the map. IWAL currently supports only one coordinate system for horizontal maps, which is CRS:84 (see Appendix B.3 in de la Beaujardiere, 2006). For vertical maps, this value is ignored.
STYLES	This mandatory WMS parameter is currently ignored by IWAL, since all map layers only provide one default rendering style.
DIM_*	IWAL uses WMS sample dimensions for the representation of the respective values of all plot properties.

Table 4.3.: The GET-parameters recognized by the `wms` application view.

these plots, ImageMagick is used in a post-processing step in order to create a transparent background.

It was not possible to integrate the PyNIO and PyNGL based scripting code directly into IWAL, due to memory leaks within the PyNGL library. Those leaks became perceivable after hundreds of calls to the corresponding library functions. We were able to track the leak down to a minimum set of PyNGL function calls. The workaround implemented in IWAL is to call the external scripts for plotting the data in a separate Python environment, created by a `os.system` call that starts a new Python interpreter and passes all plot options to the external script in form of command line parameters.

Caching of the plots

The production of a plot usually involves the handling of large data sets that have to be loaded into memory, to be interpolated, and finally to be visualized using a suitable algorithm. These steps can take a considerable amount of time, see the results of the benchmarks in Sect. 4.5.1 for a detailed analysis. In order to minimize the time a response to a WMS request could take, IWAL uses two different caching methods: one for the short-term caching and one for the long-term caching of plots.

The first method is based on Django's built-in cache framework. Django offers the possibility to integrate different external caching back ends into a Django application. On the `iwal.ethz.ch` server, for example, IWAL is configured to use *memcached*[35] with a total of two gigabytes of available RAM for a direct caching of the most recently accessed plots. The second caching method of IWAL is a custom implementation for storing the created plots on a hard disk drive, using a dedicated directory structure and automatically generated unique filenames. Both methods are configured by global settings in the `settings.py` file.

The caching is applied in the `wms_view` function. For each request, the function first queries the corresponding IWAL module about whether the requested plot is subject to caching at all. If so, the next step is the creation of a *cache key*. This key is a string composed of the parameters of the WMS query. It uniquely identifies a single plot. It starts with the name of the case study, followed by the module's server name. Finally, all parameter values of the HTTP-request are added to the string, separated by underscores ("_"). This cache key is then used as a key for memcached, and as a filename for the caching on disk. In order to prevent the storing of all plot images in the same directory, we prefix the cache key with a path consisting of a sub-directory with the name of the case study, followed by a sub-directory with the name of the currently selected variable of the plot (if available), and finally a sub-directory corresponding to the current time step (if available).

As mentioned above, not all modules advice a caching of their images. For example, in the vertical cross section modules, the position of the cross section can be arbitrarily chosen by the user. Because of this, it is very unlikely that multiple queries refer to the exact same position, which makes a caching of these plots useless. In case of the horizontal cross sections, the IWAL client splits the viewport into a set of equally sized and uniformly offset tiles. This way, the client ensures that single WMS queries have a chance of being issued multiple times, making a caching of the tiles reasonable without restricting the user in the selection of the actual viewport position.

[35] `http://memcached.org/`, last accessed: 10-January-2013

However, the number of possible tiles and with it the number of distinct WMS queries is still comparably high, since we are dealing with four-dimensional data sets on varying height levels, offering different variables to be displayed with varying coloring schemes. By comparison, Google Maps offers only a very limited set of fixed, two-dimensional map layers (a stylized map and a satellite view, together with layers such as “traffic” and “weather”). A detailed analysis of the required amount of time and memory for the precomputation and caching of all tiles of a typical IWAL case study, as well as an analysis of the performance gain achieved by the two caching strategies, can be found in Sect. 4.5.1.

4.3.3. The `wmsclient` application

The `wmsclient` Django application implements all server-side features of the IWAL client application. This IWAL client application consists of the interactive main view of the WMS client at its core. In addition, it covers all functionalities for the user management and course management, and for the case study selection.

In contrast to the single view of the `wms` application, the `wmsclient` application provides 15 different views accessible at individual server paths, as well as the complete database models of IWAL. This section focusses on the different views of the `wmsclient` application. The design of the database models is discussed in the separate Section 4.3.4.

The views can be divided into two groups. The first group consists of those views which provide all dynamic HTML files of the client application. The second group contains views which perform single actions or provide server-side services (e.g. the view at the server path `/logout` logs out the currently logged-in user and returns a redirection to the login page, and the `/get_saved_states_revision` URL suffix returns a JSON object which contains the number of the latest revision of the shared saved states).

All of the served HTML files of the `wmsclient` application contain dynamic elements that are generated individually for each request. The generation of the particular content is based on the data of the user session, and on information stored in IWAL’s database. For the creation of the dynamic source files, Django’s built-in template engine is used. All completely static files, e.g. static images and most of the JavaScript source files used for the implementation of the client’s web GUI, are located outside of the `wmsclient` application in the `static` directory and are directly served by Django without the indirection through a Django application.

The views of the interactive WMS client

The most important part of the IWAL client application is the implementation of the interactive WMS client and its GUI. This WMS client is accessible through the main view of the `wmsclient` application, which is available at the server path `/main`. The view returns an HTML document which contains the layout of the WMS client. Furthermore, the document contains JavaScript code and include-directives for all static JavaScript source files containing the remaining implementation of the WMS client. The main task of the `wmsclient` application is the generation of HTML and JavaScript code for the setup and initialization of all modules of the selected case study. In addition, information about the currently logged-in user and its privileges are inserted into the source files.

The interactive WMS client sends requests to the following additional views of `wmsclient` for the invocation of several server-side tasks:

- The `/save_state` view is used by the client to pass the current client state and some meta information, for example a user-chosen name, in form of a JSON object to the server. The server stores this state as a new object in its database for later restoration. See Sect. 4.4.7 for details on the serialization of client states.
- `/update_state` works similar to the `/save_state` view, but it updates an existing saved state, instead of creating a new one. For this, the view requires less parameters than the `/save_state` view.
- `/get_saved_states_revision` returns an internal revision number which is increased every time a saved program state is added or updated. The number is returned in form of a JSON object. It is used by the client to decide whether or not the displayed list of available saved states needs to be updated.
- `/get_num_saved_states` returns the number of available saved states of the current case study in form of a JSON object.
- `/get_saved_state_names` returns the names of a selectable subset of all available saved states associated with the current case study in form of a JSON object.
- `/get_saved_state_data` returns information required for restoring a given saved state in form of a JSON object.
- `/delete_state` deletes the given saved state from the database.
- `/set_view_size_mode` sets the user's preferred mode for resizing the viewports. See Sect. 4.4.3 for details on the available reshaping methods.
- `/convpng` is a helper view for storing the content of the current viewports in form of an image file with a predefined filename[36].

A detailed description of the implementation of the interactive WMS client, including descriptions on how the views offered by the `wmsclient` Django application are utilized, is found in Sect. 4.4.

The remaining views

The IWAL client contains additional functionalities forming the frame of the interactive WMS client view. The corresponding additional views of the `wmsclient` application provide the main login page, the pages for user creation and for the management of the user data, the handling of the enrolled courses, and finally a page for the selection of the case study and of all related options.

While the implementation of the GUI of the WMS client consists mostly of JavaScript code, the login and user management pages are implemented mostly in HTML. The basic UI elements required by those views are realized using HTML forms. The dynamic information

[36]see `http://greenethumb.com/article/1429/user-friendly-image-saving-from-the-canvas/`, last accessed: 14-January-2013, for details of this method.

that is embedded in the HTML documents is generated in the respective view functions using the data from the particular user session and from the IWAL database.

The following views provide the dynamic HTML documents for these remaining tasks, as well as all associated server-side services:

- The `/login` view provides a welcome page and a form for logging in users.
- The `/logout` view logs out the current user and redirects to the login page.
- The `/new_user` view shows a form for the creation of a new user.
- The `/change_user` view allows the changing of the data of the currently logged-in user.
- The `/course_management` view shows all courses the current user is enrolled into, and allows to add inscriptions by entering a valid course code.
- The `/select_case_study` view is used for selecting a case study out of all available case studies of the current user, together with a switch for entering the advanced user mode (if available).

Details of these supporting functionalities of the IWAL client application for the user, course, and case study management can be found in the corresponding Section 4.4.1.

4.3.4. Database design

IWAL uses its database for storing all persistent data of the application, including the user information, the setup of the modules, case studies, and courses, as well as the required information for storing and sharing individual program states.

All database models of IWAL are defined as part of the `wmsclient` Django application. IWAL uses Django's object-relational mapper for the definition of all database entities and their relations[37]. The definition of the models can be found in the file `wmsclient/models.py`. For applying changes to an existing database model, the South software tool is used (cf. Sect. 4.2.1, and the example in Appendix F.2).

The content of the database can be administrated through the built-in Django administration application. This application is accessible at the server path `/admin`. Django offers a system for the association of users with certain permissions. The administration interface is available for all users with the `is_staff` and the `is_superuser` flags set. The first flag is required for accessing the main view of the `admin` application, the second flag grants all required privileges for editing the database entries.

Besides the IWAL applications, several built-in Django applications and the South application store information in the database. All tables are prefixed by the application name. Out of all built-in database models, the only one directly referred to by IWAL is the `user` model of the `auth` Django application, stored in the `auth_user` table.

In Fig. 4.2, an entity-relationship diagram provides an overview of the user entity and all custom IWAL entities of the database, together with their relations. From all attributes

[37] see `https://docs.djangoproject.com/en/1.2/topics/db/models/`, last accessed: 19-March-2013.

of the external user entity, only those which are of interest for the IWAL application are depicted in the diagram. Below, the entities shown in the ER-diagram and their attributes are discussed in detail.

The user entity

This entity is provided by the auth Django application. The user entity contains all basic user information, such as username, first_name, last_name, and the email of the user. The dates of the last login, as well as the date the user was created, are stored in the corresponding attributes last_login and date_joined. The password of the user is stored in form of a salted hash value[38] in the password attribute. This attribute also contains some meta information about the used hashing method and the value of the salt. In addition, the entity contains several boolean flags. The is_staff flag indicates whether or not the user is allowed to access the administration web interface of the database, and setting the is_superuser flag automatically gives the user all required privileges for editing the database via this interface. The is_active flag can be set to "false" instead of deleting a user, keeping possible references in the database alive. Note, that the deletion of users is not yet implemented, but planned for the near future of IWAL. Currently, the only way to delete a user is via direct changes of the database using the administration interface.

By default, Django adds a primary key called id to each entity. It is internally used to uniquely represent single database entries and their relations. Although it is part of every entity, it is not further mentioned in the discussions of the other entities below.

The userprofile entity

The userprofile entity was introduced for storing additional attributes of a user, without changing the model of the built-in auth application. IWAL uses the signals mechanism of Django[39] for the creation of a userprofile every time a new user was created. We expand the flags of the user entity by the additional flag privileged_user, which indicates whether or not the user may access the advanced modules of a case study. Whenever the user decides to access a case study in this privileged user mode, the advanced_mode flag is set. The help_screen_shown flag indicates whether or not the user has already seen the initial help dialog in the main WMS client view. This help screen is shown only once to each user. Further, the last accessed case study is stored (last_case_study[40]), as well as the user option to automatically return to the last case study (remember_case_study). Finally, the flag can_edit_itself, which is initially set to "true", can be deactivated for user instances that are shared by multiple real users. It indicates whether or not a logged-in user can edit its own settings, remembers its last case study, and is capable of storing and deleting program states. An example use case where this flag should be set to "false" is a class of students sharing a single user account for accessing IWAL.

[38] see https://docs.djangoproject.com/en/1.2/topics/auth/#passwords, last accessed: 16-January-2013

[39] see https://docs.djangoproject.com/en/1.2/topics/signals/, last accessed: 16-January-2013

[40] Note, that this member variable of the database model class represents a relation between two database entities, and is therefore not shown by this name in the ER-diagram following the conventions of Chen (1976).

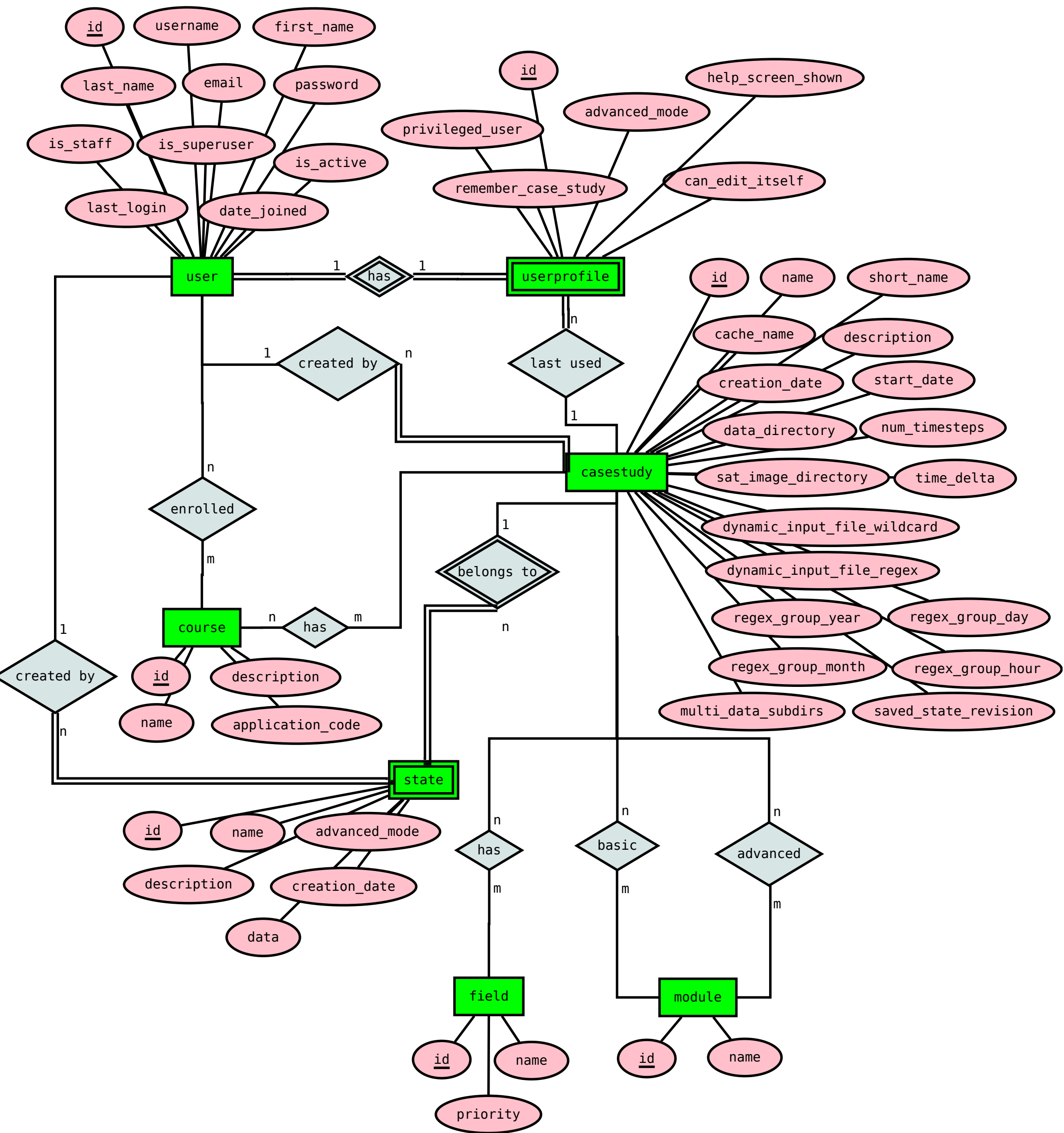


Figure 4.2.: ER diagram of the custom database entities of IWAL, together with the additional `user` entity. The name prefixes indicating the Django applications where each entity is defined are omitted. The diagram is based upon the conventions proposed by Chen (1976) and was created using Dia (`https://live.gnome.org/Dia/`, last accessed: 16-January-2013).

The course entity

This entity represents courses into which users can enroll. Each course stores its name, a short description, as well as an application_code. The application code can be entered by any user in order to enroll to the respective course. The most important data stored for each course is the set of corresponding case studies (casestudies). Users have access to their individual set of all case studies from all of their enrolled courses.

The casestudy, field, and module entities

The casestudy database entity contains attributes for storing the setup of the input data of a case study and additional meta information. This meta information consists of a name, a short_name to be displayed in areas of the UI with limited space, a name prefix used for the caching of the tiles (cache_name), a textual description of the case study (description), as well as the user (creation_user) who created the case study and the date of the creation (creation_date). The saved_state_revision attribute contains a number which is incremented by the server every time a serialized program state of this case study is stored, updated, or deleted (cf. Sect. 4.4.7).

The remaining attributes cover the setup of the input data of the case study. Based on these database settings, the input data information of the corresponding IWAL module is updated prior to the actual invocation of the plot data creation. However, it is each module's own responsibility to interpret this information and to access the correct input files. In Sect. 4.3.2, the different ways in which the current IWAL modules handle their input data are described.

In the simplest case, the metadata provided by the casestudy database entity only specifies the directories where the input files can be found (data_directory and sat_image_directory). The data may also be distributed to a set of user-selectable sub-directories, for example in case of forecast data. This set is specified through the multi_data_subdirs attribute. The range of accessible time steps of the case study can be specified by a start_date, a total number of time steps (num_timesteps), and by the duration of a single time step (time_delta), typically interpreted by the modules as number of hours. If the available time range depends on a changing set of input files, these files can be specified through a filename including wildcards (dynamic_input_file_wildcard). IWAL searches all files in the data directory matching this wildcard expression. It is assumed that each file represents a single time step, with the date given in the filename. For parsing the date out of a given filename, a regular expression (dynamic_input_file_regex) is used. This regular expression has to contain four groups, numbered from one to four, one for each component of the date (year, month, day, and hour). The mapping of the group numbers to the date components is specified by integers stored in the regex_group_year, regex_group_month, regex_group_day, and regex_group_hour attributes, respectively.

The set of available input variables (fields) is realized as a many-to-many relation between the casestudy and the field entities. A field entity represents a data variable and contains only two attributes: its name, and a priority value which is used by the client as the key for sorting the variable names. The available modules of the case study for the normal user mode (modules) and for the privileged user mode (advanced_modules) are stored in form

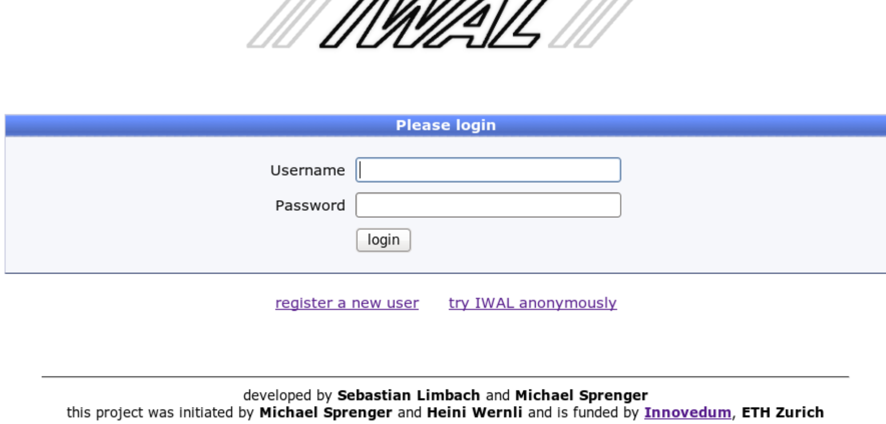


Figure 4.3.: The login screen of IWAL.

of two additional many-to-many relations between the `casestudy` and the `module` entities. A `module` entity contains only the server name of the corresponding Python `IWALModule` object.

The `state` entity

The `state` entity contains all attributes required for storing the complete program state of the IWAL WMS client. It consists of the respective case study (`casestudy`), represented through a relation between the `state` and the `casestudy` entities, the user mode, represented by the `advanced_mode` flag, as well as a large JSON representation of the viewport settings, including all current plots of each viewport with all their current plot properties, as well as all global settings of the client. This JSON representation is stored in the `data` attribute. The details of the serialization of program states is described in Sect. 4.4.7.

4.4. Client architecture

This section discusses the architecture of the IWAL client web application that runs in the individual browsers of the IWAL users. The core of the IWAL client is the implementation of the interactive WMS client. It is framed by several HTML pages for the user and course management, and for the case study selection. The realization of these framing HTML pages is described in the upcoming section, Sect. 4.4.1. The remaining parts of Sect. 4.4 cover the details of the implementation of the IWAL WMS client, including the ways in which the client creates the WMS requests that are sent to the IWAL server (see Sect. 4.4.6), as well as the serialization and sharing of the program states (see Sect. 4.4.7).

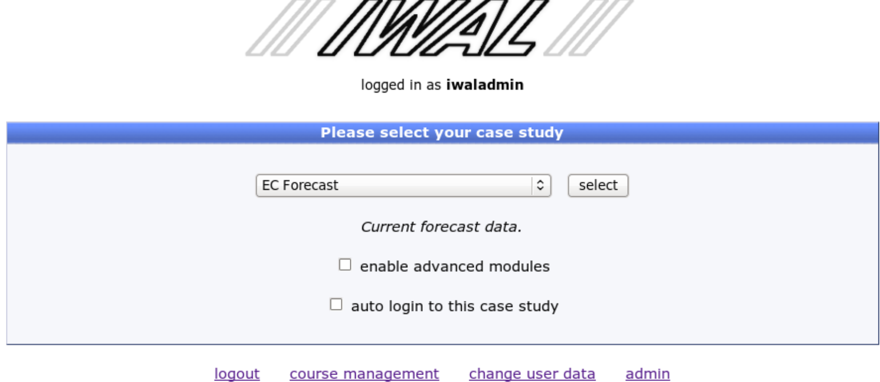


Figure 4.4.: The case selection screen of IWAL.

4.4.1. User, course, and case study management

The WMS client interface of IWAL can only be fully accessed by users who are logged in and have selected one of the available case studies. Additionally, privileged users have the choice between working with the basic set or with the advanced set of modules of a case study. In order to log in and for the case study selection, the user has to accesses IWAL via the main login page, available at the server path `/login` (cf. Fig. 4.3). If the user tries to access most of the other pages without being logged in, a redirection to the login page is returned. On the other hand, if an already logged-in user accesses the `/login` page, the user is directly redirected to the case study selection view (`/select_case_study`) or the WMS client view (`/main`). A restricted version of the main WMS client view with a predefined case study can also be accessed without being logged in.

The login screen allows an already registered user to log in, which leads to a redirection to the case study selection view, if successful. Users without login data can choose between a link to the `/new_user` page, allowing them to register as a new user, and the direct link to the `/main` page, which allows them to try IWAL anonymously using a default case study, as mentioned above.

The case study selection screen (cf. Fig. 4.4) contains a drop-down list for the selection of the case study and the mentioned switch for using the advanced modules, if available. In addition, the user can choose to be automatically redirected to the WMS client with the selected case study after the next login, skipping the case study selection screen.

The page contains a link for logging out of IWAL (`/logout`). For users with the `can_edit_itself` flag set, the page contains additional links. The first additional link leads to the `/course_management` page where the enrolled courses of a user are shown and new courses can be added by the input of valid course codes. Another link points to the `/change_user` page for editing the own user data. All staff members with the respective privileges can also enter the administration interface of Django from here (`/admin`).

The main WMS client view is accessible for all users by pressing the “select” button. This button is part of an HTML form which also includes the case study selection and the respective options. The form is configured to send its data as POST-parameters to the `/select_case_study` view itself. If a request with POST-parameters is send to this view,

it stores the persistent options in the database and the ID of the selected case study in the user session and redirects the user to the `/main` view.

The same technique for realizing the user interaction through HTML forms is used in the `/new_user`, the `/change_user`, and in the `/course_management` view. In all cases, the view functions of the `wmsclient` application decide whether they serve the HTML page or process the data entered into the HTML form based on the presence of POST-parameters. The served HTML pages contain dynamic elements that are created by Django's template engine. The only interactive functionality realized in form of a JavaScript function is the changing of the description text of the currently selected case study in the case study selection view.

4.4.2. WMS client basics

The centerpiece of the IWAL web application, the interactive WMS client, is available through the WMS client view. The corresponding server path is `/main`. The view consists of several dynamic, user-controlled interface elements. An example of the WMS client view during a typical user session can be found in Fig. 4.5. The WMS client is realized in form of JavaScript and HTML code. In Sect. 4.3.3, the server-side Django application `wmsclient` for the provision of all dynamic source files and all server-side services required by the client was described in detail. The corresponding views providing these services, as well as the single view of the `wms` application for the handling of WMS requests, are accessed by the client web application in form of *Ajax*[41] requests. The WMS requests are generated whenever the user changes the properties of the viewports of the plotted data. The server-side services are invoked mostly in response to a user interaction, they are responsible for tasks such as the storing and the restoring of the program state.

The main interactive elements of the WMS client view are the two viewports at the center of the page. To its left and its right are two columns containing additional UI elements. The left column is divided into six different areas, containing UI elements for the following means of user interaction:

1. the selection and configuration of the available IWAL modules, as well as the creation of new plots
2. obtaining information about the current cursor position
3. the configuration of the currently selected viewport
4. the configuration of global IWAL settings
5. the creation, update, and restoration of all saved program states
6. invoking general commands, such as showing a help screen, changing the current case study, ending the IWAL session, and entering the administration interface

The right column of the view contains a list of current plots, as well as the available plot properties of the currently selected plot.

[41] *Ajax* is an acronym of *Asynchronous JavaScript and XML*. It stands for web technologies that allow a client to perform asynchronous HTTP requests to the server. The origin of the term Ajax goes back to the article "Ajax: A New Approach to Web Applications" by Jesse James Garrett, 2005. See `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, last accessed: 22-Feb-2013.

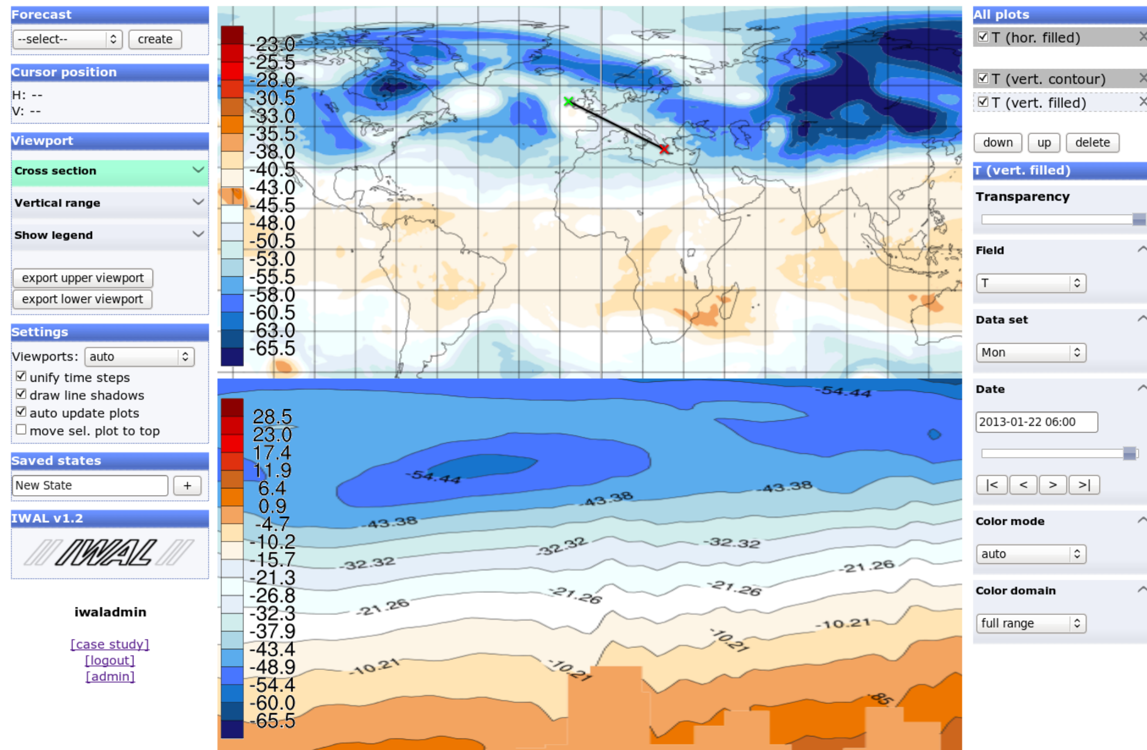


Figure 4.5.: WMS client view with two visible viewports, showing horizontal and vertical temperature plots. To the left and right of the viewports are the two columns containing the UI elements for most of the user interactions.

The basic structure of the WMS client view, the three column layout, as well as some basic UI elements, such as most of the drop-down lists, the buttons, and the complete general commands section, are implemented in form of HTML code. The layout is realized as a table with three columns. In addition to the two columns mentioned above, the viewports are positioned by their own table column at the center of the view.

Most UI elements of the application are implemented by means of JavaScript code. This code is, for example, responsible for drawing and controlling the viewports, for updating the property views whenever a new module, plot, or viewport is selected, and for the creation and invocation of WMS queries in response to changes of the plot properties and of the viewport properties.

Dynamic code generation

Parts of both the HTML code and of the JavaScript code of the WMS client are dynamically created by means of Django's template engine. For example, the JavaScript objects representing the available modules together with their default setup are created and inserted into a common list through the following template code:

```
{% for module in available_modules %}
modulesList.addModule(new Module("{{ module.name }}",
   "{{ module.name_template }}", {{ module.properties|safe }},
```

```
    {{ module.server_infos|safe }}, globalTimestepProperty));
{% endfor %}
```

The template engine uses tags delimited by the character sequences `{%` and `%}` for the definition of control structures and for template inheritance[42]. Variables are denoted by double pairs of curly brackets. The `available_modules` variable is defined in the Python code for rendering the `/main`-view. The `module` objects are instances of the corresponding `IWALModule` classes of the `wms` Django application (see Sect. 4.3.2). As this example shows, one can directly access object member variables and properties by appending them to the object name, separated by a dot. The additional `|safe` suffix indicates that the value of a property has not to be escaped in any way. In the example above, escaping could invalidate the JSON representations returned by `module.properties` and `module.server_infos`.

Object oriented design of the WMS client

The JavaScript implementation of the WMS client application realizes a complex user interface with all of its functionalities, including interactive viewports and different controls for the manipulation of the module and plot properties. The general design of the program re-uses many ideas and principles that were already successfully applied in the development of Insight, since the requirements of the two programs overlap in many points.

The basic object oriented design of the web application follows the *Model View Controller* (MVC) design pattern (Gamma et al., 1995), as it was used for the development of Insight (cf. Sect. 2.4.2). The UML diagram in Fig. 4.6 shows the main classes of the JavaScript implementation. In the upcoming sections, the main classes of these three categories are discussed in detail.

4.4.3. WMS client models

The central model objects which are created once on the start of the client, and which exist for the whole execution time, are singleton instances of the `ViewportManager` and the `ModulesList` classes. The `ViewportManager` is responsible for the creation of the (currently three) viewports available in the WMS client. The three different viewports are represented by objects of the `ViewportCylindrical`, `ViewportVertical`, and the `ViewportSeparate` classes. All these classes share the common `Viewport` interface, which allows access to the list of all plots that are part of the respective viewport in form of a `PlotsList` object. These three viewports, together with the corresponding lists of plots are also existing for the whole lifetime of the program. Together, these classes form the static core of the WMS client model structure.

Viewports

The WMS client contains, in its current state, three instances of the three different viewport classes: The two main viewports at the center of the UI, and a viewport showing its plots in separate dialog windows. The horizontal viewport, an instance of the `ViewportCylindrical`

[42] see `https://docs.djangoproject.com/en/1.2/topics/templates/#id1`, last accessed: 25-March-2013.

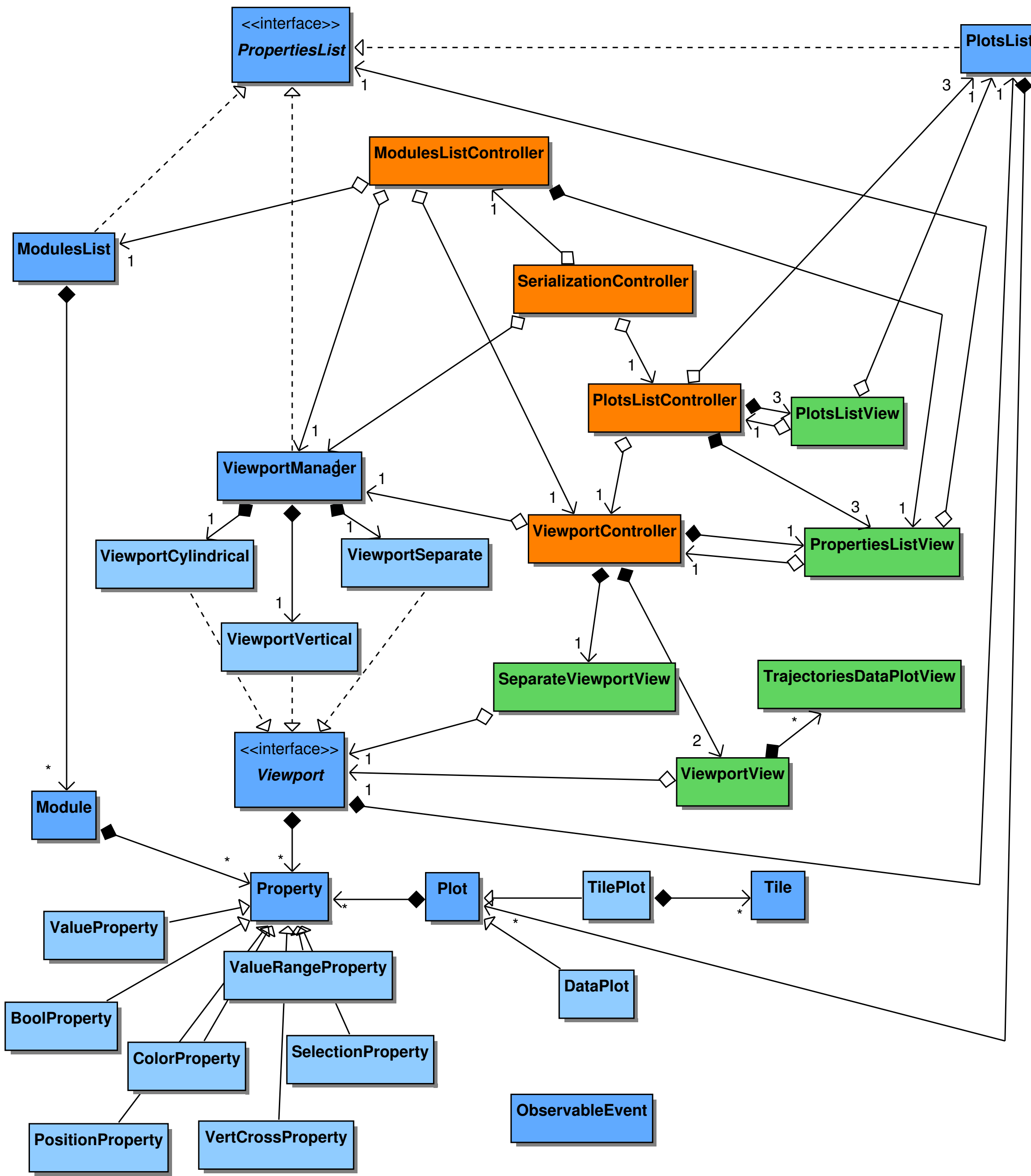


Figure 4.6.: The main classes of the JavaScript implementation of the IWAL WMS client. The model classes are depicted as blue boxes, the controller classes are visualized in orange, and the view classes are marked in green. The interfaces shown in this diagram are purely conceptional and do not have an actual implementation. Aggregations involving them are part of the classes that implement the interface. For reasons of simplicity, all connections with the `ObservableEvent` class are not shown.

class, provides a fixed equidistant cylindrical map projection of the geographic data, whereas the vertical viewport (`ViewportVertical`) supports different types of cross sections, selectable by the user. The visualization of the horizontal and the vertical viewports is realized in form of HTML5 canvas elements. The `ViewportSeparate` object operates as a container for all plots that are visualized in separate, non-modal dialog windows. These dialog windows are realized using JQueryUI dialog widgets. Each viewport manages a list of current plots, as well as a separate set of viewport properties. Both the list of plots and the list of properties are accessible through the common `Viewport` interface.

The singleton `ViewportManager` object takes care of the creation of all viewports. A main feature of the `ViewportManager` class is the implementation of the `PropertiesList` interface, which is used by a corresponding view object for the display of a list of properties of the *selected* viewport. One of the three viewports is marked as the currently selected viewport by the `ViewportManager`. This selection can be changed by the user through the selection of a plot of the respective viewport.

Another responsibility of the `ViewportManager` is the management of the viewport shapes. The manager object gets informed about the current window size which determines the available space without forcing the user to scroll in any direction. Based on this information, the manager object distributes this space along the two main viewports. The `ViewportManager` has a flag that decides whether the horizontal viewport is *enlarged*, which means that it gets the majority of the available space, or not, which results in an even distribution of the available screen space to both the horizontal and the vertical viewport.

The last responsibility of the `ViewportManager` is the representation of the one single selected *tool property*. A tool property is a special property whose value is visualized and can be modified through user interactions in either the horizontal or the vertical viewport. Tool properties are discussed in detail in the next part of this section.

Properties

The `Property` classes of the WMS client correspond directly to the Python `Property` classes of the `wms` Django application (cf. Sect. 4.3.2). They are constructed using the `createProperty` factory function that takes the JSON representation of a Python `Property` object as a parameter. This representation contains a type string, which is used for the instantiation of the correct JavaScript `Property` class. The remaining elements of the JSON representation contain the initial internal state of the property, as well as meta information such as the name of the property.

The instances of the `Module` and `Plot` classes, and all objects providing the `Viewport` interface contain their individual set of properties. All these instances are collected and managed by objects of the corresponding `ModulesList`, `PlotsList`, and `ViewportManager` classes. These container classes implement the `PropertiesList` interface in order to provide access to the list of properties of a single `Module`, `Plot`, or `Viewport` instance. Additionally, they provide functions for the notification about a change of the provided list of properties. The `PropertiesList` interface is accessed by the `PropertiesListView` objects for visualizing the corresponding lists of properties (see Sect. 4.4.4).

A special group of properties are the tool properties. A single tool property at a time can

be globally selected by the user. This selection is represented by the `ViewportManager` object and managed primarily by the `ViewportController` object, in collaboration with the `ModulesListController` and the `PlotsListController` objects. The selected tool property is visualized directly as part of either the horizontal or the vertical viewport, and can be interactively modified in the corresponding viewport area by the user. An example of a tool property is the `PositionProperty`, which stores a geographic position in form of a latitude/longitude coordinates pair, and which can be set to any position by clicking with the mouse into the area of the horizontal viewport. Another example is the `VerticalCrossSectionProperty`, for which the position of the vertical cross section is displayed in the horizontal viewport. The start point and the end point of the displayed cross section can be interactively altered by dragging them around using the left mouse button. By clicking and dragging the middle mouse button the user can set the cross section to a completely new location.

IWAL modules

The `ModulesList` object is populated on startup with all `Module` objects which represent the available IWAL modules. Each `Module` object has a one-to-one correspondence to a server-side `IWALModule` object. The initialization of the `Module` objects is realized by JavaScript code that is dynamically created based on the configuration of the user-selected case study. As described in Sect. 4.3.2, IWAL manages a sets of all available modules with properties in their default state. Out of this set, all modules of the current case study are selected. This selection takes into account whether or not the user has activated the privileged user mode.

The server updates the input data settings of these modules and passes them to Django's template engine. The template engine creates all JavaScript commands for the creation and initialization of all `Module` objects and for the population of the `ModulesList` object. For each module, the template engine extracts the respective internal representation in form of several JSON strings, and passes them on to the constructor of the `Module` class. This internal representation contains a list of all properties of the module, each of which is represented by a JSON string itself. In the constructor of the `Module` class, all `Property` JavaScript classes are created through the `createProperty` factory function that was introduced above.

Plots

Based on the information stored in the `Module` objects, an arbitrary number of plots can be created. Each plot is represented by a `Plot` object, which contains a set of `Property` objects representing the mutable plot options. Initially, these properties match the values of the properties of the corresponding module. Furthermore, a `Plot` object is responsible for querying, storing, and updating the corresponding plot data for the visualization. A special feature of the `Plot` class is the support of *name templates*. A name template is a string that may contain placeholders referring to any own property. If such properties change their value, a string representation of the new value is added to the name at the position of the respective placeholder. This technique is used to directly show, for example, the currently visualized data variable of a plot in the plot's name.

The module also defines the unique viewport each plot is associated with. Newly created `Plot` objects are stored directly in the `PlotsList` instances of the corresponding viewport. Whenever the properties of a viewport change, all plots associated with the respective viewport are updated. An update is also triggered whenever any of the plot's own properties has changed. The update of the plot data is implemented individually in the different subclasses of the `Plot` class.

Two classes derived from the `Plot` class are currently part of the implementation: `TilePlot` and `DataPlot`. The `TilePlot` instances contain a set of `Tile` objects, each of which contains the data of a single image. The `DataPlot` objects store data that is intended to be visualized directly on the client-side. Both types of plots have in common that they contain all required information for updating their internal representation whenever a plot option or the viewport settings change. The `Plot` base class additionally manages functions informing about the current status of the plot data. The plot data can be either "up-to-date" or "outdated", which is indicated by a corresponding flag. Additionally, the status of the server requests can be queried. It may be either "done" (the plot data is ready to be visualized), "loading" (the plot waits for one or more server requests to be completed), or "failed" (at least one server request failed).

All the details of creating, updating, and managing the plot data are covered in a separate section, Sect. 4.4.6, since these actions contain the fundamental functionalities of the WMS client application. The section covers the creation of appropriate WMS queries, the management of plot tiles, as well as the IWAL specific extension of the WMS standard for the processing of vertical cross sections and data plots.

4.4.4. WMS client views

The complexity of a model and its associated user interactions has a direct impact on the complexity of the view class responsible for the model. On the one hand, there are models that are visualized by a single widget, for example the `ModulesList`, which is visualized by a single HTML drop down list. On the other hand, there are more complex models requiring a combination of multiple widgets and custom visual elements to be displayed properly. An example for this is the `PlotsList`, where each plot is visualized by its name, with different formatting and symbols indicating the plot's status, and with several buttons allowing the user to trigger all plot-related actions provided by the application. For these more complex views, individual view classes are implemented. The WMS client contains five major view classes, which will be discussed in this section.

The `PlotsListView` class

The `PlotsListView` class is responsible for displaying the data of a single `PlotsList` instance. Since the WMS client consists of three viewports, each of which has its own `PlotsList` instance, the application maintains three corresponding view objects. All three views are located in the top-right corner of the IWAL application window. Each plot is visualized as a single line consisting of a checkbox controlling the visibility, the (possibly mutable) name, an optional loading indicator ( ), an optional reload button ( ), as well as a delete button ( ).

The `ViewportView` and the `TrajectoriesDataPlotView` classes

Two instances of the `ViewportView` class are responsible for displaying the content of the two main viewports. Each viewport object operates on its own HTML5 canvas element. The drawing of plot images and the visualization of data plots is realized using JavaScript drawing commands. The image of each viewport is composed of the following four components:

1. The images of all fully loaded tiles of all tile plots, and placeholder images for all loading tiles.
2. The image of a single plot's legend. For this, each `Viewport` object maintains a reference to the legend of the last selected plot of the viewport that had a legend.
3. The graphical representation of all data plots. Each data plot is visualized through its own data plot view class. The view object of the respective viewport is responsible for the instantiation of these individual view classes. This instantiation is realized through a corresponding factory method. The data plot views have access to the underlying viewport, but may also use additional visualization elements, for example additional dialog windows, for the display of the data. At the moment, the only implemented data plot view is the `TrajectoriesDataPlotView` class, which visualizes trajectories in form of line strips directly in the viewport's canvas, and in form of an HTML table in a separate JQueryUI dialog widget.
4. The graphical representation of the current tool property. The `ViewportManager` instance manages the currently selected tool property and is responsible for informing a `Viewport` about a tool property that has to be visualized. This information is stored as part of the `Viewport` and is utilized by the `ViewportView` for the actual visualization.

The `SeparateViewportView` class

The management of all dialog windows containing the plots of the separate viewport is implemented in the `SeparateViewportView`. Instead of using real browser windows, the dialog functionality provided by JQueryUI is used. The JQueryUI functions allow IWAL to control several important options of the visualization of the dialog windows, which cannot be controlled in the same way for real separate browser windows. Examples are the controlling of the visibility of single dialog windows, and the customization of the overlay sequence of multiple dialog windows. Since the JQueryUI dialog windows are directly embedded into the single page of the application, they are seamlessly visible even in browsers which have no multiple window support (for example the Safari browser of iOS).

The `PropertiesListView` class

Multiple instances of the `PropertiesListView` class are responsible for visualizing the interface of all properties of the currently selected viewport, the currently selected module, and of the currently selected plot. Each `PropertiesListView` object is connected with exactly one object providing the `PropertiesList` interface. Because of this one-to-one relation, for each of the three `PlotList` objects of the viewports, a corresponding `PropertiesListView` instance exists. However, this is not noticeable by the user, since only one plot can be selected

at a time. Each PropertiesListView manages the UI representations of all properties of the list. These representations are created through the createPropertyView factory function. The individual views of each property are not realized in form of traditional classes, but as functions that directly create and prepare the corresponding HTML and JQueryUI widgets. The event handling is performed via anonymous callback functions, which are closures preserving the references to all created widgets and other local variables.

4.4.5. WMS client controllers

The current implementation of the WMS client contains four controller classes, which will be briefly discussed in this section. One instance of each controller is created at the start of the application, the objects exist for the whole lifetime of the program. All controllers have in common that they are responsible for observing the view classes and for carrying out the correct sequences of operations on the model classes, whenever a relevant event was triggered by the UI. Some of the more complex operations require multiple different controllers to work together, for example for realizing the strategy for the selection of the tool property.

As mentioned in the previous section, not every view is realized in form of a separate view class. Some views are realized in form of HTML elements that are already part of the HTML document provided by the IWAL server or created directly by the controller classes during their construction. The controller classes cover both the communication with these HTML elements, as well as with the instances of the view classes.

The `ViewportController` object

The ViewportController object is responsible for the two ViewportView objects, the SeparateViewportView object, as well as for the PropertiesListView object for displaying the properties of the selected viewport. The UI contains additional viewport related widgets that are also handled by this controller: Two buttons initiate the export of the content of a viewport in form of an image file, and several checkboxes realize the global user options regarding the viewports.

The directly controlled model of the ViewportController is the global instance of the ViewportManager class. One of the most important tasks of the controller is the translation of mouse and touch events of the two main viewport views into corresponding model function calls, invoking either changes of the viewport's properties (e.g. the currently visible area), or changes of the current tool property (e.g. when the user drags the start and end points of the vertical cross section property).

In addition, the controller manages the selection of the globally unique selected tool property in collaboration with the ModulesListController and the PlotsListController object. The ModulesListController, the PlotsListController, and the ViewportController objects each react on changes of their corresponding PropertiesList objects. They decide whether or not the availability of a candidate tool property changed locally with respect to the group of objects stored in the observed list (viewport, modules, or plots). This information is passed on to a central function of the ViewportController object, which finally decides about the update. The ViewportController then directly communicates with the ViewportManager object in order to set the actual selection of the tool property.

The PropertiesListView instances also inform the ViewportController object directly whenever a user explicitly selects a tool property, for example by clicking on the widgets representing the property.

The main global option controlled by the ViewportController influences the distribution of available screen space as mentioned in Sect. 4.4.3. The available settings of this option are *auto* (the horizontal viewport gets the majority of the available space as long as there are no vertical plots), *maximize* (the horizontal viewport always gets the majority of the available space), and *fit vertically* (both viewports get the same amount of available space).

The ModulesListController and the PlotsListController objects

The ModulesListController controls the ModulesList instance and the corresponding PropertiesListView object which shows the properties of the currently selected module. The list of modules itself is represented by an HTML selection element in the top left corner of the page, and the creation of a plot out of the currently selected module can be triggered by the HTML button next to it. These additional HTML widgets are defined in the main HTML page of the WMS client and are observed by this controller. Whenever the selected module changes, the ModulesList model instance is updated accordingly, and the ViewportController is instructed to set the currently selected viewport to the viewport associated with the module.

The single PlotsListController object is responsible for controlling the three PlotsList instances of all viewport objects. Besides the corresponding PropertiesListView objects, the controller additionally handles three HTML buttons allowing the sorting and deletion of single plots. One main responsibility of the controller is to ensure that only one plot is selected at a time out of all plots of the three plots lists.

As mentioned above, both the ModulesListController and the PlotsListController objects check the properties of a newly selected module or plot for tool properties and pass this information on to the ViewportController object.

The SerializationController object

The SerializationController contains the implementation of the storing and restoring of application states. The controlled views are the HTML elements contained in the “Saved states” section in the left column of the application’s layout, as well as the HTML widgets representing the global application settings. The controller interacts directly with the ViewportManager model object, and indirectly with the PlotsList model through the PlotsListController and the ModulesListController instances. The latter is used for the creation of restored plots based upon the available modules. Details of the serialization mechanism, with a focus on the representation of the program state and the necessary asynchronous communication with the server for sharing program states between multiple users, are covered in Sect. 4.4.7.

4.4.6. Creation and management of plot data

One of the main tasks of the WMS client implementation is the creation of WMS queries for obtaining the data required for the visualization of the plots. On the server-side, the instances of the subclasses of the `IWALModule` Python class contain the necessary code for the creation of the plot data (cf. Sect. 4.3.2). On the client-side, based on each module, an arbitrary number of plots can be created by the user. Each plot knows its associated module, and maintains its individual set of properties representing the plot options of the module (see the description of the `Module` model class in Sec 4.4.3). The server requests are created either directly by the plot object, or indirectly through any object the plot is composed of (e.g. by a `Tile` object in case of a `TilePlot` object). The plot class observes the requests and updates its status information accordingly to any of the possible values "done", "loading", or "failed". In addition, the flag indicating whether or not the plot data is up-to-date is set to the appropriate values. Whenever the viewport properties or any of the plot properties change, the plot is marked as "outdated" and needs to be updated. Each plot has a setting for whether this updated is performed immediately after any change, or whether it has to be explicitly triggered by the user.

WMS requests

This section covers the process of creating the WMS requests for the different types of plots of the WMS client. Each plot is responsible for querying the plot data from the server through WMS requests to its associated IWAL module. The general format of a WMS request was already discussed in Sect. 4.3.2. The following example shows a typical WMS GetMap request to a `HorizontalCrossSection` module of IWAL:

```
http://www.iwal.ethz.ch/iwal/m/horizontalcs?VERSION=1.3.0&REQUEST=
GetMap&CRS=CRS:84&LAYERS=map&STYLES=&BBOX=-180,90,0,-90&WIDTH=512&HEIGHT=
512&DIM_FIELD_VALUE=T&DIM_DATE_VALUE=0&DIM_LEVEL_VALUE=250&DIM_LEVEL_UNIT=
hPa&DIM_COLOR_DOMAIN_VALUE=full
```

This example was taken from a query issued for a single tile of a `TilePlot` object (see below). One can see how the position and resolution of the tile are encoded in form of the `CRS`, `BBOX`, and the `WIDTH/HEIGHT` GET-parameters. All parameters starting with "`DIM_`" contain information for transferring the state of all plot properties from the client to the server. For example, the `DIM_LEVEL_VALUE` and the `DIM_LEVEL_UNIT` parameters determine the state of the property that defines the height level of the visualization.

We decided not to use the optional `ELEVATION` parameter of the WMS standard for the selection of the height level. Instead, this selection is integrated seamlessly by a simple property without the need of special treatments in our implementation. Out of the same reasons, the modules do not use the optional `TIME` WMS-parameter for the selection of the current time step, but an additional plot dimension (e.g. `DIM_DATE_VALUE`) of a corresponding plot property. However, all queries generated by the tile plots of the horizontal viewport completely adhere to the WMS standard. In contrast, we had to extend and adapt the set of parameters offered by the WMS standard in order to realize plots of vertical cross sections through the data, and to realize data plots (which are currently only used for the computation of trajectories).

TilePlots

All TilePlot objects divide their visual representation into one or more Tile objects, each of which is responsible for the storing and for the invocation of a single WMS request. If a request was successful, a PNG image is returned by the server and stored in the Tile object. All tiles are visualized by the corresponding viewport object.

The set of all Tile objects of a plot, along with the section of the WMS request regarding the geometry of each tile, is created by the createTilesForPlot function of the Viewport interface. This function is called directly by the TilePlot implementation. The number and geometry of the created tiles depends on the current display settings of the viewport. It also depends heavily on the type of the corresponding viewport, which is the reason why this function is realized as part of the viewport classes and not as part of the TilePlot implementation. The section of the WMS request containing all DIM_* parameters is created straightforward by the getWMSQueriesFromProperties function, which is defined as part of the Property base class implementation.

The different classes implementing the Viewport interface handle the creation of tiles in the following ways:

- The horizontal viewport (ViewportCylindrical) subdivides the currently visible longitudinal and latitudinal range based on a fixed image resolution R of each square tile (we choose $R := 512$ in our implementation) and a zoom level $z \in \mathbb{N}_0$. The final tiles cover a square area with an edge length of $180 \cdot 2^{-z}$ degrees, which means the maximum area of a single tile covers $180° \times 180°$. The zoom level z is calculated as $z := \lfloor \left(\log_2 \frac{180 \cdot r}{R}\right) + 0.5 \rfloor$, with r being the current pixel-to-degrees ratio of the viewport, such that $180 \cdot r$ is the number of pixels in the viewport corresponding to $180°$. For any non-privileged user, the maximum allowed zoom level is $z = 3$. Such a limit is required for an efficient precalculation of tiles, as discussed in Sect. 4.5.1.
- The vertical viewport (ViewportVertical) creates only a single tile per plot. The geometry of the cross section is passed on to the server by means of the BBOX parameter, which contains the horizontal coordinates of the start and end point of the cross section. The height limits are given by two custom GET-parameters, VERTLOW and VERTHIGH. Furthermore, the vertical cross section supports four different types of interpolation, represented by the additional CSTYPE parameter. The corresponding values are linear (linear interpolation), ns (linear interpolation in longitudinal direction, ignoring the longitude coordinate of the end point), ew (linear interpolation in latitudinal direction, ignoring the latitude coordinate of the end point), and finally gc (interpolation along a great circle on the earth through the start and end position).
- The separate viewport (ViewportSeparate) also creates only one single tile with the full size of the dialog window that displays the plot. This size is currently fixed to 512×512 pixels. The BBOX parameter is not used in the WMS requests of these separate plots.

Tile plots optionally store one additional plot image containing the legend of the plot. This additional image always covers the full size of the viewport. The corresponding WMS request is generated by the TilePlot objects using just the getWMSQueriesFromProperties function. The only required external information is the resolution of the viewport.

Data plots

The data plots implemented in the `DataPlot` class are using a modified set of WMS parameters to obtain the required data for the client-side visualization. The request string is created as part of the `DataPlot` implementation, using the `getWMSQueriesFromProperties` function for the creation of all `DIM_*` parameters. Since the only current module for the creation of data plots, the trajectories module, requires no information about the viewport, the WMS parameters `BBOX` and `CRS` are excluded from the WMS request. Additionally, the creation of the plot data is independent from the resolution of the viewport, so the `WIDTH` and `HEIGHT` parameters are omitted as well. The `REQUEST` parameter is set to the nonstandard value `GetData`. Instead of an image file, the server returns a JSON representation of the computed data.

4.4.7. Serialization of program states

IWAL supports the storing and restoring of program states of the WMS client. The information about a state is stored persistently in the database of the IWAL server. Program states are shared between all users of IWAL which are working with the same case study on the same server. Every user can restore these shared program states, but only certain users can create new states and update or delete the states they have created. All users that are intended to be shared by multiple real users (i.e., users with their `can_edit_itself` flag set to "false" in their profile, see Sect. 4.3.4), are limited to the restoration of existing states.

Representation of program states

The singleton `SerializationController` instance is responsible for the management of all program states on the client-side. For the creation of a new program state, three parameters are passed to the server by the corresponding function of the `SerializationController` object: The `name` of the program state chosen by the user, a short description (`desc`) of the program state (currently unused), as well as a large JSON-encoded string containing all important information about all viewports, plots, and global settings of the client (`state`). This information is gathered by the `SerializationManager` object. For the creation of the `state` string, a single local JavaScript object containing all required information is created, which is then encoded into JSON and passed to the server. The server stores all this information, and some additional information (the user who created the state, the date of the creation, as well as the current case study and the current user mode) in its database.

The JavaScript object for the creation of the `state` string contains the following elements:

- The `globalOptions` member variable contains an object representing the states of all global IWAL options, for example global drawing options such as `drawLineShadows`, the `viewportSizeMode`, and the `autoRecompute` flag.
- The `viewportStates` member variable contains an array of objects describing the states of each individual viewport. A viewport state contains the following information:

- The `name` of the viewport as a string, currently one of "cylindrical", "vertical", and "separate".
- The `state` of the viewport properties in form of a dictionary which maps property names to the `_internalState` variable values of each property. The information stored in the `_internalState` variable of a property is a JSON-serializable object containing the complete current state of the property.
- An array containing all `plots` of the viewport. Each plot is represented through these attributes:
  - The `nameTemplate` of the plot, i.e., the name with possible placeholders that are later replaced by corresponding property values.
  - The `serverInfos` of the plot. This object contains all information about the module name, the map and legend layer names, as well as the viewport type the plot belongs to.
  - The `selected` flag of the plot.
  - The `visible` flag of the plot.
  - The `properties` of the plot in form of an array that contains a serializable object for each property. These objects contain the name and type, as well as the value of the `_internalState` variable of each property. In contrast to the representation of the properties of the viewport, the storing of the additional type string is required. During the restoration of a state, the function for creating new plots dynamically instantiates all properties using this type string.

Sharing of program states

IWAL shares the serialized program states along all users of the same case study on the same server. For this, the client application has to be informed about all available saved states. This information is subject of asynchronous changes, since users can create new states or change and delete existing saved states at any time. While the technique for sending asynchronous requests from the client to the server, Ajax, is easily implementable and widely used throughout the WMS client application, the other direction, an asynchronous communication from the server to the client, is not as easy to realize. Several approaches to solve this problem exist, commonly using long-lasting HTTP connections to the server, with the server postponing the response until an event of interest for the client occurs. These techniques for realizing asynchronous server to client communications are known under the term *Comet*[43].

In HTML5, a new API called *WebSockets*[44] was introduced, allowing asynchronous communication in both directions between the client and the server using the WebSockets protocol[45].

[43]This term was coined in the article "Comet: Low Latency Data for the Browser" by Alex Russel, 2006. See `http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/`, last accessed: 22-February-2013.

[44]see `http://dev.w3.org/html5/websockets/`, last accessed: 22-February-2013

[45]see `http://tools.ietf.org/html/rfc6455`, last accessed: 22-February-2013

Unfortunately, Django neither provides direct support of Comet nor of WebSockets in their current implementations, so we decided to fall back to a simpler approach. We realize the update by means of periodically recurring Ajax requests, which is possible, albeit not optimal, in our case, since the updates are not very time critical.

In order to minimize the amount of communication overhead, the client only queries a global *revision number* of all saved states from the server and compares it to the result from its previous query, if available. On the server-side, the revision number is stored as part of the `casestudy` entities in the database and is incremented every time a state is added, modified, or deleted. The revision number is available via the `/get_saved_states_revision` view. If necessary, the client triggers an update of the information about the currently available saved states of the current case study. This update is realized through the IWAL server's `/get_saved_state_names` view. This view takes two parameters defining the start index of the first saved state to be returned, as well as the number of states to be included in the response. For every state in the given range, the name, the creation date, the ID, and a flag indicating whether or not the state was created by the same user who initialized the query is returned.

The client has a fixed maximum number of saved states that can be displayed at a time (currently four). If there are more states available, additional buttons for the navigation through the complete list of states are shown. Every user can see the name and the creation date of the displayed saved states. A state can be restored by clicking on it. This triggers an additional query for the JSON string containing the full description of the program state, using the `/get_saved_state_data` view. If a state was created by the currently logged-in user, the client displays additional buttons for updating the state ( ), which means that the state is overwritten with the current state information, and for the deletion ( ) of the state.

4.5. Results

In the previous sections, the architecture of both the server and the client of IWAL were described in detail. The discussed concepts, techniques, and implementations build the expandable foundation of IWAL. Based upon this foundation, custom IWAL modules for the creation of the final plots can be implemented and integrated into the software. For each IWAL module, a set of plot options can be defined, whose number and variability can freely be chosen. The deployed version of IWAL that runs on the server at `http://iwal.ethz.ch` already provides a set of several IWAL modules for the production of a range of different plots.

In this section, we complete the description of IWAL by focussing on the achieved performance and on the usability of IWAL. Since the production of a plot image from scratch can consume a significant amount of computing time, it is necessary to rely on caching systems that are capable of serving already computed plot images in a fraction of the original computing time. Such caching systems can only operate efficiently, as long as the plot options and the covered geographic areas of each plot are constrained to a limited set of allowed values. In Sect. 4.3.2, the techniques and software tools used by IWAL for the caching of plot images were introduced. Section 4.5.1 covers a discussion of the trade-off between the available plot options and the required space and precomputation time for caching all pos-

sible plots. In addition, the effects of the different caching options on the responsiveness of IWAL, taking different numbers of simultaneous users into account, are measured in a series of benchmarks.

IWAL was already used in class by a group of students in the context of the Weather Discussion event at the ETH Zurich in the spring semester of 2012. During the event, the students were asked to share their impressions of IWAL by means of a user survey that was prepared and carried out by Michael Sprenger and Heini Wernli. The main results of this survey are presented at the end of this section, in Sect. 4.5.2. The results show some of IWAL's strengths, as well as potential aspects for further improvements. Many of the suggestions of the first users of IWAL were already taken into account during the later development of IWAL, other aspects still need to be addressed. This leads over to the final main section of this chapter, Sect. 4.6, where a closing discussion of further improvements of the software can be found.

4.5.1. Performance and Benchmarks

As discussed in Sec 4.3.2, IWAL uses two different caching methods to decrease the amount of time the server needs for replying to a single WMS request. The first method realizes a caching of plot images[46] in the server's RAM through Django's caching framework. Internally, Django can be configured to work with different software back ends for the caching. On the `iwal.ethz.ch` server, Django is configured to use the memcached software tool operating on two gigabytes of RAM. The second caching system is a custom implementation for additionally storing the plot images in the server's file system.

Limitation of plot options

The additional efforts necessary for caching can only pay off in cases where single plots are requested multiple times. Since each individual set of WMS parameters leads to an individual response, caching is not feasible if combinations of parameter values are unlikely to occur multiple times. Even a single parameter that can take on arbitrary floating point values, for example as part of the `BBOX`-WMS parameters, renders our caching strategy useless.

In order to make the caching work, we need to restrict the possible WMS parameters to finite sets of possible parameter values. The number of possible plot images can then be computed as the product of the cardinalities of all these sets. The limiting factors for the number of plot options are the available time for the precomputation of the plot images and the available space for storing the plot images on disk.

The limiting of the plot options is not always a trivial task, and certain modules require a high flexibility of their options in order to remain useful for the users. Because of this, some types of plots are completely excluded from caching. All vertical cross section plots are examples for this, because the location of the cross section cannot easily be constrained in a reasonable way. The corresponding modules may have arbitrary plot options, since for each request they always have to generated the plots anew. This means, for example, that

[46]The caching is currently only applied to image data.

the coloring scheme and the size of the resulting image of all vertical plots can completely freely be chosen.

In contrast, the location of the horizontal cross sections can easily be constrained to levels of equal pressure for a limited number of pressure values. The current modules typically allow pressure values ranging from 50 hPa to 1000 hPa in steps of 50 hPa. In addition, each level is divided into equally spaced tiles, see the description in Sect. 4.4.6 for details. This subdivision of the horizontal plots into a limited number of tiles is the prerequisite for the caching of these plots.

Most plot options, for example the time step and the data variable for the visualization, require no further adaption for caching, because they already take on only a manageable amount of possible values. One substantial exception is the plot option regarding the coloring scheme of the filled horizontal cross section plots. The limiting of this option requires a more involved strategy, in order to avoid a too negative impact on the plot quality. The coloring scheme determines the mapping from data values to colors. A suitable mapping for the production of useful plots depends on factors such as the displayed variable, the elevation, and the season. However, giving the user full freedom in the determination of the coloring scheme would foil any of our caching efforts.

The solution to this problem is an automatically varying coloring scheme that is controlled by each module on the server-side. For this, the server maintains a two-dimensional table, containing default color schemes for different combinations of variables and elevations. If the target elevation is not present in the table, the color values are interpolated. Another requirement of the coloring scheme is to prevent nonuniform decimal fractions in the visual representation of the color bar legend. In order to ensure this, the algorithm optionally adapts the range of the coloring scheme to create uniform digits after the decimal point at the known and fixed subdivision points. The automatic coloring does not take into account the date of the plot. To cope with this, a plot option allows the user to optionally restrict the range of the coloring scheme to the lower, the middle, or the upper third of the whole range. Because of the possible expansion of the range for ensuring uniform decimal fractions, the final interval does not generally cover exactly one third of the original, unaltered range.

Precomputation of plots

In this section, we want to estimate the amount of time and space required for the precomputation of all the plot images that are accessible through the restricted options offered by the IWAL WMS client for a typical module of a single case study. Such a precomputation becomes reasonable for case studies that are accessed by a large number of users at the same time, as it is usually the case when IWAL is utilized in class. We exemplarily analyze the `HorizontalCrossSection` IWAL module of the “Default” case study of the IWAL instance running at `iwal.ethz.ch`. We consider the plot options of a non-privileged user and take into account the limits on the zoom level and the subdivision scheme provided by the IWAL client. This leads to the following set of plot options:

- **Field**: In the default case study, there are three data variables available (T, PV, and Q).
- **Date**: The default case study covers four time steps.

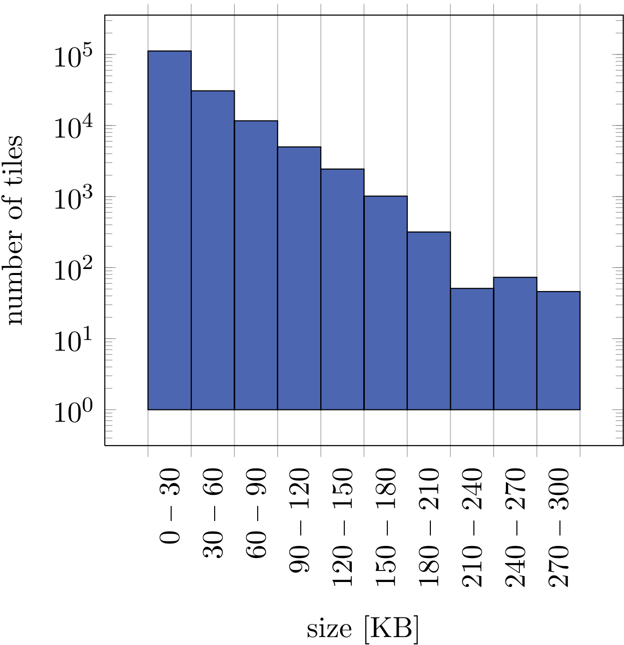


Figure 4.7.: Histogram of the file sizes of all cached tile images of the case study used for the benchmarks.

- **Level**: There are 20 possible elevations, ranging from 50 hPa up to 1000 hPa in steps of 50 hPa.
- **Color domain**: This property has four possible settings (full range, upper third, middle third, and lower third).

The client allows four different zoom levels. On the first level, the whole geographic range of $([-180°, 180°], [-90°, 90°])$ is subdivided into two square tiles. On each additional zoom level, the number of tiles grows by a factor of four. In total, we have 170 tiles and 960 possible combinations of user options, resulting in 163200 plots to be precomputed.

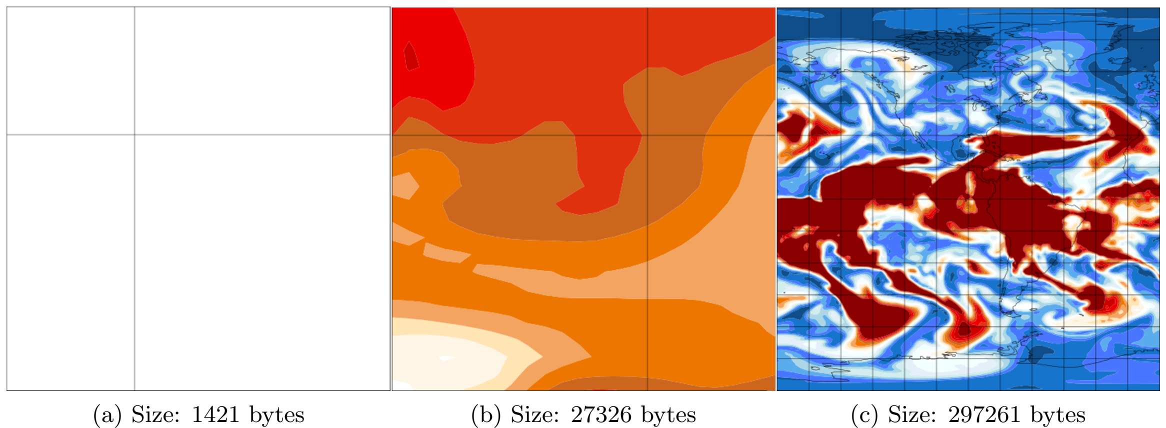
We precomputed the set of tiles on the IWAL server, officially named *IAC-IWAL*[47], which is a virtual Red Hat server running on the VMware farm of the *Department of Environmental Systems Science* (D-USYS) of the ETH Zurich. The available hardware resources are four Intel Xeon L5640 CPUs running at 2.27 GHz, and a total of four gigabytes of RAM. The required computation time and disk space for the precomputation of the plots of each variable are shown in Table 4.4.

The differences in the required disk space have their origin in the different applicability of image compression to the resulting plots. In general, the image sizes are very heterogeneous, ranging from 1421 bytes up to 297261 bytes. A histogram of all tile sizes can be found in Fig. 4.7. The mean tile size is approximately 27326 bytes, with a standard deviation of $\sigma \approx 31718$ bytes. In Fig. 4.8, tile images with minimum, maximum, and mean file size are shown.

[47]IAC stands for *Institute for Atmospheric and Climate Science.*

	computation time	disk space (PNG images)
T	1 h 40 m 43 s	1,301,640,941 bytes
PV	1 h 39 m 58 s	988,497,486 bytes
Q	1 h 49 m 15 s	2,169,399,694 bytes
total	5 h 09 m 56 s	4,459,538,121 bytes

Table 4.4.: Time and space requirements for the precomputation of all 163200 plot tiles of the `HorizontalCrossSection` IWAL module as part of the "Default" case study on the `iwal.ethz.ch` server.



(a) Size: 1421 bytes (b) Size: 27326 bytes (c) Size: 297261 bytes

Figure 4.8.: Comparison of cached tile images of different sizes.

Benchmarking of WMS requests

We additionally performed a series of benchmarks on the IWAL server in order to measure the influence of varying numbers of simultaneous users in combination with different caching techniques on the server's response time. The setup of the benchmarks was as follows:

A script running on a client generates sequences of simultaneous WMS requests and sends them to the IWAL server. It uses multiple threads in order to simulate the activities of multiple users. Each thread initially establishes the required connections to the IWAL server and then performs the following steps in a loop for a fixed number of times:

1. Wait for a random amount of time between zero and three seconds.
2. Generate four random WMS queries taking into account the constraints on the plot parameters and the viewport.
3. Send these four requests simultaneously to the server using four sub-threads.
4. Wait until all requests are done.

Finally, the connections to the IWAL server are closed. Using this setup, we measure the required response time for each complete group of four requests. These groups simulate the way in which the WMS client application performs single updates of a tile plot consisting of multiple tiles.

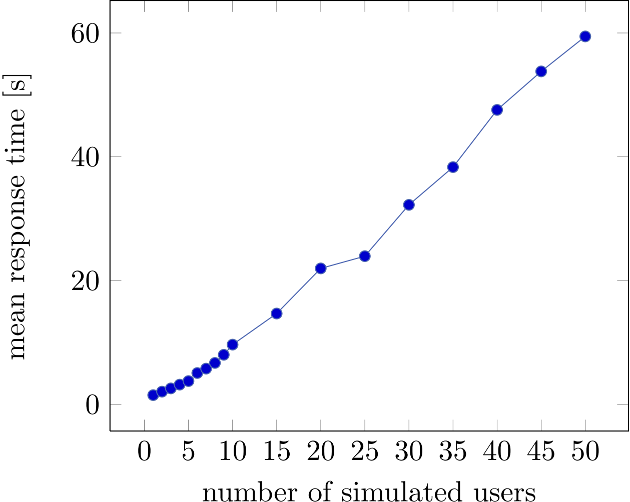


Figure 4.9.: Mean response time for a group of four simultaneously issued WMS requests without any server-side caching. An increasing number of simultaneous users was simulated.

All benchmarks were performed using requests to the IWAL server at `iwal.ethz.ch`. The target module was the `HorizontalCrossSection` module using the input data configuration of the "Default" case study and the non-privileged user mode.

For the first benchmark, the server-side caching was completely disabled. So for each request, the returned plot data had to be created anew by the server. The script simulated an increasing number of users from one up to 50. For each simulated user, the previously described loop for simulating a plot update consisting of four WMS request was repeated twenty times. The response time of each plot update was measured and the mean time over all updates of each number of simulated simultaneous users was computed. The results can be found in Fig. 4.9. The mean response time for one simulated user was 1.49 seconds, whereas for 50 simultaneous users, the mean response time significantly increased to 59.44 seconds. Our measurements show, in the analyzed range, a linear dependency between the number of simulated users and the mean response time.

We installed the two caching systems of IWAL (caching of plot images on disk and in RAM using memcached) with the goal of significantly lowering the time required for each WMS response. In order to verify the achieved improvement, we performed additional benchmarks with the same setup as described above, but with enabled caching on the server-side. The full set of precomputed plot images was available on disk. The memcached memory was initially filled with a random subset of plot images from the disk cache. We ran two series of benchmarks, one using disk-caching only, and another one with both activated disk-caching and enabled memcached. Due to the faster response time, we increased the maximum number of simultaneous users to 100. The results of both benchmarks are visualized in Fig. 4.10.

As expected, the measured response times were much lower than the corresponding times achieved without any caching. The benchmarks with enabled memcached caching performed constantly better than the benchmarks without memcached. The percentage of memcached cache hits in the benchmark with enabled memcached was 36.31 %. Globally, the response

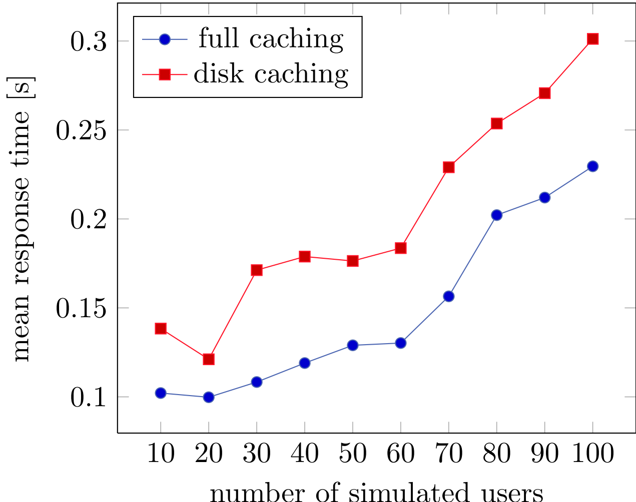


Figure 4.10.: Mean response time for a group of four simultaneously issued WMS requests with enabled server-side caching. An increasing number of simultaneous users was simulated.

times of both benchmarks increased along with the number of simultaneous users. However, there were also some local minima recognizable, which can be explained by slight differences of the plot image sizes causing varying transmission times due to the randomly generated requests. In addition, the complex interactions between the Apache web server and the mod_wsgi software led to "noise" that is recognizable in the measurements of such low response times.

The performed benchmarks show how the response times of the WMS requests benefit significantly from the caching strategies provided by IWAL. However, the necessary precomputations of plots require a considerable amount of time and disk space resources. Some modules are, at the moment, generally excluded from caching. Many of these modules, for example the module for the calculation of trajectories, require considerable amounts of computational power to create their plot data. New strategies need to be developed in the future in order to make working with these modules feasible for large numbers of simultaneous users.

4.5.2. User survey

IWAL's first real application in class was during the Weather Discussion event at the ETH Zurich in the spring semester of 2012. The students of this event were the first persons not directly involved in the production process of IWAL, which had a chance to work, test, and evaluate IWAL. In doing so, they provided valuable help by reporting bugs, testing the performance of the server, and by suggesting improvements of the software.

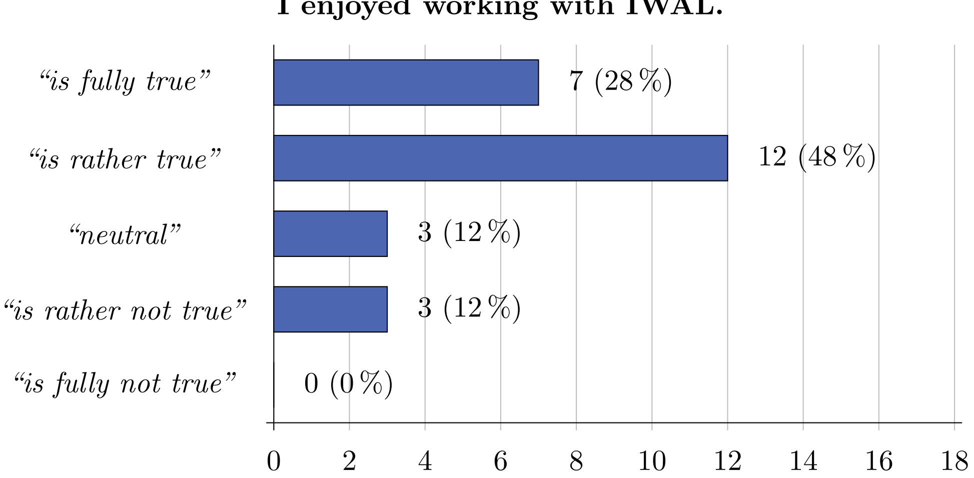
As part of this evaluation process, a user survey was organized and carried out by Michael Sprenger and Heini Wernli. The questions of this study were prepared by Michael Sprenger, Heini Wernli, and by myself. The participants were confronted with a set of statements, which they could judge by choosing one out of five possible responses ranging from *"is not true at all"* up to *"is fully true"*. In addition, the students were asked to name the good aspects and the problematic aspects of IWAL from their experience in working with the

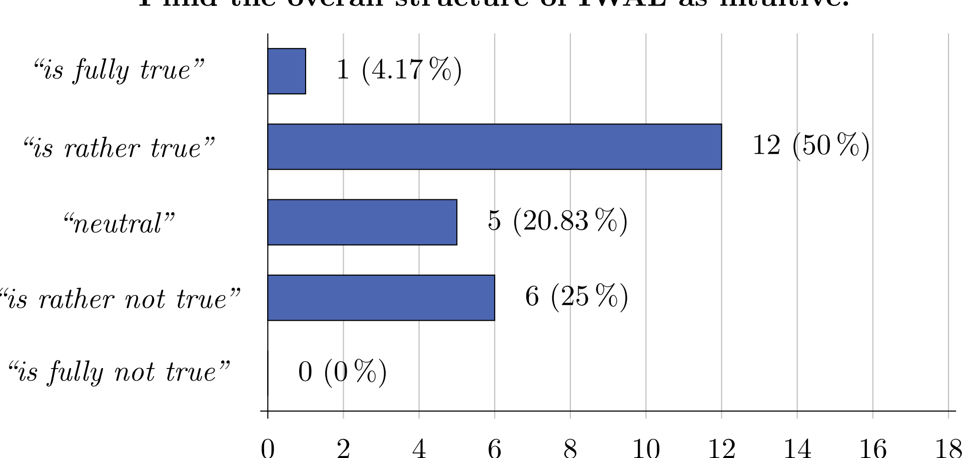
software. The results of the survey showed that the software reached many, but not all of the goals we aimed for during the planning and the development of IWAL. Fixes for the observed bugs and most of the proposed features are already incorporated into the current version of IWAL.

Judgment of given statements

We examine the results of the given statements of the survey which were judged by the students. The statements can be grouped into those targeted at the technical aspects of the software, and those with focus on the didactical usefulness of IWAL. We will present the responses for each statement, together with a short discussion and estimation of the results.

The following two questions indicate that working with IWAL is perceived as intuitive and enjoyable by a majority of users:



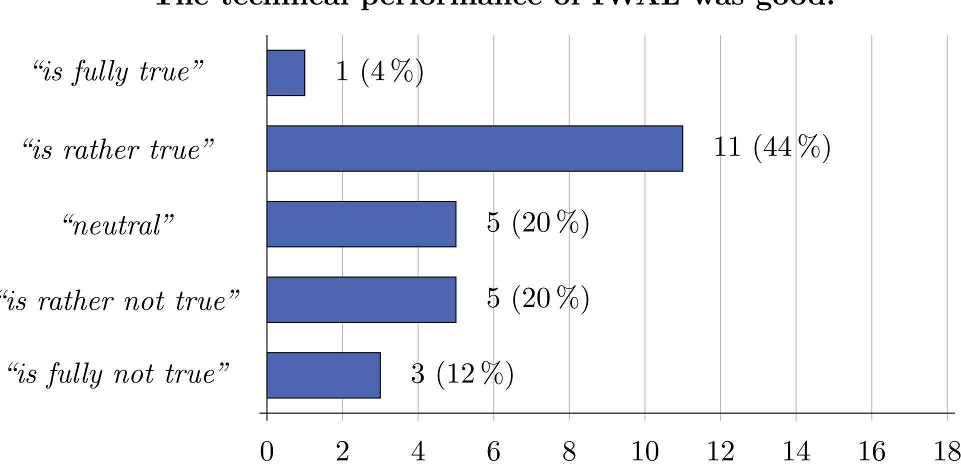


Taking into account the complexity of the software and the degrees of freedom a user has in the production of the plots, we think that this is a fairly good result. Nevertheless, numerous

possibilities and ideas exist for further improving the user experience and the ways in which the users can interact with the software.

The design of the user interface and the responsiveness of the software are, in our opinion, important aspects for the overall user experience. As mentioned above, we already improved the performance of the application in response to the feedback we got from the survey. Nevertheless, we propose in the sections below some additional concepts for the further reduction of the response times of the server. Aside from the performance, other improvements of the user experience could include a better documentation, an online-help, or an interactive tutorial aimed at new users of IWAL.

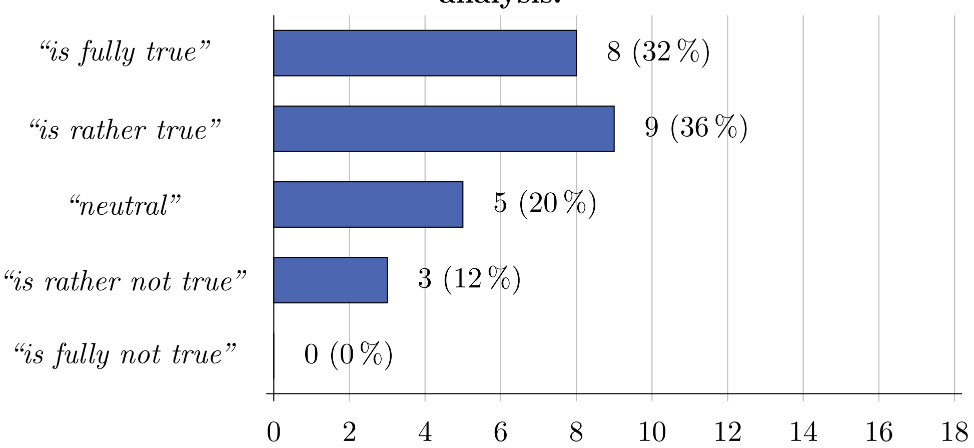
The next question with a technical focus is about the overall performance of IWAL:

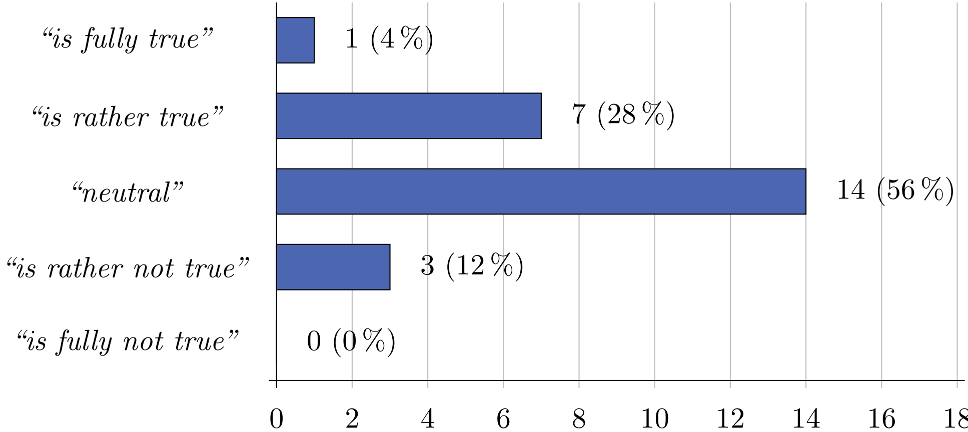


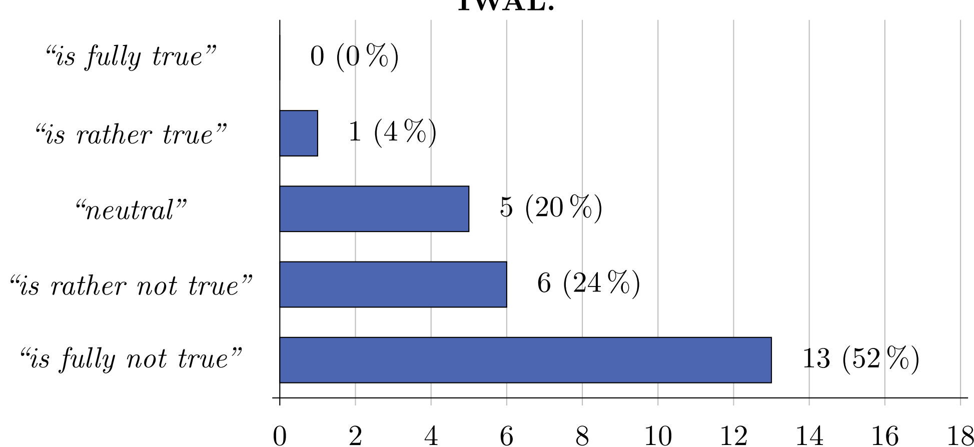
Although a majority of users rated the technical performance of IWAL as neutral or good, the negative ratings of 32 % of the students clearly indicates that the performance of the software needs to be improved. A more specific investigation would be useful in order to distinguish between the server/UI responsiveness, and the quality of the plots and data created by the IWAL modules. Our tests and benchmarks from Sect. 4.5.1 show the impact of precomputations and caching on the performance. In practice, more tiles could be precomputed in order to increase the performance of the server. Another way to improve the performance would be the deployment of IWAL on multiple servers or an improvement of the provided hardware resources of the current server.

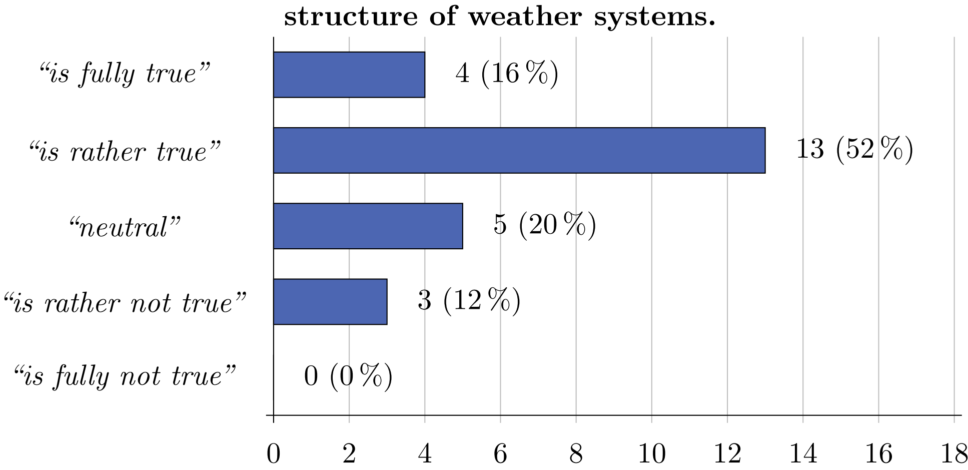
The plot images leave also room for improvements, see for example the discussion of the colored cross-section plots in Sect. 4.6. As it was the case for the questions above, one needs to take into account that many criticized aspects have already been fixed in the current version of IWAL.

The last set of questions aims more at the educational and didactic qualities of IWAL:









These mixed answers show that there is still room for improvements regarding the educational and didactic qualities of IWAL. However, the development of new educational concepts and their implementation in form of IWAL modules is out of the scope of this thesis. Here, we focus more on the technical foundation and the correct implementation of the existing features of IWAL.

One goal for the future should be the improvement of the quality of the scripts used for the production of the plots. In addition, many potential technical improvements and new features are conceivable that could help the students to develop a deeper understanding of the presented meteorological data. A detailed list of concrete ideas can be found in Sect. 4.6.

Negative aspects

Finally, we present the mentioned negative and positive aspects of IWAL, as well as some ideas for improvements mentioned by the students.

One of the most criticized point was the server's response time for loading and updating the plots (e.g. *"sometimes it took long to load the plots"*). Our benchmarks showed, that most of these performance issues could be eliminated through more comprehensive precomputations of plot tiles. Other possibilities would involve improvements of the scripts used for plotting, probably together with a complete replacement of the plotting back ends (which are, at the moment, mostly PyNGL and PyNIO, see Sect. 4.2.3). An additional issue that was mentioned by multiple students was the often insufficient quality of the automatic chosen color ranges (e.g. *"colorbar does not adjust when zooming to smaller region"*, *"improve contour setting, e.g., for variables that go over a large range like liquid water content"*). These issues could probably be solved by tweaking the tables that form the fundament of the automatic coloring, possibly taking into account the season as an additional parameter. One general problem regarding the coloring is that we cannot color individual regions of a zoomed-in tiled plot differently. This is because we have to be able to precompute the single tiles and the user can freely choose the visible section of the plot. Because of this, any discontinuity in the coloring scheme would be visible if the user navigates to the corresponding image section. An approach to fix all of these coloring issues would be a client-side coloring of the

plot data. However, the achievable performance of such a solution is unknown, the server requests could no longer adhere to the WMS protocol (unless the raw data is encoded using a suitable image format), and the implementation would require an additional restructuring of essential parts of the IWAL client.

The remaining remarks of the students were mostly proposals for additional plot features (*"units are missing in the plots", "would be good to have a height axis in the vertical plots", "would be nice to produce plots with wind vectors", "the plots don't have titles so you don't know what is plotted afterwards"*), as well as some requests and remarks regarding the user interface and the documentation (*"users guide would be useful", "would be nice to login with nethz-account", "would be good to have a pop-up saying "loading" during the loading phase", "possibility to produce animations and save them"*).

Positive aspects

For the sake of completeness, we present the following list which contains the mentioned *positive* aspects of IWAL. Similar statements are gathered into one statement, the number of mentions of each statement is specified in the parentheses at the end of each point.

- Variety of alternative weather maps, opportunity to look at different parameters (4)
- Access to past time data (2)
- Gives a wide range of possibilities to explore different aspects of weather systems and to "play around" (2)
- Interactive working and instantaneous production of the plots (5)
- Very intuitive understanding
- Zooming in and out is easy (2)
- Good performance
- Nice vertical cross-sections (4)
- Nice coloured plots (4)
- Easy to use (3)
- Vertical cross-sections and corresponding horizontal plots side by side (2)

4.6. Outlook

In Chapter 4, we presented a detailed discussion of the technical concepts of IWAL and their practical realizations. We analyzed the performance of the software and looked into the results of a user survey. The software was already used in class in the Weather Discussion event of the spring term of 2012, and is used again as part of the same event in 2013.

There are a lot of possible extensions and improvements that could be integrated into the upcoming versions of IWAL. The feedback from the user survey already contained several good ideas, many of which were already integrated into the current version of the software.

The following list contains additional possible future enhancements of IWAL:

- We have to improve the visual quality of some of the plots produced by the IWAL modules. Currently, the automatic coloring (cf. Sect. 4.5.1) often leads to poor results. A fine-tuning of the specific coloring ranges could lead to improvements. However, there are additional problems in the quality of the visualization, for example regarding the borders of tiles and the transparency and labeling of the color bar legend, that have to be addressed.

- Another idea to overcome the problems that are currently present in the automatic coloring would be to transmit the raw variable data to the client instead of the final image, and to visualize this data on the client-side. This would allow the user to freely choose and change the coloring scheme without the need of additional WMS requests. The drawbacks of this method would be the more complicated handling of raw data in contrast to image files (we would, for example, need to explicitly compress the data for transmission) and the increased computational effort on the client-side. A possible solution could be the encoding of the raw data into an image of a suitable image format, which is not displayed directly, but filtered by the client in order to apply the custom coloring scheme.

- IWAL should allow the user to probe the shown data values at single positions interactively. In order to integrate a intuitively usable probing mechanism seamlessly into the existing structure of IWAL, we would need to extend the basic structure of an IWAL module on the server-side, as well as the handling of mouse or touch events on the client-side. The reason for not implementing a separate probing IWAL module is that it is not possible to keep the properties of such a module in sync with the topmost visible plot of the respective viewport.

- Currently, IWAL supports only a single type of map projection. In the future, additional types of projections, for example stereographic projections, could be added to IWAL.

- IWAL uses WMS conform queries for the creation of horizontal plots. The vertical plots and data plots require some additional parameters that are not part of the WMS standard. At the moment, IWAL uses its own set of extended parameters for the realization of these plots. Other software developers propose different solutions for the creation of their tools. For example, Matula (2009) introduced the usage of custom values for the `CRS` parameter, e.g. `IBL:SKEWT`, to indicate the request of a nonstandard diagram. It would be desirable to have an extended WMS standard, or a common set of custom extensions, which permit the definition of such queries.

- At the moment, IWAL does not use the optional WMS parameters `TIME` and `ELEVATION` for the selection of the time step and the model level of a plot. We should evaluate how this affects the interoperability of the IWAL server with third-party WMS clients. If it is reasonable to support the optional `TIME` and `ELEVATION` parameters, we should adapt our implementation accordingly.

- IWAL would benefit from the possibility to include images provided by third-party WMS servers not running IWAL. Such an inclusion could be realized through an additional IWAL module with the option to enter the URL of such an external WMS server.

- A possible major addition to IWAL would be the implementation of three-dimensional data visualization techniques, similar to those of Insight. The most modern web browsers support hardware-accelerated 3D graphics directly via *WebGL*[48].

The development of IWAL can be observed on the IWAL project page, which is currently still hosted on the Insight server at `http://insight.zdv.uni-mainz.de/iwal/trac`. The running IWAL client software is accessible at `http://iwal.ethz.ch`.

[48]see `http://www.khronos.org/webgl/` last accessed: 04-April-2013

5. Outlook and Conclusion

In this thesis, we presented two software tools, Insight and IWAL, for the processing and visualization of atmospheric data. In addition, we introduced a new segmentation algorithm implemented as part of Insight, for the detection and tracking of three-dimensional atmospheric phenomena and their development over time. Although, at a first glance, the three presented tools seem to be diverse, they all share the common goal of generating new insights into large sets of data. All individual tools had to overcome similar problems with the formulation of suitable and efficient algorithms, taking into account the limitations and restrictions resulting from these large amounts of data.

The common link between the three main chapters of this thesis is *Insight*, the software tool presented in Chapter 2. The novel segmentation algorithm (Chapter 3) is completely implemented as part of Insight. For IWAL (Chapter 4), we not only adapted many software design principles directly from Insight, we also used an Insight-based WMS server for the production of the first plots of IWAL during development. Currently, no direct connection between IWAL and the segmentation method exists. In the future, however, IWAL could serve as a web interface for the segmentation algorithm, by providing a module for the visualization of the segmentation results, similar to the two-dimensional plots for the cyclone studies (cf. Fig. 3.11).

5.1. Achieved goals

Insight combines visualization and data processing features in a flexible and modular way, offering new methods for the analysis of large sets of atmospheric data to the user. A core feature is the possibility to freely map data variables and their dimensions of varying shape from multiple sources to different data processing and visualization units. The interactivity and the adaptability of the software are unique features of Insight in its specific area of application. The fruitful cooperations with other scientists and the many example applications of Insight are evidence of the software's advantages over comparable tools regarding many different tasks. The application of Insight was not restricted to the work presented in this thesis – due to its flexible architecture, we were also able to use Insight over the last years for a series of quick visualization tasks, as a tool for the batch-processing of large data sets, and for converting data from different sources into a common format.

During the development of Insight and for the formulation of the segmentation algorithm, we had to overcome a lot of shortcomings and problems related to the sheer amount of data to be processed. We had to develop new algorithms specifically designed for handling these large amounts of data. For very large data sets, reading in the data from disk in a single iteration over all available files and time steps already consumes a considerable amount of time on common hardware. The limited amount of available memory additionally prohibits

a random access of the data. These facts had a huge impact on the creation and optimization of our algorithms.

The new segmentation algorithm presented in Chapter 3 is a good example for avoiding both problems by using appropriate data structures and only a single iteration over the input data. Besides its efficiency, we demonstrated the usefulness of the algorithm through different examples of application: a climatology of jet streams and their events, a case study of storm Kyrill showcasing the segmentation of cyclones, and finally a five-years cyclone climatology on several different height levels. We presented additional techniques for reducing the over- and under-segmentation of the data.

In Chapter 4, we presented the IWAL project. Besides the general challenge of handling large data sets, IWAL, as a web application, caused further problems we had to overcome. For example, in the face of limited computational power of the server and limited network bandwidths, we had to balance the available user options and the necessary effort for precomputations. Efficient means of caching were necessary in order to ensure fast server responses and the intended level of interactivity for the clients. A series of benchmarks and a user survey were carried out in order to evaluate the efficiency and quality of IWAL.

The tools discussed in this theses were utilized by other scientists already at very early stages of their development. This was a big advantage for the development process of all three tools. Such close cooperation enables an immediate discussion of new features, as well as direct feedback on any newly implemented functionality. This kind of development process forces the engineer to keep the software and algorithms flexible, modular, and extensible, and at the same time it helps to avoid mistakes already at early stages and to fit the needs of the target users more precisely.

On the downside, the increasing number of users led to a rising amount of requests for technical support and to rising numbers of reported compatibility problems due to the different hard- and software configuration of the users. Therefore, at least for the future of the two software tools, it will be important to address these technical and organizational issues as well. One of the minor goals of this thesis was to provide a useful documentation of the software projects targeted at three different audiences: people interested in the general concept and ideas behind the implementation, people interested in using the software, and finally people planning to work with the source code (e.g. maintaining or extending it).

5.2. Outlook

We presented several practical applications and example use-cases of our tools in this thesis. Some of these examples contained our own work, but most of them contained joint work with colleagues from several different institutions. All of these collaborations led to numerous new ideas and feature requests. Many of these ideas were already presented in the individual outlook sections of the three main chapters. Below, we summarize the next steps we consider to be most important for each of the three tools, and give some general outlook on future developments.

The most important extensions of Insight regard the stability and responsiveness of the user interface. These issues are currently the biggest roadblocks on Insight's way to a first post-beta release. A more transparent way to show the progress of extensive computations, and

a user option for canceling these computations are two examples of necessary extensions for achieving this goal. Some other possible extensions are additional visualization techniques, especially for the improved visualization of vector-valued data, the support of more file formats (e.g. HDF and newer netCDF versions), and the development of a non-GUI version completely controllable through Python commands and scripts. There are also ideas for improving the GUI and for adding the possibility of accessing and manipulating the visualized data directly in the three-dimensional viewports.

For the segmentation algorithm, the improvement of the event localization by incorporating a non-uniform growing speed of the seed regions could be one of the next major extensions. We also discussed several possible additions for an improved feature tracking. Feature segmentation is a complex task, and the formulation of the three selection criteria is generally non-trivial. In our experiments with the segmentation of several different atmospheric phenomena, we added multiple options and extensions to the algorithm. However, the plan for the future is to improve the techniques used for feature tracking and for the sub-segment assignment, such that we can drop the inferior approaches and finally reduce the number of user-controlled parameters again. For example, a wind field or pressure based displacement of the 3D features for their tracking could make the dilation of features obsolete.

In the conclusions of Chapter 3, we mentioned that the segmentation algorithm seems generally well-suited for parallelization. Parallel processing of data, either on multi-core CPUs or on modern GPUs, is one of the current trends in software development. However, we barely covered this topic in this thesis, although we experimented with the creation of parallel algorithms operating on meteorological data from the beginning of our work in this field. For example, we realized and benchmarked the computation of trajectories on the GPU, using CUDA (cf. Nickolls et al., 2008). This experiment resulted in only slightly improved running times with a factor of about two. The bottleneck was (again) the IO operations on the data. Even this moderate speed-up could only be achieved for a very large number of trajectories (up to seven million trajectories in parallel) using an elaborate scheduling of the computations that minimized the transfers of the large input data sets to the memory of the graphics hardware. We concluded that, at our current stage of development, and as long as we perform comparable simple operations on very large sets of data, the achievable performance gain by parallelization is not worth the increasing complexity of the implementation. In the future, however, especially for Insight, further experiments and extensions in this direction are planned.

At the end of this outlook, it remains to mention some possible future extensions of IWAL. The next steps of the development should focus on improving the overall efficiency of the program and the quality of the plots. The current visualization scripts still contain some minor bugs, for example regarding the visualization of the color bar legends and the automatic coloring of some variables of the filled cross-section plots. In the context of these problems, a client-side visualization of the data could be tested. If such a client-side visualization was efficient enough, the overall performance could increase due to the reduced server load. In addition, this principle would allow for an interactive probing of the data values at arbitrary positions on the client-side. Another interesting enhancement would be the inclusion of data from external WMS servers. For the future, we already discussed plans to offer IWAL to students of fields different from meteorology. In principle, IWAL modules can be implemented for any type of two-dimensional data visualization. In a more distant future, even a 3D data visualization through technologies such as WebGL could be possible.

5.3. Conclusion

In the course of the development of the tools presented in this thesis, we had the fortunate chance to experience at first-hand the impact of our software and algorithms on the work of many different scientists and students. It was a revelation to see users recognizing new aspects in their data sets, for example when they first used Insight to explore an interactive, three-dimensional visualization of it. Static and often two-dimensional visualizations are, for good reasons, well established in the meteorological community. However, adding the third dimension, increasing the level of interactivity, and making the exploration of large amounts of data more convenient, allows both to obtain a quick overview of the data, and to explore small details of interesting structures. By this means, the chances of finding interesting time steps and locations worth a further analysis are increased. The new segmentation algorithm aims at an automatization of the process of extracting interesting structures out of the large input data sets. IWAL allows for an even easier exploration of large data sets, eliminating the needs for local access to the data and for the installation, and often complicated handling, of local visualization tools.

We hope that our tools find many useful applications and that they inspire and enable people to explore and work with large sets of atmospheric data in useful new ways.

A. Installing and launching of Insight

A.1. Installing Insight

In this section, we provide step-by-step instructions for the installation of Insight. As it was the case for Insight's predecessor Vis3d, the implementation uses the Qt library and their included build tool `qmake` and it contains no platform-dependent code. Therefore, Insight can be compiled for Linux, Windows, and Mac OS operating systems. We developed and tested Insight using an Ubuntu Linux distribution, so the following instructions are focused on compiling and installing Insight on Linux.

We do not cover the installation of all required third-party libraries and software here, since this can differ a lot with respect to the target operating system. The following list contains the main relevant external software components and third-party libraries for which Insight was developed and tested. Required basic tools, such as GNU Make, are not explicitly listed. The distribution we used for most of the development was Ubuntu 10.04 (Lucid Lynx). All of the following components could be installed using the according packages (often also the development version) from the Synaptic package management system.

- GCC 4.4.3
- Qt 4.6.2 (including the WebKit, XML, and OpenGL modules)
- Python 2.6
- The Python and Assignment libraries from Boost 1.40
- NetCDF C and C++ library 4.0
- AMD-OpenGL proprietary FGLRX drivers 8.723.1 (supporting OpenGL 3.2 and GLU)
- Subversion 1.6.6

The source code of Insight additionally contains the full sources of the two libraries *GLee*[1] (**GL** **e**asy **e**xtension library) and *GLEW*[2] (Open**GL** **E**xtension **W**rangler Library).

Checking out the sources using SVN

The first step for installing Insight is to download the newest version of the source code from SVN.

```
:~$ svn export ⤸
    ⤷http://insight.zdv.uni-mainz.de/svn/insight/trunk insight-src
```

[1] `http://elf-stone.com/glee.php`, last accessed: 28-May-2013
[2] `http://glew.sourceforge.net/`, last accessed: 28-May-2013

The next step is to create the `Makefile` out of the Qt project file `vis3d.pro` using `qmake`:

```
:~$ cd insight-src/
:~/insight-src$ qmake
```

Finally, we are ready to build Insight using `make`:

```
:~/insight-src$ make
```

If some of the required third-party libraries cannot be found, it may become necessary to edit the `Makefile` and the `vis3d.pro` files manually. In addition, Insight recognizes the following two preprocessor flags, which can be activated and deactivated in the `vis3d.pro` file:

- `INSIGHT_QWEBVIEW_AVAILABLE`: This flag is set by default, but it can be removed for Qt versions not providing the `QWebView` widget. This widget is used for showing the help file containing the data entity reference.
- `INSIGHT_DISABLE_GL_2_0`: If this flag is defined, Insight disables all OpenGL features requiring an OpenGL version of 2 or higher. A version compiled with this flag lacks some important features, for example the support of different map transformations and advanced transparency calculations. However, these versions can easily be run through an SSH connection, or using a virtual framebuffer X server[3].

Installation of required files and directories

After the successful compilation, Insight can directly be invoked from the source directory. However, sometimes it is preferable to copy the executable and all required files into a clean program directory. This can be achieved by the following commands:

```
:~/insight-src$ mkdir ../insight
:~/insight-src$ mkdir ../insight/cache
:~/insight-src$ cp -R __* insight netcdf_formats open scripts
    Shader view ../insight
:~/insight-src$ cp -R --parents scripting_models/insight
    ../insight
:~/insight-src$ cp earth* ../insight
```

These commands create the target `~/insight` installation directory and the required subdirectories. All necessary and some additional files are copied to the installation directory. The optional last command adds to the installation a composition save file and the required data[4] for the visualization of a map of the earth.

At this point, Insight can finally be started from the new installation directory:

```
:~/insight-src$ cd ../insight
:~/insight$ ./insight
```

[3] see `http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml`, last accessed: 28-May-2013

[4] The data in the `earth.xml` file contains a converted version of the FreeGIS Worlddata v0.1 (simple) data set, see `http://ftp.intevation.de/freegis/worlddata/freegis_worlddata-0.1_simpl.tar.gz`, last accessed: 28-May-2013.

Updating the documentation

The Insight installation contains an HTML file (`__doc.html`) with a reference documentation of of all data entities. This help file can be shown in Insight by pressing `F1`. The HTML file was created using an Insight Python script, which is also included in the installation. For updating the reference documentation, this script can be invoked by the following command:

```
:~/insight$ ./insight scripts/__create_doc.py > __doc.html
```

A.2. Launching of Insight

Command line parameters

Insight divides the command line parameters into two groups: Parameters starting with a dash "`-`" and parameters starting with any other character. For the first group of parameters, the dash has to be followed by one or multiple characters from the set {h,i,q,s,v,?}, each of which represents one of the following options:

- **h**, **?**: A short **h**elp text is shown, all other command line parameters are ignored.
- **i**: All parameters following this switch are **i**gnored.
- **q**: All output of Insight is suppressed ("**q**uiet mode"). This does not affect the output generated by Python.
- **s**: Insight does not automatically terminate after the execution of a user script ("**s**tay open").
- **v**: All generated messages are output by Insight ("**v**erbose mode").

The first command line parameter not starting with a dash is interpreted as the name of a script file to be executed by Insight. All following command line parameters not starting with a dash are passed on to the script through the global `insight.params` variable.

Two examples: For starting Insight in "verbose mode" and redirecting the standard output into a log file with the name `output.log`, Insight can be started using these parameters:

```
:~/insight$ ./insight -vi > output.log
```

The following command executes a script file using Insight's "quiet mode" without quitting Insight afterwards:

```
:~/insight$ ./insight -q scripts/__create_doc.py -s
```

Extending the command line functionalities

The way in which Insight parses the command line parameters allows the user to extend the available functionalities through custom Python scripts located in the root directory of

Insight. Two examples for this technique that are included in the default installation are the scripts `open` and `view`.

The `open` script takes exactly one additional parameter representing the filename of an Insight XML save file. The script opens the given file directly on startup. The usage of this custom command is shown in the following example:

```
:~/insight$ ./insight open example.insight.xml
```

The `view` script simplifies the task of visualizing a single variable from a netCDF file. It takes two additional parameters: the filename of the netCDF file and the name of the data variable to be visualized. An example invocation looks as follows:

```
:~/insight$ ./insight view P20110820_12 T
```

B. Reference of the data entities of Insight

Insight, in its current version, provides 42 different data entities. In this appendix, we list all data entities, sorted by their primary function as either data sources, data processors, or visual representations. As discussed in Sect. 2.2.5, some entities fit into multiple categories, but in order to avoid duplications, we sort each entity into a single category here. Note, that the specified output variables, dimensions, and the properties correspond to the initial state of each data entity after its creation.

The following list contains all entities in alphabetical order:

2d-model data source (B.1.1)
Binary operation (B.2.1)
Box representation (B.3.1)
Caption (B.3.2)
Color combiner (B.2.2)
Color mapper (B.2.3)
Color mapper (ext. position) (B.2.4)
Color mapper 2D (B.2.5)
Color setter (B.2.6)
Color source (B.1.2)
Complete segmentation (4D) (B.2.7)
Contour lines (B.2.8)
Data adder (B.1.3)
Date mapper (B.2.9)
Euclidean vector norm (B.2.10)
Event graph representation (B.3.3)
Graticule (B.3.4)
Grid mapper (B.2.11)
Grid representation (B.3.5)
Ground (B.3.6)
Iso surface (B.2.12)
Iso surface representation (B.3.7)
Labeled axis (B.3.8)
Line strip representation (B.3.9)
Linear interpolation source (B.1.4)
Mean vertical profile source (B.2.13)
NetCDF cluster source (B.1.5)
NetCDF data source (B.1.6)
NetCDF time series source (B.1.7)
Point of interest (B.3.10)
Point representation (B.3.11)
Point volume representation (B.3.12)

Position adder (B.2.14)
Scripted operation (B.2.15)
Simple grid (B.3.13)
Statistics (B.4.1)
Trajectories data source (B.1.8)
Unary operation (B.2.16)
Unit converter (B.2.17)
Variable combiner (B.2.18)
Vertical profile source (B.1.9)
[wrapper] Iso surface (B.3.14)

B.1. Data sources

B.1.1. 2d-model data source

This entity reads in files representing 2D models defined as polygons consisting of vertices in the plane. The model has to be stored in a proprietary, XML-based format. The XML file consists of the tags <model> (spanning the whole file), <polygon>, and <point>. Each <point> has to provide two attributes, “x” and “y”, containing the coordinates of the point. The output is available as the single 2D variable “vertex” with the dimensions “polygon” and “vertex”. The output variable is associated with the respective position, while its value is always “1”. The z-coordinate of the position can be specified by the user using the “Z position” property. The polygons of the model are commonly interpreted as piecewise linear curves that can be visualized using a “Line strip representation” (B.3.9).

Output variables:

  vertex

Output dimensions:

  polygon, vertex

Properties:

- Model file [Python: `model_file`, type: *filename*]: The filename of the XML file containing the 2D model.
- Z position [Python: `z_position`, type: *double*]: The fixed z-position for all vertices of the 2D model.

B.1.2. Color source

This data source outputs a single, fixed, user-defined color. This color can, for example, be used as input for entities such as the “Iso surface representation” (B.3.7).

Output variables:

  color

Output dimensions:

Color component

Properties:

- Color [Python: `color`, type: *color*]: The fixed color to be output.

B.1.3. Data adder

This entity provides an arbitrary number of output variables with fixed, user-defined scalar values.

Properties:

- Added fields [Python: `added_fields`, type: *namedValues*]: This property manages a set of key/value pairs representing the names and the fixed scalar values to be added to the output.

B.1.4. Linear interpolation source

This data source provides a one-dimensional output variable whose values are the result of a linear interpolation. The size of the output dimension, as well as the values at the two interpolation points, are defined through corresponding properties.

Output variables:

value

Output dimensions:

dimension

Properties:

- Min value [Python: `min_value`, type: *double*]: The value at the lower interpolation boundary.
- Max value [Python: `max_value`, type: *double*]: The value at the upper interpolation boundary.
- Steps [Python: `steps`, type: *integer*]: The number of interpolation steps, i.e., the size of the output dimension.

B.1.5. NetCDF cluster source

This entity provides access to clusters of netCDF files with equally named and structured output dimensions and variables. For example, clusters of netCDF files containing trajectory data of one single trajectory per file can be processed by this entity. An additional file dimension is added to each output variable in order to provide a mechanism for the user to select the single files of the cluster. The supported netCDF file formats are equivalent to the formats recognized by the "NetCDF data source" (B.1.6). See also: "NetCDF time series source" (B.1.7).

Properties:

- NetCDF file [Python: `netcdf_file`, type: *filename*]: The names of all input netCDF files of the cluster.
- Format [Python: `format`, type: *selection*]: The netCDF file format, describing the mapping of the dimension positions to geographic positions. This selection property contains the names of all available netCDF formats which fit the available dimensions and variables of the current input. The set of available formats is specified through the present XML files in the `netcdf_file_formats/` directory of the Insight installation.
- Sort files alphabetically [Python: `sort_files_alphabetically`, type: *boolean*]: Controls whether or not the input files are sorted alphabetically based on their name.

B.1.6. NetCDF data source

This entity provides access to a single netCDF file. If the format of the netCDF file is recognized, all output variables are automatically associated with the respective grid and positional information. Several types of netCDF files are currently supported, including data on hybrid and rotated grids, as well as netCDF files with associated constant files. See also: “NetCDF cluster source” (B.1.5) and “NetCDF time series source” (B.1.7).

Properties:

- NetCDF file [Python: `netcdf_file`, type: *filename*]: The name of the input netCDF file.
- Format [Python: `format`, type: *selection*]: The netCDF file format, describing the mapping of the dimension positions to geographic positions. This selection property contains the names of all available netCDF formats which fit the available dimensions and variables of the current input. The set of available formats is specified through the present XML files in the `netcdf_file_formats/` directory of the Insight installation.

B.1.7. NetCDF time series source

This entity provides access to multiple files with equally named and structured output dimensions and variables, including one time dimension, forming a time series of the respective data variables. The local time dimension of each file is replaced by one unified, global time dimension in the entity’s output. The supported netCDF file formats are equivalent to the formats recognized by the “NetCDF data source” (B.1.6). See also: “NetCDF cluster source” (B.1.5).

Properties:

- NetCDF file [Python: `netcdf_file`, type: *filename*]: The names of all netCDF files of the time series.
- Format [Python: `format`, type: *selection*]: The netCDF file format, describing the mapping of the dimension positions to geographic positions. This selection property contains the names of all available netCDF formats which fit the available dimensions

and variables of the current input. The set of available formats is specified through the present XML files in the `netcdf_file_formats/` directory of the Insight installation.

- Sort files alphabetically [Python: `sort_files_alphabetically`, type: *boolean*]: Controls whether or not the input files are sorted alphabetically based on their name.

B.1.8. Trajectories data source

This data source reads in trajectory data from a file. The input file has to adhere to the Lagranto (Wernli and Davies, 1997) file format. Each traced variable is output as a single output variable, with corresponding associated positions. The "Trajectory" dimension selects a single trajectory, whereas the "Sample" dimension allows an iteration over the single samples of each trajectory.

Output dimensions:

Trajectory, Sample

Properties:

- Trajectories file [Python: `trajectories_file`, type: *filename*]: The name of the input trajectories file.
- Frequency [Python: `frequency`, type: *integer*]: The value of this property determines how many trajectories from the input file are included in the output of this entity. If set to "1", every trajectory is included. A "2" means that every second trajectory is considered, and so on.

B.1.9. Vertical profile source

This entity represents a vertical profile, that is, a mapping from height levels to scalar values. Its output is designed to match the input of the external profile of the "Complete segmentation (4D)" (B.2.7).

Output variables:

height, value

Output dimensions:

points

Properties:

- Vertical profile [Python: `vertical_profile`, type: *verticalProfile*]: This property represents the vertical profile. The profile consists of value pairs representing a height in hPa, as well as a corresponding scalar (threshold) value.

B.2. Data processors

B.2.1. Binary operation

This entity combines up to two input variables into one output variable using a binary operation selected by the user. The user can specify fixed values which replace the values of missing input variables. The grid and positional information of the output variable are taken over from either the first or the second input variable, checking the availability in this order.

Variable requests:

a, b

Output variables:

result

Properties:

- Operation [Python: `operation`, type: *selection*]: The binary operation to be applied to the input data (or the default values). Options are: *addition, subtraction, multiplication, division, minimum, maximum.*
- Fixed A [Python: `fixed_a`, type: *double*]: The default input value to be used if no input variable is mapped to input “a”.
- Fixed B [Python: `fixed_b`, type: *double*]: The default input value to be used if no input variable is mapped to input “b”.

B.2.2. Color combiner

This entity combines two input colors into one output color. It allows more complex ways of combination than, for example, a simple component-wise combination using the “Binary operation” (B.2.1) entity. The grid and positional information are taken over from the connected input variables, whereas input “a” is checked first.

Variable requests:

color a, color b

Dimension requests:

color a component, color b component

Output variables:

color

Output dimensions:

Color component

Properties:

- mix mode [Python: `mix_mode`, type: *selection*]: Sets the way in which the colors are combined. Currently, only one mode is available: *replace black*, which always outputs the input color "a" except if "a" is completely black. In this case, the output is color "b".

B.2.3. Color mapper

This entity maps input values to specific colors. The mapping is specified by a set of values associated with colors. Colors in-between are obtained by means of linear interpolation. A corresponding, user-configurable legend can be added to the viewport. The legend can either depict absolute or relative values. Some default color maps can be obtained by using manipulators available through the "Entity"-(context-)menu or directly via the corresponding classes defined in the `colormaps.py` script file.

Variable requests:

dataIn

Output variables:

color

Output dimensions:

component

Properties:

- Legend pos [Python: `legend_pos`, type: *selection*]: The position of the legend overlay. Options are: *none, leftmost, left, right, rightmost.*
- Caption [Python: `caption`, type: *string*]: The caption displayed below the legend overlay. By default, this is set to the name of the mapped input variable.
- Text color [Python: `text_color`, type: *color*]: The color for drawing the caption, the outer lines, and the values of the legend overlay.
- Text size [Python: `text_size`, type: *double*]: The font size of the caption and the values of the legend.
- Show percentage [Python: `show_percentage`, type: *boolean*]: Flag to specify whether relative or absolute values are used for the legend. If relative values shall be used, the settings of the "Percentage range begin" and "Percentage range end" properties determine the relative values displayed.
- Percentage range begin [Python: `percentage_range_begin`, type: *double*]: The lower border for computing the relative values of the legend overlay, corresponding to zero percent.
- Percentage range end [Python: `percentage_range_end`, type: *double*]: The upper border for computing the relative values of the legend overlay, corresponding to 100 percent.

- Regular dist. legend [Python: `regular_dist_legend`, type: *boolean*]: If enabled, the range of the displayed legend is subdivided into ten regular intervals. The tick marks of the legend are displayed at the borders of these intervals. If this option is disabled, the number of tick marks and their labels are defined by the the given value/color pairs.
- Reverse legend order [Python: `reverse_legend_order`, type: *boolean*]: Normally, the lowest value is displayed at the bottom of the legend. If this option is enabled, the order is reversed. This is useful, for example, for pressure values.
- Mapping [Python: `mapping`, type: *colorMap*]: The actual color map as a set of value/color pairs. Values in-between are obtained by means of linear interpolation.

B.2.4. Color mapper (ext. position)

This entity is a variant of the regular “Color mapper” (B.2.3) with the difference that the grid and position of the output are not taken over form the main input variable, but from an additional input variable. The value of the additional input variable is ignored, only its position is taken into account. One example case where this is useful is the coloring of voxels (provided by a segmentation) with a color provided by a different variable (cf. the output of “Complete segmentation (4D)” (B.2.7)).

Variable requests:

dataIn, posIn

Output variables:

color

Output dimensions:

component

Properties:

- Legend pos [Python: `legend_pos`, type: *selection*]: The position of the legend overlay. Options are: *none, leftmost, left, right, rightmost.*
- Caption [Python: `caption`, type: *string*]: The caption displayed below the legend overlay. By default, this is set to the name of the mapped input variable.
- Text color [Python: `text_color`, type: *color*]: The color for drawing the caption, the outer lines, and the values of the legend overlay.
- Text size [Python: `text_size`, type: *double*]: The font size of the caption and the values of the legend.
- Show percentage [Python: `show_percentage`, type: *boolean*]: Flag to specify whether relative or absolute values are used for the legend. If relative values shall be used, the settings of the “Percentage range begin” and “Percentage range end” properties determine the relative values displayed.

- Percentage range begin [Python: `percentage_range_begin`, type: *double*]: The lower border for computing the relative values of the legend overlay, corresponding to zero percent.
- Percentage range end [Python: `percentage_range_end`, type: *double*]: The upper border for computing the relative values of the legend overlay, corresponding to 100 percent.
- Regular dist. legend [Python: `regular_dist_legend`, type: *boolean*]: If enabled, the range of the displayed legend is subdivided into ten regular intervals. The tick marks of the legend are displayed at the borders of these intervals. If this option is disabled, the number of tick marks and their labels are defined by the the given value/color pairs.
- Reverse legend order [Python: `reverse_legend_order`, type: *boolean*]: Normally, the lowest value is displayed at the bottom of the legend. If this option is enabled, the order is reversed. This is useful, for example, for pressure values.
- Mapping [Python: `mapping`, type: *colorMap*]: The actual color map as a set of value/color pairs. Values in-between are obtained by means of linear interpolation.

B.2.5. Color mapper 2D

This entity is similar to the "Color mapper" (B.2.3) with the difference that it takes two variables as input. The first input variable determines the basic color, whereas the other input variable determines an interpolation between the basic color and another fixed color (the "blend color"). This can, for example, be used to control the saturation of the basic color, if the blend color is set to white. The range used for the interpolation between basic and blend color is controlled through additional properties.

Variable requests:

dataIn, secondDataIn

Output variables:

color

Output dimensions:

component

Properties:

- Legend pos [Python: `legend_pos`, type: *selection*]: The position of the legend overlay. Options are: *none, leftmost, left, right, rightmost.*
- Caption [Python: `caption`, type: *string*]: The caption displayed below the legend overlay. By default, this is set to the name of the mapped input variable.
- Text color [Python: `text_color`, type: *color*]: The color for drawing the caption, the outer lines, and the values of the legend overlay.

- Text size [Python: `text_size`, type: *double*]: The font size of the caption and the values of the legend.
- Show percentage [Python: `show_percentage`, type: *boolean*]: Flag to specify whether relative or absolute values are used for the legend. If relative values shall be used, the settings of the "Percentage range begin" and "Percentage range end" properties determine the relative values displayed.
- Percentage range begin [Python: `percentage_range_begin`, type: *double*]: The lower border for computing the relative values of the legend overlay, corresponding to zero percent.
- Percentage range end [Python: `percentage_range_end`, type: *double*]: The upper border for computing the relative values of the legend overlay, corresponding to 100 percent.
- Regular dist. legend [Python: `regular_dist_legend`, type: *boolean*]: If enabled, the range of the displayed legend is subdivided into ten regular intervals. The tick marks of the legend are displayed at the borders of these intervals. If this option is disabled, the number of tick marks and their labels are defined by the the given value/color pairs.
- Reverse legend order [Python: `reverse_legend_order`, type: *boolean*]: Normally, the lowest value is displayed at the bottom of the legend. If this option is enabled, the order is reversed. This is useful, for example, for pressure values.
- Mapping [Python: `mapping`, type: *colorMap*]: The actual color map as a set of value/color pairs. Values in-between are obtained by means of linear interpolation.
- Second value lower bound [Python: `second_value_lower_bound`, type: *double*]: The lower bound for the additional interpolation controlled by the second input value. The final color results from a linear interpolation between the blend color (at the lower bound) and the original color (at the upper bound).
- Second value upper bound [Python: `second_value_upper_bound`, type: *double*]: The upper bound for the additional interpolation controlled by the second input value.The final color results from a linear interpolation between the blend color (at the lower bound) and the original color (at the upper bound).
- Second value blend color [Python: `second_value_blend_color`, type: *color*]: The blend color for the additional interpolation.
- Second value caption [Python: `second_value_caption`, type: *string*]: The caption to be displayed for the second dimension of the mapping.

B.2.6. Color setter

This data entity replaces the value of a given input data variable with a fixed color, keeping the dimensions, the associated grid and the position of the original variable. A dimension for accessing the different color components is added. An example application is the coloring of vertex data from a "2d-model data source" (B.1.1).

Variable requests:

value

Output variables:

color

Output dimensions:

component

Properties:

- Color [Python: `color`, type: *color*]: The fixed color to replace the value of the mapped input variable.

B.2.7. Complete segmentation (4D)

This entity performs a complete, four-dimensional segmentation of the provided input data and a localization of the occurring genesis, lysis, merging, and splitting events on a per-grid-point basis. The input data has to be given on an associated grid. The segmentation criteria are controlled by the user through properties, see the descriptions below for details. The resulting segments are represented as sets of voxels (selected grid points). Each voxel is associated with the position of the respective grid cell and with the value of the input variable at the corresponding position. In the output, the voxels are represented as a 4D variable with the dimensions “time” (if available), “segment”, “feature”, and “voxel”. Additional 4D variables with the same dimensions are provided to output the locations of merging, splitting, genesis, and lysis events. These variables cover all grid cells of the features, the actual occurrence of an event is indicated by the value of the variable. For this, a “1” is associated with an event, whereas a value of “0” indicates that there is no event at the respective position. Additional (statistical) information about the segments, features, and sub-segments is available through the respective output variables. Single sub-segments can be accessed through the “subsegment voxel” variable. Information about the nodes and edges of the event-graph can be obtained using the “global feature id” and the “previous feature id” output variables provided for each 3D feature.

Variable requests:

value, GSC-value (optional), ext. threshold value (optional), ext. profile height (optional), ext. profile value (optional)

Dimension requests:

x, y, z, time, profile point, ext. threshold time

Output variables:

value threshold ratio, merge voxel, split voxel, genesis voxel, lysis voxel, feature pos, feature events, feature size, feature size change, global feature id, previous features id, global feature id per voxel, segment id, segment lifespan, subsegment id, subsegment lifespan, subsegment average size, subsegment average pos, subsegment min lon, subsegment max lon, voxel, subsegment voxel

Output dimensions:

segment, feature, voxel, component, previous feature, subsegment, subsegment feature, subsegment voxel

Properties:

- Criterion [Python: `criterion`, type: *selection*]: The local selection criterion to be tested at every voxel position. Necessary parameters are specified through additional properties. Options are: *Below fixed threshold, Above fixed threshold, Below vertical profile, Above vertical profile, Deviation from profile.*
- Global selection criterion [Python: `global_selection_criterion`, type: *selection*]: The global selection criterion to be tested at the end of the segmentation. Options are: *None, Mean below threshold, Mean above threshold, Min below threshold, Min above threshold, Max below threshold, Max above threshold, Min below mult. of local threshold, Max above mult. of local threshold.*
- GSC threshold [Python: `gsc_threshold`, type: *double*]: The threshold of the global selection criterion (or the factor, if the GSC is a multiple of the local threshold).
- Min. segment lifespan [Python: `min_segment_lifespan`, type: *integer*]: The minimum lifespan (in time steps) a segment has to have. Any segment with a lifespan of less time steps is discarded at the end of the segmentation process.
- Threshold [Python: `threshold`, type: *double*]: The threshold of the local selection criterion. Its value is ignored if a vertical profile is used.
- Use external threshold/profile [Python: `use_external_thresholdprofile`, type: *boolean*]: Enables/disables the usage of an external source for either the single threshold value or for the vertical profile of the local selection criterion. The external source is provided through the corresponding optional input variables.
- Use double thresholding [Python: `use_double_thresholding`, type: *boolean*]: Enables/disables the performing of double thresholding. Double thresholding means, that the original threshold is used to obtain seed 3D features, which are then grown into the full features using a weaker threshold obtained by multiplying the original threshold with the "DT factor" (see below).
- DT factor [Python: `dt_factor`, type: *double*]: This factor is used to calculate the weaker threshold used for double thresholding.
- DT variable growing [Python: `dt_variable_growing`, type: *boolean*]: If enabled, the growing of the 3D features is not performed uniformly, but via a probability-based method that simulates different growing speeds. The speed is reduced in areas where the variable value is closer to the weaker threshold than to the stronger threshold, such that, especially in narrow areas (assuming smooth data), a better separation of features is achieved.
- Vertical profile [Python: `vertical_profile`, type: *verticalProfile*]: The vertical profile for the local selection criterion. This profile is only used, if the corresponding local selection criterion is selected, and if the usage of an external source is disabled.

- Min. deviation [Python: `min_deviation`, type: *double*]: The required minimum deviation from the profile for fulfilling the local selection criterion. This value is only relevant if the respective local selection criterion ("Deviation from profile") is selected.
- X wrapping [Python: `x_wrapping`, type: *boolean*]: Enables/disables the wrapping of the x-dimension (selecting the longitude) in the feature detection phase.
- Y wrapping [Python: `y_wrapping`, type: *boolean*]: Enables/disables the wrapping of the y-dimension (selecting the latitude) in the feature detection phase.
- Vertical core area [Python: `vertical_core_area`, type: *boolean*]: If this option is selected, any segment has to have at least one voxel inside the vertical core area defined by the respective properties (see below). This is an additional constraint extending the global selection criterion.
- Core area bottom (hPa) [Python: `core_area_bottom_hpa`, type: *double*]: The lower bound (highest pressure) of the core area.
- Core area top (hPa) [Python: `core_area_top_hpa`, type: *double*]: The upper bound (lowest pressure) of the core area.
- Horizontal dilation steps [Python: `horizontal_dilation_steps`, type: *integer*]: All detected 3D features are dilated by letting them grow for the given number of steps. Dilation can be used to avoid an over-segmentation of the data. If the dilation is not limited to the feature tracking phase (see below), it can be seen as an extension of the local selection criterion.
- Dilation only for tracking [Python: `dilation_only_for_tracking`, type: *boolean*]: If true, the segments are dilated only internally for feature-tracking reasons. If false, the resulting features are dilated, and newly connected features are merged.
- Sub-segment method [Python: `subsegment_method`, type: *selection*]: This property determines the strategy for the detection of sub-segments. The available options are: *Two-way size comparison, Most similar successor, Two-way size comparison below threshold, Size ratio comparison.*
- Sub-segment size threshold [Python: `subsegment_size_threshold`, type: *double*]: If the "Two-way size comparison below threshold" strategy is selected, this property defines the threshold for the grouping into major and minor features. The two-way size comparison is then applied to determine the chains of minor features connecting the major features. For the "Size ratio comparison" strategy, this property defines the threshold on the minimum allowed size-ratio between the smaller and the larger of two connected features.
- Extended feature tracking [Python: `extended_feature_tracking`, type: *boolean*]: Enables/disables the extended feature tracking based on the attributes size and center of mass of the features. Thresholds on the allowed differences between these attributes are determined by additional properties.
- Distance threshold [Python: `distance_threshold`, type: *double*]: The maximum allowed distance (in km) between features during extended feature tracking.
- Vertical threshold [Python: `vertical_threshold`, type: *double*]: The maximum allowed vertical distance (in km) between features during extended feature tracking.

- Size ratio threshold [Python: `size_ratio_threshold`, type: *double*]: The maximum allowed factor for the size ratio between features for detecting a continuation during extended feature tracking. The ratio is always ≥ 1.
- Event location growing steps [Python: `event_location_growing_steps`, type: *integer*]: The number of growing steps in the second growing phase during event localization. Since the location of an event can only be estimated with a limited accuracy, this second growing phase allows for a controllable amount of fuzziness of the result.
- Ext. threshold time mapping [Python: `ext_threshold_time_mapping`, type: *selection*]: Determines the mapping from the value's time dimension to the external threshold's time dimension. Possible values are: *1:1, interpolate month.*
- Time mapping start year [Python: `time_mapping_start_year`, type: *integer*]: This is the value's starting year at time step zero (only required for the "interpolate month" time mapping).
- Time mapping start month [Python: `time_mapping_start_month`, type: *integer*]: This is the value's starting month at time step zero (only required for the "interpolate month" time mapping).
- Time mapping start day [Python: `time_mapping_start_day`, type: *integer*]: This is the value's starting day at time step zero (only required for the "interpolate month" time mapping).
- Time mapping start hour [Python: `time_mapping_start_hour`, type: *integer*]: This is the value's starting hour at time step zero (only required for the "interpolate month" time mapping).
- Time mapping delta hours [Python: `time_mapping_delta_hours`, type: *integer*]: This is the value's time difference in hours between two consecutive time steps (only required for the "interpolate month" time mapping).

B.2.8. Contour lines

This entity creates a set of line strips and provides a visual overlay for the visualization of contour lines (including captions) of given 2D input data. The input data has to be given on an associated grid. The entity is designed to provide an output that matches the "Line strip representation" (B.3.9) entity's input.

Variable requests:

value

Dimension requests:

x, y

Output variables:

iso lines

Output dimensions:

polygon, vertex

Properties:

- Start [Python: `start`, type: *double*]: The start isovalue at which a contour line should be drawn.
- Delta [Python: `delta`, type: *double*]: The distance between the values of two neighboring contour lines.
- Vertical offset [Python: `vertical_offset`, type: *double*]: This offset is added to the z-coordinate of the positions of the computed contour lines. This helps to avoid clipping errors if more data is to be displayed at the same grid positions, for example when using an additional “Simple grid” (B.3.13) visual representation.
- Draw captions [Python: `draw_captions`, type: *boolean*]: Switch to turn all captions on or off.
- Text color [Python: `text_color`, type: *color*]: The color of the captions.
- Min. length for caption [Python: `min_length_for_caption`, type: *integer*]: A threshold on the minimum contour line length for a caption to be added.

B.2.9. Date mapper

This entity maps the index of the time dimension of a given input variable to a date. The date is output through a set of output variables, as well as through read-only properties that can be accessed by scripts.

Variable requests:

value

Output variables:

value, year, month, day, hour

Properties:

- Start year [Python: `start_year`, type: *integer*]: The start year at time index “0”.
- Start month [Python: `start_month`, type: *integer*]: The start month at time index “0”.
- Start day [Python: `start_day`, type: *integer*]: The start day at time index “0”.
- Start hour [Python: `start_hour`, type: *integer*]: The start hour at time index “0”.
- Delta hour [Python: `delta_hour`, type: *integer*]: The time difference in hours between two consecutive time steps.
- Date (out) [Python: `date_out`, type: *string*]: This read-only property contains a string representation of the date corresponding to the current index of the time dimension.

B.2.10. Euclidean vector norm

The up to three input variables of this entity are interpreted as a vector whose components all have the same unit. The output is the Euclidean norm of the vector (its length).

Variable requests:

x, y, z

Output variables:

length

Properties:

- Square length [Python: `square_length`, type: *boolean*]: If enabled, the output is the square of the length of the vector.

B.2.11. Grid mapper

This entity collects data associated with a grid and outputs the collected data as a new variable directly accessible through the grid's dimensions. Some data, for example the vertices that are output by the "Complete segmentation (4D)" (B.2.7) entity, are associated with a grid, but have dimensions different than the grid's own dimensions. For further processing of such data with other entities, for example using a "Simple grid" (B.3.13), it is necessary that the data is not only associated with a grid, but that the variable's dimensions also directly reflect the adjacencies on the grid. This is what the grid mapper achieves. Several user options control the way in which the input data is mapped onto the grid. If the summation mode is "count", the input values are simply counted at each grid position (summed up with a fixed value of one). If the summation mode is "sum", the respective values are added up at each position. In the "average" mode, the average value at each position is computed. In addition, the user may change the way in which the z-dimension is handled. If the user enables the "vertical integration" option, the z-coordinate is completely ignored, resulting in a two-dimensional output grid. Additionally, the user controls the "unify timesteps" property, which enables or disables an averaging over all variable values present at exactly the same position in space (3D or 2D) and time. Currently, values which are essentially zero (with respect to a small delta to avoid rounding errors), are taken out of this averaging. As an additional output, the lowest covered grid level per horizontal position is provided as a 2D data set.

Variable requests:

value

Dimension requests:

Points dimension, time

Output variables:

data, lowest model level

Properties:

- Summation mode [Python: `summation_mode`, type: *selection*]: Selects the way in which values at the same position are processed. Options are: *count, sum, average.*
- Vertical integration [Python: `vertical_integration`, type: *boolean*]: If enabled, the z-dimension of the input data is ignored, resulting in output data on a two-dimensional grid.
- Unify timesteps [Python: `unify_timesteps`, type: *boolean*]: If enabled, all values of the same time step at the same position are averaged.

B.2.12. Iso surface

This entity takes a three-dimensional variable as input and computes, based on a given isovalue, a set of voxels representing the corresponding isosurface. This set contains all voxels in the lower-left-bottom corner of a $2\times2\times2$ neighborhood of grid cells, in which values above and below the isovalue exist. The output is designed to work seamlessly as input of the “Iso surface representation” (B.3.7) entity.

Variable requests:

value

Dimension requests:

x, y, z

Output variables:

voxel, value

Output dimensions:

voxel

Properties:

- Iso value [Python: `iso_value`, type: *double*]: The isosurface’s isovalue.

B.2.13. Mean vertical profile source

This entity outputs height/value pairs corresponding to a vertical profile computed for a given input data set. The profile can be modified by specifying the number of vertical bins for grouping the values before their mean values are computed, a fixed factor to be multiplied to the profile, and by specifying a height cutoff (all values of this height and above will be reduced to zero).

Variable requests:

value

Dimension requests:

lon, lat, lev, time

Output variables:

height, value

Output dimensions:

points

Properties:

- Vertical resolution [Python: `vertical_resolution`, type: *integer*]: Determines the number of height level intervals that are considered for the creation of the vertical profile.
- Reduction factor [Python: `reduction_factor`, type: *double*]: A fixed factor that is applied to all input values.
- Height cutoff [Python: `height_cutoff`, type: *integer*]: All input values at this height and above are set to zero.

B.2.14. Position adder

This entity takes a main input variable and associates it with new positional information provided by three additional input variables. The output variable takes over all dimensions of the main input variable, and all additional dimensions from the positional input variables.

Variable requests:

value, lon, lat, p

Output variables:

value

B.2.15. Scripted operation

This entity provides an arbitrary n-ary operation. The operation is specified through a (usually short) Python script. The script is divided into two parts: One part is executed only once, at the beginning of the access of the entity's single output variable, and the second part is executed every time the value of the output variable is queried. This second part is responsible for computing this output value based on the respective values of an arbitrary number of input variables.

Variable requests:

var1

Output variables:

result

Properties:

- Script [Python: `script`, type: *text*]: The main part of the script which is executed every time the value of the output variable is queried. The corresponding values of

the input variables can be accessed inside the script as “var1”, “var2”, and so on. The result has to be stored in a global variable with the name specified by the “Output name” property.

- Init [Python: `init`, type: *text*]: The part of the script that is executed once at the beginning of the access of the output variable.
- Output name [Python: `output_name`, type: *string*]: The name of the output variable of the entity, and of the global variable in which the script stores the result. The default variable name is “result”.

B.2.16. Unary operation

This entity performs a unary operation on the values of the input variable and outputs the result through a single output variable. This output variable takes over the grid and positional information from the input variable.

Variable requests:

value

Output variables:

result

Properties:

- Operation [Python: `operation`, type: *selection*]: The unary operation to be performed on the input values. Options are: *abs, fit lon, int, sqrt, square.* The “fit lon” option ensures that the value stays in the interval [-180, 180].

B.2.17. Unit converter

This entity converts the values of an input variable from one unit to another. Missing data values/fill values can be excluded from this conversion.

Variable requests:

value

Output variables:

convertedValue

Properties:

- Conversion [Python: `conversion`, type: *selection*]: This property controls the unit conversion to be performed. Options are: $m \rightarrow hPa$, $km \rightarrow hPa$.
- Keep missing data [Python: `keep_missing_data`, type: *boolean*]: Option to explicitly check the input data for missing data values, which are then excluded from the conversion.

B.2.18. Variable combiner

This data entity combines multiple input variables into one output variable with an additional "component" dimension for selecting the input source. In addition to this "component" dimension, the output variable takes over all dimensions from the input variables. The data grid and positional information are taken over from the first input variable which offers these.

Variable requests:

> var1

Output variables:

> value

Output dimensions:

> component

B.3. Visual representations

B.3.1. Box representation

This entity takes a one-dimensional series of color data as input. The color values have to be associated with a grid and positions. Each value is visualized as a colored box at the associated position. The corner vertices of each box match the corners of the associated grid cells. A common practice is to access a multidimensional input variable by combining its dimensions into the one requested "Points dimension".

Variable requests:

> color

Dimension requests:

> Points dimension, Color component

Properties:

- Shading [Python: `shading`, type: *boolean*]: Determines whether the boxes are shaded (affected by lighting) when drawn, or not.
- Single z-slice [Python: `single_zslice`, type: *boolean*]: If this option is enabled, out of the set of all boxes only a single slice at a fixed z level is drawn. The level is specified through the "Z-slice level" property.
- Z-slice level [Python: `zslice_level`, type: *integer*]: Specifies the single z level at which the slice of boxes should be drawn. This value is ignored, if the "Single z-slice" property is not set.

B.3.2. Caption

This entity adds a caption overlay to the visualization. The position, size, and color of the caption can freely be chosen by the user.

Properties:

- Caption [Python: `caption`, type: *string*]: The text to be displayed.
- X [Python: `x`, type: *double*]: The x position of the lower left corner of the caption.
- Y [Python: `y`, type: *double*]: The y position of the lower left corner of the caption.
- Point size [Python: `point_size`, type: *integer*]: The font size of the caption.
- Color [Python: `color`, type: *color*]: The color of the caption.

B.3.3. Event graph representation

This entity provides a three-dimensional visualization of the event graph resulting from a complete four-dimensional segmentation. Its input requests are designed to match the output of the "Complete segmentation (4D)" (B.2.7) entity. The visualization consists of single spheres for each node of the event graph. The sizes of the spheres are proportional to the sizes of the associated 3D features. The edges of the event graph are visualized in form of colored arcs connecting the spheres. The user decides which time steps are included in the visualization.

Variable requests:

feature pos, feature size, feature events, global feature id, previous features id

Dimension requests:

time, feature, feature pos component, previous feature

Properties:

- Graph type [Python: `graph_type`, type: *selection*]: Selects the type of the event graph visualization. Currently, only one type is available. Options are: *3D.*
- Display mode [Python: `display_mode`, type: *selection*]: This property controls the visualization of events from different time steps. Options are: *all timesteps, single timestep, highlight timestep.*
- Timestep [Python: `timestep`, type: *integer*]: The time step to be displayed in "single timestep"-mode, or the time step to be highlighted in "highlight timestep"-mode.

B.3.4. Graticule

This entity draws a horizontal graticule at a fixed elevation.

Properties:

- Minimum x [Python: `minimum_x`, type: *double*]: The minimum longitudinal bound for the graticule.
- Maximum x [Python: `maximum_x`, type: *double*]: The maximum longitudinal bound for the graticule.
- Minimum y [Python: `minimum_y`, type: *double*]: The minimum latitudinal bound for the graticule.
- Maximum y [Python: `maximum_y`, type: *double*]: The maximum latitudinal bound for the graticule.
- Z position [Python: `z_position`, type: *double*]: The fixed height (in hPa) of the graticule.
- Color [Python: `color`, type: *color*]: The color of the graticule.
- Text size [Python: `text_size`, type: *double*]: The text size of the legend.

B.3.5. Grid representation

This entity visualizes a horizontal slice of the underlying grid of a data variable. The "z" and "time" dimensions of the grid are set to zero for the visualization.

Variable requests:

value

Properties:

- Color [Python: `color`, type: *color*]: The color of the grid visualization.
- Width [Python: `width`, type: *double*]: The line width for the grid visualization.
- X frequency [Python: `x_frequency`, type: *integer*]: The step size for traversing the x-dimension during the visualization of the longitudes.
- Y frequency [Python: `y_frequency`, type: *integer*]: The step size for traversing the y-dimension during the visualization of the latitudes.

B.3.6. Ground

This entity draws a flat horizontal surface bounded by given constraints at a fixed elevation. This is useful, for example, for covering averted features of a 3D plot when using spherical projection.

Properties:

- Minimum x [Python: `minimum_x`, type: *double*]: The minimum longitudinal bound for the ground surface.
- Maximum x [Python: `maximum_x`, type: *double*]: The maximum longitudinal bound for the ground surface.

- Minimum y [Python: `minimum_y`, type: *double*]: The minimum latitudinal bound for the ground surface.
- Maximum y [Python: `maximum_y`, type: *double*]: The maximum latitudinal bound for the ground surface.
- Z position [Python: `z_position`, type: *double*]: The fixed height (in hPa) of the ground.
- Color [Python: `color`, type: *color*]: The color of the ground.

B.3.7. Iso surface representation

This entity is responsible for the visualization of an isosurface. As input, this entity requires a series of voxels representing the positions of the isosurface (see “Iso surface” (B.2.12)), a variable for accessing the complete underlying 3D data set, and a variable defining the color of the isosurface at every position of the 3D data set. The isovalue is either specified through the corresponding property, or as the value of the “voxel” input variable (this is the case for the output of the “Iso surface” (B.2.12) entity).

Variable requests:

voxel, value, value color

Dimension requests:

voxel, x, y, z, Color component

Properties:

- Enable lighting [Python: `enable_lighting`, type: *boolean*]: Enables or disables smooth lighting of the isosurface.
- Invert normals [Python: `invert_normals`, type: *boolean*]: When lighting is enabled, flipping the direction of the normals may improve the quality of the visualization.
- Iso value (optional) [Python: `iso_value_optional`, type: *double*]: The isovalue of the isosurface to be visualized. If this value is set to -999, the isovalue has to be given via the “voxel” input variable.

B.3.8. Labeled axis

This entity visualizes a single axis of a data grid in form of a line with tick marks. The axis is labeled with the positional information obtained from the grid. The grid itself is taken from the single input variable, the spatial dimension for which the axis is drawn is selected by the user.

Variable requests:

value

Properties:

- Number of ticks [Python: `number_of_ticks`, type: *integer*]: The number of tick marks to be displayed.
- Tick multiples [Python: `tick_multiples`, type: *integer*]: For avoiding inconsistent decimal fractions, all ticks are placed at multiples of the given number. If set to zero, the original marks are kept.
- Dim type [Python: `dim_type`, type: *selection*]: The value's grid dimension for which the axis should be drawn.
- Offset [Python: `offset`, type: *double*]: The offset (in OpenGL units) from the original position to the drawn axis.
- Color [Python: `color`, type: *color*]: The color used to draw the axis, tick marks, and labels.
- Text size [Python: `text_size`, type: *double*]: The text size of the labels.

B.3.9. Line strip representation

For the visualization of sets of piecewise linear curves, this entity can be used. The line strips are either visualized using simple lines with a perspective-independent thickness, or they are drawn as fully shaded 3D cylinder curves. Common input for the line strip representation is the (colored) output of the "2d-model data source" (B.1.1) or the "Contour lines" (B.2.8) entity.

Variable requests:

color

Dimension requests:

polygon, vertex, Color component

Properties:

- Line thickness [Python: `line_thickness`, type: *integer*]: The thickness of the simple lines to be drawn.
- Draw cylinder curves [Python: `draw_cylinder_curves`, type: *boolean*]: If enabled, the simple lines are replaced by 3D cylinder curves.
- Cylinder thickness [Python: `cylinder_thickness`, type: *double*]: The thickness of the 3D cylinders to be drawn.
- Filled line strips [Python: `filled_line_strips`, type: *boolean*]: If enabled and no cylinder curves are drawn, all line sequences are replaced by filled polygons sharing the same edges as the line strip.
- Interrupt lines at date line [Python: `interrupt_lines_at_date_line`, type: *boolean*]: Option to split line segments at the date line transition, in order to avoid display errors in combination with certain projection types.
- Use size constraints [Python: `use_size_constraints`, type: *boolean*]: If enabled, line segments starting or ending outside of the given size constraints are not drawn.

- Minimum x [Python: `minimum_x`, type: *double*]: The lower bound on the longitudinal coordinate.
- Maximum x [Python: `maximum_x`, type: *double*]: The upper bound on the longitudinal coordinate.
- Minimum y [Python: `minimum_y`, type: *double*]: The lower bound on the latitudinal coordinate.
- Maximum y [Python: `maximum_y`, type: *double*]: The upper bound on the latitudinal coordinate.
- Fixed vertical position [Python: `fixed_vertical_position`, type: *boolean*]: Flag for using a fixed vertical position, instead of the real vertical position associated with the input data.
- Vertical position [Python: `vertical_position`, type: *double*]: The fixed vertical position to replace the real vertical position of the input data, if "Fixed vertical position" is enabled.

B.3.10. Point of interest

This entity visualizes a mark at a single geographic point of interest, annotated with a user-defined text.

Properties:

- Label [Python: `label`, type: *string*]: The text of the annotation to be displayed next to the mark.
- Lon [Python: `lon`, type: *double*]: The longitudinal coordinate of the point of interest.
- Lat [Python: `lat`, type: *double*]: The latitudinal coordinate of the point of interest.
- P [Python: `p`, type: *double*]: The elevation (given in hPa) of the point of interest.
- Point color [Python: `point_color`, type: *color*]: The color of the mark to be displayed.
- Text color [Python: `text_color`, type: *color*]: The text color of the annotation.

B.3.11. Point representation

A set of OpenGL points is used for the visualization of a series of input voxel data. The voxel data has to be given as color values with an associated position. The size of all points is a fixed, user-defined value. Similar to the "Box representation" (B.3.1) entity, it is common to combine multiple input dimensions as input for the single "Points dimension" of this entity.

Variable requests:

color

Dimension requests:

Points dimension, Color component

Properties:

- Point size [Python: `point_size`, type: *double*]: The uniform size for all points.
- Minimum Z [Python: `minimum_z`, type: *double*]: All higher points with lower pressure values are discarded.
- Maximum Z [Python: `maximum_z`, type: *double*]: All lower points with higher pressure values are discarded.

B.3.12. Point volume representation

This visual representation works similar to the “Point representation” (B.3.11) entity. In addition to the basic visualization of voxels in form of colored points, the size and transparency of the points are varied with respect to the data values at each position.

Variable requests:

value, color

Dimension requests:

Points dimension, Color component

Properties:

- Point size factor [Python: `point_size_factor`, type: *double*]: This factor is used for the calculation of the point size. The basic point size varies in the range from one up to 25, depending on the global minimum and maximum value, and on the individual values at each position. The specified factor is multiplied afterwards to determine the final point size.
- Skip probability [Python: `skip_probability`, type: *integer*]: To speed up the visualization, some points are randomly ignored. This property controls the probability (0–100) for skipping a point.

B.3.13. Simple grid

This entity visualizes data given on a simple, two-dimensional grid in form of a colored surface. For this, the entity requires a color value as input, associated with positions and two spatial dimensions. For a correct visualization, it is important that the data dimensions reflect the spatial alignment of the data. Although the dimension requests are named “x” and “y”, arbitrary spatial dimension can be mapped as input. For example in case of three-dimensional data, three different axis-aligned orientation can be achieved this way. Multiple user options allow the customization of the visualization.

Variable requests:

color, ext. height (optional)

Dimension requests:

x, y, Color component, ext. height x (optional), ext. height y (optional)

Properties:

- Draw mode [Python: `draw_mode`, type: *selection*]: This property defines whether the surface is drawn solid, or as a wire frame. Options are: *Solid, Wireframe.*
- Color mode [Python: `color_mode`, type: *selection*]: The user can choose between a smooth interpolation of the input color values, and the usage of the nearest color to represent each grid cell. In the latter case, the required number of polygons is increased by a factor of four. Options are: *Nearest, Smooth.*
- Wrap grid [Python: `wrap_grid`, type: *boolean*]: If enabled, the entity connects the polygons used for the visualization of the rightmost grid cells with the leftmost grid cells (along the "x"-dimension). This can be useful for closing visible gaps when the spherical projection is used.
- Fixed vertical position [Python: `fixed_vertical_position`, type: *boolean*]: Instead of drawing each grid cell at the associated three-dimensional position, the elevation at each point is set to the fixed position specified via the "Vertical position" property (see below).
- Vertical position [Python: `vertical_position`, type: *double*]: The fixed vertical position to be used when the "Fixed vertical position" option is enabled.
- Use color key [Python: `use_color_key`, type: *boolean*]: If enabled, all grid cells with the specified "Color key" color are not drawn. This can be useful for the overlapping of multiple different visualizations.
- Color key [Python: `color_key`, type: *color*]: The color to be used for color keying (see the "Use color key" property).
- Shading [Python: `shading`, type: *boolean*]: Enables/disables the shading of the surface based on the surface normals at each position.
- Show grid [Python: `show_grid`, type: *boolean*]: If enabled, an experimental visualization of grid lines is shown. The "Graticule" (B.3.4) or "Labeled axis" (B.3.8) entities are often better choices for visualizing the geographic position and alignment of the data.

B.3.14. [wrapper] Iso surface

This entity wraps up the functionalities of the "Iso surface" (B.2.12) and the "Iso surface representation" (B.3.7) entities for simplifying the visualization of an isosurface. As input, the entity requires a data variable with an associated grid for which the isosurface should be computed, and an optional color value associated with the same grid for coloring the isosurface.

Variable requests:

value, color (optional)

Properties:

- Iso value [Python: `iso_value`, type: *double*]: The isosurface's isovalue.
- Use fixed color [Python: `use_fixed_color`, type: *boolean*]: If enabled, the isosurface is colored using a fixed color.
- Color [Python: `color`, type: *color*]: The fixed color of the isosurface.
- Invert normals [Python: `invert_normals`, type: *boolean*]: When lighting is enabled, flipping the direction of the normals may improve the quality of the visualization.
- Enable lighting [Python: `enable_lighting`, type: *boolean*]: Enables or disables smooth lighting of the isosurface.

B.4. Other entities

B.4.1. Statistics

This entity takes a three-dimensional data variable as input and computes several statistical values of the input data. The results are represented in form of read-only properties. Missing data values are excluded from the computations.

Variable requests:

value

Dimension requests:

x, y, z

Properties:

- Min [Python: `min`, type: *double*]: The minimum input data value.
- Max [Python: `max`, type: *double*]: The maximum input data value.
- Mean [Python: `mean`, type: *double*]: The mean input data value.

C. Reference of the Insight script interface

This appendix provides a reference documentation of the public interfaces of all elements of the `insight_core` module (Sect. C.1) and of the `insight` package (Sect. C.2).

C.1. The `insight_core` module

C.1.1. `Observer` class

The task of this class is to notify Python objects about events from data entities and data compositions that are invoked using Insight's internal notification mechanism. Python objects of classes derived from `Observer` are notified about all occurring events through a private callback function. The `Observer` class is internally used in the `insight` package.

This class has no public interface.

C.1.2. `Output` class

An object of this class is internally used in the implementation of Insight's `ScriptingInterface` class for replacing the default `sys.stdout` and `sys.stderr` Python objects. This way, the corresponding output is passed on to the `write` member function, which enables the display of the output in the scripting console that is a part of Insight's GUI.

write(*self*, text)

Text passed to this function is stored in a buffer for being displayed in the scripting console of Insight.

C.1.3. `UserQuery` class

The purpose of this class is to show the user a dialog window with a set of properties. The values of these properties can be set before the dialog is shown and the user-selected values can be queried after the dialog has been closed.

add_property(*self*, name, type)

Adds a new property to the object. The parameters specify a custom name and the type string of the property. The function returns a `UserQueryProperty` Python object from the `insight` package.

get_property(*self*, name)

Returns the value of the property with the specified name.

get_property_ext(*self*, name)

Returns a dictionary containing the value of the specified property and (depending on the type of the property) any additional settings.

query_user_input(*self*)

This function triggers the presentation of the dialog with all specified properties to the user. After the dialog has been closed, a boolean value is returned indicating whether the dialog was canceled (`False`), or not (`True`).

set_property(*self*, name, value)

Sets the value of the property with the specified name.

set_property_ext(*self*, name, values)

Sets the value and the internal state of the specified property to the value and to the extended settings given in form of a Python dictionary.

properties [*read-only property*]

A list of the names of all properties.

C.1.4. Global functions

_export_netcdf(filename, comp_name, variables[, prop_dim_comp_name, prop_dim_entity_id, prop_dim_property_name, slice_size])

This function invokes the export of a netCDF file containing the stated variables from the specified composition. The variables have to be given in form of a list of tuples. Each tuple has to contain the ID of the entity providing the variable and the variable's name. If an additional dimension derived from an integer property should be exported, it has to be identified through the respective composition name, the ID of the property's provider, and through the property's name. In addition, the export of multiple files can be triggered by dividing the range of the additional dimension into blocks of the stated `slice_size`. If the value is not positive, a single file is output. Note: The reason for the leading underscore of the function name was to avoid a conflict with the more convenient `export_netcdf` function of the `insight` package.

add_composition_property(comp_name, entity_id, property_name)

Adds the specified property of an entity to the composition of the entity.

add_file(filename)

Loads the saved program state from the XML file of the given name, creating new data compositions in the process, while all existing data compositions remain unchanged.

begin_var_access(comp_name, provider_id, output_var_name)

Prepares the access of the specified output variable.

connect(comp_name, provider_id, output_var_name, receiver_id, var_request_name)

Connects the specified output variable to the corresponding variable request.

dim_next(comp_name, provider_id, dim_name)

Increases the current position of the specified dimension by one.

dim_reset(comp_name, provider_id, dim_name)

Resets the current position of the specified dimension.

disconnect(comp_name, receiver_id, var_request_name)

Removes any connected input from the specified variable request.

end_var_access(comp_name, provider_id, output_var_name)

Ends the access of the specified output variable.

entity_id_valid(comp_name, id)

Returns `True`, if the given ID belongs to a valid data entity of the given data composition, `False` otherwise.

fix_var_access_dim(comp_name, provider_id, output_var_name, dim_name, dim_pos)

Sets the corresponding dimension to the given position and marks it as "fixed" for subsequent access of the variable. As a prerequisite, the `begin_var_access` function has to be called in advance.

get_camera([viewport_id])

Returns the current orientation of the camera of the specified or active viewport as a tuple with four components. The way in which the orientation is encoded depends on the selected view controller.

get_connected_var(comp_name, receiver_id, var_request_name)

Returns the data variable connected to the given variable request. The variable is returned in form of a tuple containing the ID of the provider and the variable name. If no variable is connected, `None` is returned.

get_current_composition()

Returns the name of the currently selected data composition.

get_current_dim_size(comp_name, provider_id, dim_name)

Returns the current size of the specified data dimension.

get_dim_mapping(comp_name, receiver_id)

Returns all input dimension mappings of the specified data receiver. The mappings are returned in form of a Python dictionary, mapping the dimension request names to lists of dimensions. Each dimension is represented as a tuple containing the ID of the provider and the name of the dimension.

get_dim_pos(comp_name, provider_id, dim_name, pos)

Sets the current position of the specified dimension to the given position.

get_dim_range(comp_name, provider_id, dim_name)

This function returns the current, possibly restricted, range of the specified dimension's positions.

get_entity_description(comp_name, id)

Returns a string containing the help text of the specified data entity.

get_entity_name(comp_name, id)

Returns a string containing the name of the specified data entity.

get_max_dim_size(comp_name, provider_id, dim_name)

Computes and returns the maximum size of the given data dimension. The computation may take some time since the maximum size is determined by an iteration over all valid combinations of positions of the parent dimensions.

get_property(comp_name, entity_id, property_name)

Returns the value of the specified property.

get_property_description(comp_name, entity_id. property_name)

Returns a string containing the help text of the specified property.

get_property_ext(comp_name, entity_id, property_name)

Returns a dictionary containing the value of the specified property and (depending on the type of the property) any additional settings.

get_property_type(comp_name, entity_id, property_name)

Returns a string identifying the type of the specified property (e.g. "BooleanProperty").

get_real_dim_size(comp_name, provider_id, dim_name)

Returns the real (unconstrained) size of the specified data dimension.

get_unmapped_input_dims(comp_name, receiver_id)

Returns a list containing all incoming dimensions that are not mapped to any input dimension request of the given data receiver. The dimensions are represented as tuples containing the IDs of the providers and the names of the dimensions.

get_var_grid_index(comp_name, provider_id, output_var_name)

Returns a tuple representing the data grid index with respect to the current positions of the data dimensions of the specified data variable. The access of the variable has to be prepared in advance by calling the `begin_var_access` function.

get_var_pos(comp_name, provider_id, output_var_name)

Returns a tuple containing the current geographic position of the specified data variable. The access of the variable has to be prepared in advance by calling the `begin_var_access` function.

get_var_value(comp_name, provider_id, output_var_name)

Returns the current value of the specified data variable. The access of the variable has to be prepared in advance by calling the `begin_var_access` function.

get_viewports_count()

Returns the current number of viewports.

has_property(comp_name, entity_id, property_name)

Returns `True`, if the specified data entity has a property of the given name, `False` otherwise.

has_visual_representation(comp_name, id)

Returns `True`, if the specified data entity provides a visual representation, `False` otherwise.

is_dim_valid(comp_name, provider_id, dim_name)

Returns `True`, if the current position of the specified data dimension is valid, `False` otherwise.

is_var_accessed(comp_name, provider_id, output_var_name)

Returns `True`, if the specified data variable is ready to be accessed, `False` otherwise. The functions `begin_var_access` and `end_var_access` control the accessibility of the data variables.

list_composition_properties(comp_name)

Returns a list containing all composition properties of the specified data composition. Each property is identified by a tuple containing the ID of the provider of the property and the property's name.

list_compositions()

Returns a list containing the unique name of all existing data compositions.

list_dim_requests(comp_name, entity_id)

Returns a list of the names of all input dimension requests of the specified data entity.

list_entities(comp_name)

Returns a dictionary representing all data entities of the specified data composition. The dictionary maps the names of the entities to their unique IDs.

list_entity_types()

Returns a list containing strings with the names of all data entity types (e.g. "Color mapper").

list_output_dims(comp_name, entity_id)

Returns a list containing all available output dimensions of the specified entity. Each dimension is represented as a tuple containing the ID of the provider and the dimension's name.

list_output_var_dims(comp_name, provider_id, output_var_name)

Returns a list containing all dimensions of the specified output variable. Each dimension is represented as a tuple containing the ID of the provider and the dimension's name.

list_output_vars(comp_name, entity_id)

Returns a list of the names of all available output variables of the specified entity.

list_projections()

Returns a list of the names of all available projection types.

list_properties(comp_name, entity_id)

Returns a list of the names of all properties of the specified entity.

list_transformations()

Returns a list of the names of all available map transformation types.

list_var_request_dim_requests(comp_name, receiver_id, request_name)

Returns a list of the names of all input dimension requests associated with the specified variable request.

list_var_requests(comp_name, entity_id)

Returns a list of the names of the input variable requests of the specified data entity.

list_view_controllers()

Returns a list of the names of all available view controllers.

load_file(filename)

Deletes all existing data compositions and loads the saved program state from the XML file of the given name.

move_camera(direction, amount[, viewport_id])

Simulates a mouse move in the specified direction (0: right, 1: up, 2: left, 3: down) in the specified or active viewport. The `amount` parameter controls the amount of movement in the specified direction.

new_composition()

Creates a new data composition and returns its name. The parameter is the proposed name of the composition.

new_entity(comp_name, entity_type_name[,entity_name])

Creates a new entity of the given type and the given or default name and adds it to the composition with the specified name. Returns the ID of the new entity.

new_file(name_of_first_composition)

Invoking this command is equivalent to the "New" operation from the "File" menu. All existing data compositions are deleted, the viewports are reset, and a new composition with the default or the specified name is created.

new_viewport()

Opens an additional viewport window.

register_manipulator(manipulator, command_text, entity_type)

Registers a manipulator, which is a short script available through the context menu of data entities of the specified type. The `command_text` parameter specifies the name of the command, as it shall appear in the user interface. The `manipulator` parameter is the name (as a string) of the Python function implementing the manipulator's functionality. The prefix `insight.manipulators.` is automatically added to the function name. Because of this, all manipulator functions have to be located in the corresponding package.

remove_composition_property(comp_name, entity_id, property_name)

Removes the specified composition property from the corresponding composition.

remove_entity(comp_name, entity_id)

Removes the specified data entity from the corresponding composition.

remove_viewport([viewport_id])

Closes either the active viewport window or the viewport window with the given ID.

save_file(filename)

Saves the current program state as an XML file with the given filename.

set_auto_recompute([comp_name,] auto_recompute)

This function controls the automatic recomputation of the current or specified data composition. The recomputation can be switched on (`True`) and off (`False`).

set_background_color(color[, viewport_id])

Sets the background color of the given or active viewport. The color has to be given as a tuple with three floating point values in the range from zero to one.

set_box_pos(comp_name, entity_id, x, y)

Sets the position of the box representing the specified data entity in the single instance of the `DataCompositionGridView` class to the given x and y coordinates.

set_camera(camera_vec[, viewport_id])

Sets the current orientation of the camera of the specified or active viewport. The orientation has to be given as a tuple with four components. The way in which the orientation is encoded depends on the selected view controller.

set_current_composition(comp_name)

Sets the specified data composition as the currently selected composition.

set_dim_mapping(comp_name, receiver_id, dim_request_name, dim_mapping[, combine_dims])

Connects a set of output dimensions to the specified input dimension request of the given data receiver. The mapping has to be passed to the function as either a single tuple containing the ID of the dimension provider and the output dimension's name, as a list of multiple tuples defining all output dimensions to be mapped to the specified request, or as `None`, indicating that all mapped dimensions of the request shall be removed. The `combine_dims` flag indicates whether multiple dimensions shall be accessed simultaneously (`False`), or by sequentially accessing all valid combinations of positions of the set of dimensions (`True`).

set_dim_pos(comp_name, provider_id, dim_name, pos)

Sets the current position of the specified dimension to the given value.

set_dim_range(comp_name, provider_id, dim_name, range)

Sets the restricted range of the specified dimension to the given interval. The interval has to be given in form of a tuple with two integer values, indicating the lower and upper boundary of the range.

set_draw_axes(draw_axes[, viewport_id])

Switches the drawing of the coordinate axes of the active or the specified viewport either on (`True`) or off (`False`).

set_draw_cube(draw_cube[, viewport_id])

Switches the drawing of the coordinate axes of the active or the specified viewport either on (`True`) or off (`False`).

set_pole_correction(pollon, pollat[, viewport_id])

Tells the active or the specified viewport to perform a global rotation of the visualized data in a way that data on a grid with a rotated pole of the given coordinates appears unrotated.

set_projection(projection_name[, viewport_id])

Sets the projection of the given or active viewport to the projection type of the given name. The available projection types can be queried by the `list_projections` function.

set_property(comp_name, entity_id, property_name, value)

Sets the value of the specified property.

set_property_ext(comp_name, entity_id, property_name, ext_settings)

Sets the value and any additional settings specified in the given dictionary (`ext_settings`) of the specified property. The structure of the dictionary depends on the type of the property. The current extended settings of a property can be queried through the `get_property_ext` function.

set_scaling(scaling[, viewport_id])

Sets the global scaling factors of the three spatial dimensions of the active or the specified viewport. The three factors have to be given as a tuple with three components.

set_transformation(transformation_name[, viewport_id])

Sets the transformation of the given or active viewport to the transformation type of the given name. The available transformation types can be queried by the `list_transformations` function.

set_translation(translation[, viewport_id])

Sets the translation vector of the camera of the active or specified viewport to the given values. The vector has to be given as a tuple with three components.

set_vertical_ratio(ration[, viewport_id])

Sets the scaling factor used to compute the height coordinate of the visualized data for the active or the specified viewport. In contrast to the scaling factors of the `set_scaling` function, this vertical scaling is applied to the geographic coordinates, and not to the final OpenGL world coordinates of the data.

set_view_controller(controller_name[, viewport_id])

Sets the view controller of the given or active viewport to the view controller of the given name. The available view controllers can be queried by the `list_view_controllers` function.

set_vis_rep_visibility(vis_rep_name, visibility[, viewport_id])

Controls the visibility (`True` or `False`) of a visual representation in the single active or any specified viewport. The visual representation is identified through its name.

set_zoom_factor(factor[, viewport_id])

Sets the zoom factor of the active or specified viewport to the given value.

take_screenshot(filename[, res_x, res_y][, viewport_id])

Takes a screenshot and saves it using the specified filename. All image formats supported by Qt can be exported. If no viewport is specified, the active viewport is used. The same holds for the image resolution.

C.2. The `insight` package

A general remark: The `insight` package contains, among other things, the classes for representing and manipulating data compositions, data entities, data variables, data dimensions, variable requests, dimension requests, and properties. In contrast to the `insight_core` module where we use unique names and IDs to identify these objects, most functions of this package expect instances of these classes as parameters.

C.2.1. `colormaps.Colormap` class

This class represents a color mapping (set of value/color pairs) together with several member functions for the manipulation of the mapping. It is derived from the built-in type `dict` and can therefore directly be assigned to a `EntityColorMapProperty` instance of a data

entity. Note that the corresponding setter and getter C++ functions use the `dict` type for representing color maps, such that querying the property returns a standard Python `dict` instance, not a `colormaps.Colormap` instance. As a global convention, the key of a single color mapping is a scalar value, whereas the color is represented as a tuple with three components, containing RGB values between zero and one.

Several default instances of the `Colormap` class are available in the `colormaps` module. Examples are: `colorful`, `rainbow`, and `grayscale`.

added(*self*, key, color)

Returns a copy of the colormap with one additional mapping specified by the given key and color.

bordered(*self*,distance=0.001)

Returns a copy of the colormap where each mapped color is enclosed with two additional mappings at keys with a small offset (±`distance`). This can have the effect of creating "hard borders" in cases where the color values are interpolated between the given mappings.

inverted(*self*)

Returns a copy of the colormap where all colors are inverted.

multiplied(*self*, factor)

Returns a copy of the colormap where all color components are multiplied by the given factor.

remapped(*self*, values)

Returns a new colormap with the given values as keys. The corresponding colors of the new colormap are taken from the original colormap by means of linear interpolation.

repeated(*self*, times, distance=`None`)

Returns a new colormap containing the specified number of copies of the original colormap. The first key of a copy has the specified `distance` from the last key of the previous copy or the original colormap. If no `distance` parameter is given, the distance between the last key and the next to last key is used. If there is only one key, the distance is set to one.

reversed(*self*)

Returns a copy of the colormap with the sequence of colors being reversed, that is, the last color is associated with the first key, the next to last color is associated with the second key, and so on.

transformed(*self*, min, max)

Returns a copy of the colormap where the key values are affinely transformed, such that the new given minimum and maximum values are reached and the relative distances between the keys are kept.

C.2.2. colormaps global function

random_cm(size)

Returns a colormap with the given number of mappings to random colors. The key values start at zero with a step size of one.

C.2.3. composition._Composition class

This is the Python class for representing a data composition. The data composition is identified internally by its name. The `composition._Composition` class is derived from the `insight_core.Observer` class, allowing for an update of the internal representation whenever the name of the composition changes. All instances of this class should be created by the `composition.Composition` factory function, which takes care that each unique data composition is represented by the same instance.

Composition(name=`None`)

This is the factory function for the construction of `composition._Composition` objects. It ensures that for each Insight composition at most one instance is created. If the `name` parameter is omitted, the current composition is returned. If there is no current composition, the default name "<new>" is chosen. If the default name, or any specified name, matches no existing composition, a new composition of the given name is created.

__init__(*self*, name)

Constructor of the `composition._Composition` class, taking the name of the composition as argument.

add_property(*self*, prop)

Adds a property to the composition.

get_entity(*self*, name)

Returns an existing entity of the composition with the given name.

remove_property(*self*, prop)

Removes the given composition property from the composition.

entities [*read-only property*]

Returns a list containing the names of all entities of the composition.

name [*read-only property*]

Returns the name of the composition.

properties [*property*]

The list of all properties of the composition can be accessed and set through this property.

C.2.4. `entity._Entity` class

This is the Python class for representing a data entity. The data entity is identified internally by its data composition (a corresponding `composition._Composition` object) and its unique ID. The `entity._Entity` class is derived from the `insight_core.Observer` class, allowing for an update of the internal representation whenever the properties, the output variables, or the output dimensions of the entity change. All instances of this class should be created by the `entity.Entity` factory function, which takes care that each unique data entity is represented by the same instance.

Entity(type=`None`, name=`None`, comp=`None`, id=`None`)

> This is the factory function for the construction of `entity._Entity` objects. It ensures that for each unique data entity at most one instance is created. If the type of the entity is specified, a new entity with the given or a default name is created. If no explicit composition object is given, the default composition is assumed. The ID is automatically assigned, it is not allowed to pass both a valid `type` and an `id` parameter value to the function. If no `type` parameter is given, the function tries to reference an existing data entity, based on either the given ID or the given name.

__init__(*self*, comp, entity_id)

> Constructor of the `entity._Entity` class. Takes the composition and the ID of an existing data entity.

__getattr__(*self*, attr_name)

> This function checks whether the entity has a property, an output variable, an output dimension, a variable request, or a dimension request of the given converted name (in this order), and returns a corresponding Python object.

__setattr__(*self*, attr_name, value)

> This function checks whether the entity has a property, an output dimension, a variable request, or a dimension request of the given converted name (in this order), and performs one of the following actions: The value of the property is set, or the current position of the output dimension is set, or the variable request is connected with a given output variable, or the mapping of the dimension request is set.

get_dim(*self*, dim_name)

> Returns the specified output dimension of this entity.

get_dim_request(*self*, dim_request_name)

> Returns the specified dimension request of this entity.

get_property(*self*, property_name)

> Returns the specified property of this entity.

get_var(*self*, var_name)

> Returns the specified output variable of this entity.

get_var_request(*self*, var_request_name)

Returns the specified variable request of this entity.

set_box_pos(*self*, x, y)

Sets the position of the box representing the entity in the single instance of the `DataCompositionGridView` class to the given x and y coordinates.

set_vis_rep_visibility(*self*, visibility, viewport_id=-1)

Sets the visibility of this entity in the default or the given viewport to the specified value (`True` or `False`).

comp [*read-only property*]

The parent composition of this entity.

description [*read-only property*]

The description (help text) of this entity.

dim_requests [*read-only property*]

A list containing the names of all dimension requests of this entity.

dims [*read-only property*]

A list containing the names of all output dimensions of this entity.

has_vis_rep [*read-only property*]

A flag indicating whether or not this entity provides a visual representation.

id [*read-only property*]

The ID of this entity.

name [*read-only property*]

The name of this entity.

properties [*read-only property*]

A list containing the names of all properties of this entity.

var_requests [*read-only property*]

A list containing the names of all variable requests of this entity.

vars [*read-only property*]

A list containing the names of all output variables of this entity.

C.2.5. `entity.DimRequest` class

This subclass of the `entity.EntityIOObject` class represents a dimension request of a data entity.

set_mapping(*self*, mapping, combine_dims=`True`)

Sets the mapping of this dimension request. The mapping can be either empty (`None`), a single dimension, or a list of multiple output dimensions. The `combine_dims` flag indicates whether multiple dimensions shall be accessed simultaneously (`False`), or by sequentially accessing all valid combinations of positions of the set of dimensions (`True`).

mapping [*property*]

Gets and sets the current mapping of input dimensions. The mapping is represented as a list containing all data dimensions connected to this dimension request. For setting this property, either the value `None`, a single dimension, or a list containing one or more dimensions can be used. If multiple dimensions are set, they are combined in the default way as described above for the `set_mapping` function with `combine_dims=True`.

C.2.6. `entity.EntityIOObject` class

This is the common base class of the classes representing output dimensions, output variables, dimension requests, and variable requests of a data entity. One of the main tasks of this class is to manage the reference to the parent data entity.

__init__(*self*, entity, name)

The constructor of the `entity.EntityIOObject` class takes the parent data entity and the name of the represented output dimension, output variable, dimension request, or variable request as parameters.

entity [*read-only property*]

The parent data entity of this output dimension, output variable, dimension request, or variable request.

name [*read-only property*]

The name of this output dimension, output variable, dimension request, or variable request.

C.2.7. `entity.OutputDim` class

This subclass of the `entity.EntityIOObject` class represents an output dimension of a data entity.

next(*self*)

Increases the dimension's position. Returns `True`, if the dimension is still valid, `False` otherwise.

reset(*self*)

Resets the dimension's position to zero.

max_size [*read-only property*]

The maximum size of the dimension, taking into account all parent dimensions.

pos [*property*]

Gets and sets the current position of the dimension.

range [*property*]

Gets and sets the restricted range of the dimension as a tuple containing two elements.

real_size [*read-only property*]

The current real size of the dimension, not taking into account any range restrictions.

size [*read-only property*]

The current size of the dimension.

valid [*read-only property*]

A flag indicating whether or not the dimension's current position is valid.

C.2.8. `entity.OutputVar` class

This subclass of the `entity.EntityIOObject` class represents an output variable of a data entity.

__getattr__(*self*, attr_name)

Returns the associated output dimension of the given converted name, if such a dimension exists.

__setattr__(*self*, attr_name, value)

Sets the position of the associated output dimension of the given converted name to the given value.

begin_access(*self*)

Resets the positions of all associated fixed dimensions and prepares the output variable for being accessed. This preparation is necessary prior to any access to the current value, the current position, or the grid index (if available) of the variable.

end_access(*self*)

Ends the variable access and frees some associated resources that are created for the variable access. Some variables may also free the memory for storing the variable values.

fix_dim(*self*, dim, pos)

Marks the given dimension as fixed to the given position, with respect to this particular variable access. This function needs to be called after the `begin_access` and prior to the `end_access` functions. After all desired dimensions are fixed, the next access to the value, the position, or the grid index triggers the preparation of the variable with the new fixed dimension settings.

get(*self*, *args, **kwargs)

This function returns the value of the data variable. If no parameters are specified, the value with respect to the current positions of the dimensions is returned. All simple function parameters are interpreted as dimension indices and are passed to the dimensions associated with this variable in the sequence of the association. For all parameters given with a keyword (e.g. `dim_x`=23), the keyword is interpreted as the name of the dimension for which the index shall be set.

dims [*read-only property*]

The list of names of the dimensions associated with this output variable.

index [*read-only property*]

The current grid index of the output variable. This property is only available as long as the variable is being accessed.

pos [*read-only property*]

The current position associated with the output variable. This property is only available as long as the variable is being accessed.

value [*read-only property*]

The current value of the output variable. This property is only available as long as the variable is being accessed.

C.2.9. `entity.VarRequest` class

This subclass of the `entity.EntityIOObject` class represents a variable request of a data entity.

__getattr__(*self*, attr_name)

Offers direct access to the dimension requests associated with this variable request.

connect(*self*, var)

Connects the given data variable to this variable request.

disconnect(*self*)

Removes any connected data variable from this variable request.

dim_requests [*read-only property*]

A list of names of the dimension requests associated with this variable request.

C.2.10. `entity_property.Property` class

This class represents a property of a data entity. These properties can optionally be associated with a data composition. Internally, the property is represented by the name of the

entity's data composition, the ID of the entity, and the unique name of the property.

__init__(*self*, composition_name, entity_id, property_name)

The constructor of the `entity_property.Property` class takes the name of the data composition, the ID of the entity, and the name of the property as parameters.

get(*self*)

Returns the value of the property.

get_ext(*self*, name=`None`)

Returns a dictionary containing the value and all extended settings of the property, or a single entry from this dictionary with the specified name.

set(*self*, value)

Sets the value of the property.

set_ext(*self*, values, name=`None`)

Sets the value and the extended settings of the property to the values given in form of a Python `dict`. If the name of a single setting is specified, only this single setting is set to the given value.

description [*read-only property*]

The description (help text) of this property.

entity [*read-only property*]

The parent data entity of this property.

name [*read-only property*]

The name of this property.

type [*read-only property*]

A string identifying the type of this property.

C.2.11. `globals.InsightException` class

This is a custom exception class for representing all exceptions regarding Insight. It is derived from the built-in `Exception` Python class and has no custom public interface.

C.2.12. `globals` global functions

convert_name(name)

This function converts the original names of objects such as data entity properties, output dimensions, output variables, etc. into valid Python attribute names that can be used for accessing these objects via corresponding `__getattr__` and `__setattr__` member functions.

export_netcdf(filename, var_list, int_property_dim=`None`, slice_size=0)

This function exports a netCDF file of the given name with the specified set of variables and one optional dimension based on a given integer property. If the additional dimension is included, a `slice_size` value greater than zero results in a set of output files containing the specified number of dimension steps per file.

plot_northern_hemisphere(filename)

This function adjusts the camera, projection, and transformation settings of the current viewport and takes a screenshot covering exactly the area of the northern hemisphere using an equidistant cylindrical map projection.

C.2.13. `user_query_property.UserQueryProperty` class

This class represents a property as provided and managed by instances of the `insight_core.UserQuery` class. Its interface is a subset of the interface of the `entity_property.Property` class. The implementation, however, is different, since this class uses the `insight_core.UserQuery` interface, whereas the entity property uses the global `insight_core` functions for providing its functionality.

__init__(*self*, user_query, property_name)

The constructor of the `user_query_property.UserQueryProperty` class takes the `insight_core.UserQuery` object and the unique name of the property as parameters.

get(*self*)

Returns the value of the property.

get_ext(*self*, name=`None`)

Returns a dictionary containing the value and all extended settings of the property, or a single entry from this dictionary with the specified name.

set(*self*, value)

Sets the value of the property.

set_ext(*self*, values, name=`None`)

Sets the value and the extended settings of the property to the values given in form of a Python `dict`. If the name of a single setting is specified, only this single setting is set to the given value.

name [*read-only property*]

The name of this property.

user_query [*read-only property*]

The associated `insight_core.UserQuery` object.

D. Example script for a simple Insight-WMS-server

This section contains the full script that implements a rudimentary WMS server which uses Insight for the visualization.

```
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
from cgi import parse_qs
from urlparse import urlparse

# setup of the entities
ncdf = Entity("NetCDF time series source")
ncdf.netcdf_file = 
    "P20070114_00;P20070114_06;P20070114_12;P20070114_18"

cm = Entity("Color mapper")
cm.mapping = rainbow.transformed(-20, 30)
cm.legend_pos = "none"

sg = Entity("Simple grid")
sg.fixed_vertical_position = True

coasts = Entity("2d-model data source")
coasts.model_file = 'daten/earth.xml'

coasts_cs = Entity("Color setter")
coasts_cs.color = (0.8, 0.8, 0.8)

lsr = Entity("Line strip representation")

slp_cl = Entity("Contour lines")
slp_cl.value = ncdf.slp

slp_contour_cs = Entity("Color setter")
slp_contour_cs.color = (1.0, 1.0, 1.0)

slp_contour_lsr = Entity("Line strip representation")
slp_contour_lsr.line_thickness = 2

# connect the entities
cm.datain = ncdf.t
sg.color = cm.color
```

```
coasts_cs.value = coasts.vertex
lsr.color = coasts_cs.color

slp_contour_cs.value = slp_cl.iso_lines
slp_contour_lsr.color = slp_contour_cs.color

sg.is_visible = False
lsr.is_visible = False
slp_cl.is_visible = False
slp_contour_lsr.is_visible = False

# invariant viewport settings
set_draw_cube(False)
set_draw_axes(False)
set_background_color((0,0,0))

set_camera((0,90,0))
set_transformation("xyz-box_transformation")
set_projection("ortho")

# dynamic viewport settings
def prepare_camera(resx, resy, left, right, top, bottom):
  ratio = float(resx) / resy

  x_scaling = float(top - bottom) / (right - left) * ratio
  set_scaling((x_scaling, 1.0, 1.0))

  zoom_factor = 40.0 / (top - bottom)
  set_zoom_factor(zoom_factor)
  set_translation((-ratio / zoom_factor - left * x_scaling / 20.0,
                  1.0 / zoom_factor - top / 20.0,
                  0 ))

class MyHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    params = dict([[k, v[0]] for k, v in
        parse_qs(urlparse(self.path)[4]).iteritems()])

    try:
      if params['REQUEST'] == 'GetMap':
        # obtain the bounding box borders and the resolution
        bbox = [float(v) for v in params['BBOX'].split(',')]
        resx = int(params['WIDTH'])
        resy = int(params['HEIGHT'])

        prepare_camera(resx, resy, bbox[0], bbox[2], bbox[1],
            bbox[3])
```

```
        if params['LAYERS'] == 'map':
          sg.is_visible = True
          take_screenshot('wms_image.png', resx, resy)
          sg.is_visible = False
        elif params['LAYERS'] == 'coastlines':
          lsr.is_visible = True
          take_screenshot('wms_image.png', resx, resy)
          lsr.is_visible = False
        elif params['LAYERS'] == 'contour':
          slp_cl.is_visible = True
          slp_contour_lsr.is_visible = True
          take_screenshot('wms_image.png', resx, resy)
          slp_contour_lsr.is_visible = False
          slp_cl.is_visible = False
        else:
          self.send_error(500)
          return

        # return the image
        image = open('wms_image.png', 'rb')
        self.send_response(200)
        self.send_header('Content-type', 'image/png')
        self.end_headers()
        self.wfile.write(image.read())
        image.close()
      elif params['REQUEST'] == 'GetCapabilities':
        # we load the capabilities from an XML file
        xml_file = open('capabilities.xml', 'r')
        self.send_response(200)
        self.send_header('Content-type', 'text/xml')
        self.end_headers()
        self.wfile.write(xml_file.read())
        xml_file.close()
      else:
        self.send_error(500)
    except KeyError:
        self.send_error(500)

try:
  server = HTTPServer(('', 8000), MyHandler)
  server.serve_forever()
except KeyboardInterrupt:
  server.socket.close()
```

E. Full script for the idealized cyclones case study

The following script produces the sequence of images that was used for the core animation in the video showcasing the results of the work on simulations of idealized dry and moist cyclone development by Schemm et al. (2012).

```
#----------------------------------------------
# Script for the production of the
# animation of the cyclone data series
#----------------------------------------------

import datetime
import glob

#----------------------------------------------
# General settings
#----------------------------------------------

# start date of trajectory data
startdate      = datetime.datetime(2010, 1, 13, 0)

# setting the path to the netCDF files and
# to the file containing all trajectories
ncdf_p_path         = 'case_study/P_data'
ncdf_s_path         = 'case_study/S_data'
lsl_file            = 'case_study/Lsl/Day3-5_600hPa'

# output name of plot
plot_prefix         = "idealWCB_"

# obtain all input P and S files
input_p_files       = sorted(glob.glob(ncdf_p_path + '/P*'))
input_s_files       = sorted(glob.glob(ncdf_s_path + '/S*'))
assert(len(input_p_files) == len(input_s_files))

# obtain the dates from the P files
dates = [datetime.datetime.strptime(p_file, ncdf_p_path + 
    '/P%Y%m%d_%H%M') for p_file in input_p_files]
# the S files should have the same dates
assert(dates == [datetime.datetime.strptime(s_file, ncdf_s_path 
    + '/S%Y%m%d_%H%M') for s_file in input_s_files])
```

```
#------------------------------
# Basic camera setup
#------------------------------

# do not draw the cube/axes
set_draw_cube(False)
set_draw_axes(False)

# set background color
set_background_color((0.8, 0.8, 0.8))

# set vertical stretching
set_vertical_ratio(2.5)

# set pole correction
set_pole_correction(-220, 50)

#----------------------------------------
# Setting up all data sources
#----------------------------------------

ncdf_p = Entity("NetCDF time series source")
ncdf_p.netcdf_file = ';'.join(input_p_files)

ncdf_s = Entity("NetCDF time series source")
ncdf_s.netcdf_file = ';'.join(input_s_files)

lsl = Entity("Trajectories data source")
lsl.trajectories_file = lsl_file
# we only consider every fourth trajectory
lsl.frequency = 4

# truncate the domain
# x dimension
ncdf_p.dimx_u.range = (30,600)
ncdf_s.dimx_p.range = (30,600)
# y dimension
ncdf_p.dimy_u.range = (50,500)
ncdf_s.dimy_p.range = (80,350)

#----------------------------------------
# Preparing the visual representations
#----------------------------------------

# a) PV on TH surface

# -- define the PV colormap
```

```
cmpv                    = Entity("Color mapper")
cmpv.legend_pos         = "rightmost"
cmpv.caption            = "[PVU]"
cmpv.mapping            = pv_color.transformed(0, 4)
cmpv.text_size          = 20
cmpv.text_color         = (0.0,0.0,0.0)

# -- create the TH isosurface
iso                     = Entity("[wrapper] Iso surface")
iso.value               = ncdf_s.th
iso.iso_value           = 317
iso.use_fixed_color     = False
iso.transparency        = 0.4

# -- color the TH isosurface with PV
cmpv.datain             = ncdf_s.pv
iso.color_optional      = cmpv.color

# b) pressure at surface

ps          = Entity("Contour lines")
ps.value                = ncdf_p.ps
ps.start                =  960
ps.end                  = 1040
ps.delta                =   10
ps.line_thickness       =    1
ps.text_color           = (0.0, 0.0, 0.0)
ps.draw_captions        = False

colorset                = Entity("Color setter")
colorset.value          = ps.iso_lines
colorset.color          = (0.0, 0.0, 0.0)

linestrip               = Entity("Line strip representation")
linestrip.color         = colorset.color
linestrip.line_thickness = 2

# c) isosurface: latent heating

# -- create the isosurface
isolh                   = Entity("[wrapper] Iso surface")
isolh.value             = ncdf_p.latentheat
isolh.iso_value         = 1.5
# keep it default in red

# d) isosurface: saturated air

isosat                  = Entity("[wrapper] Iso surface")
```

```
isosat.value        = ncdf_p.qc
isosat.iso_value    = 0.2
isosat.color        = (0.6, 0.95, 0.95)
isosat.transparency = 0.25

# e) potential temperature at surface

th                  = Entity("Simple_grid")
th.fixed_vertical_position = True

thcmp               = Entity("Color_mapper")
thcmp.datain        = ncdf_s.th
thcmp.legend_pos    = "none"
thcmp.mapping       = colorful.transformed(255,300)

th.color            = thcmp.color

# f) jet stream

u                   = Entity("Contour_lines")
u.value             = ncdf_p.u
u.start             =  20
u.end               =  80
u.delta             =  10
u.line_thickness    =   1
u.text_color        = (1, 1, 1)
u.draw_captions     = False
u.dimz_u            = 55

ucolorset           = Entity("Color_setter")
ucolorset.value     = u.iso_lines
ucolorset.color     = (1.0, 1.0, 1.0)

ulinestrip          = Entity("Line_strip_representation")
ulinestrip.color    = ucolorset.color
ulinestrip.line_thickness = 2

# g) WCB-trajectories

colortra            = Entity("Color_mapper")
colortra.datain     = lsl.time
colortra.legend_pos = "none"
colortra.mapping    = chess.remapped(range(0,2)).repeated(24)

lines               = Entity("Line_strip_representation")
lines.color         = colortra.color
lines.draw_cylinder_curves = True
lines.cylinder_thickness = 0.3
```

```
# h) plot caption

cap                    = Entity("Caption")
cap.point_size         = 22
cap.x                  = -0.9
cap.y                  =  0.8
cap.color              = ((0.0,0.0,0.0))

#------------------------------------------------
# Creation of all animation frames
#------------------------------------------------

for timestep in range(0, len(input_p_files)):
    # set the timestep of the visual representations...
    iso.time = timestep
    isolh.time = timestep
    isosat.time = timestep

    th.time = timestep
    ps.time = timestep

    # loop step for trajectories
    date    = dates[timestep]
    # find the trajectory time step matching the current date
    traj_diff = date - startdate
    t = traj_diff.days * 24 + traj_diff.seconds / 3600
    lsl.sample.range = (0,int(t))

    # zoom the camera into the plot
    set_zoom_factor(4.2 + timestep * 0.065)

    # set the camera angle
    set_camera((20 + timestep * 0.5, 18 - timestep * 0.02, 0))

    # set a camera translation
    set_translation((-0.22, 0.1 - timestep * 0.001, 0.09))

    # update the "date" caption
    cap.caption = "Day " + str(date.day) + " - Hour " + 
        str(date.hour).zfill(2)

    # finally, we take the screenshot
    take_screenshot(plot_prefix + str(int(date.day)-10) + 
        str(date.hour).zfill(2) + '.png', 2560, 1600)

# do not close the window at end of loop
insight_core.window_stay_open_hint = True
```

F. Maintenance of IWAL

This section covers short descriptions of the basic steps for deploying an IWAL server, including the creation and initialization of the database. Additionally, this section covers the utilization of South for applying changes to the database background of IWAL.

F.1. Deployment of IWAL

F.1.1. Database creation

We start the setup of the IWAL PostgreSQL database by creating a new database user called "iwal". For this, we use the interactive PostgreSQL terminal, which we start as the database administrator:

```
:~$ sudo -u postgres psql
postgres=# CREATE USER iwal WITH PASSWORD '<password>';
```

The string "<password>" should be replaced with an actual password. We assume that the selected mode of user authentication given in the `pg_hba.conf` file of the PostgreSQL installation is `md5`, not `ident`, such that no Linux user of the given name has to exist.

Next, we create a new database and give the new user the required privileges.

```
postgres=# CREATE DATABASE iwaldb;
postgres=# GRANT ALL PRIVILEGES ON DATABASE iwaldb TO iwal;
```

F.1.2. Setup and local testing

The next step for installing and deploying IWAL is to obtain an up-to-date copy of the source code from the subversion repository:

```
:~$ svn co http://insight.zdv.uni-mainz.de/iwal/svn/trunk iwal
```

Before we can invoke the creation of the initial database tables and entries, we have to configure the database access and several other settings by editing the `iwal/settings.py` file. Additionally, the path to the static files has to be set in the `iwal/urls.py` file. As soon as the configuration of IWAL is complete, we are able to create an initial version of our database:

```
:~$ cd iwal
:~/iwal$ ./manage.py syncdb
```

When Django proposes the creation of a superuser for the `auth` Django application, we accept and enter an arbitrary name and password for the administrator of IWAL.

At this point, we are able to start the integrated test server of Django and access the database administration interface.

```
:~/iwal$ ./manage.py runserver 8080 &
:~/iwal$ firefox localhost:8080/admin
```

The next step is to populate the database with all information about the available modules, case studies, courses, and so on. A minimum set of required information contains:

- the name of at least one module in the `wmsclient_module` table, e.g. “horizontalcsfree”,
- the name and priority of at least one field (data variable) in the `wmsclient_field` table, e.g. “T” with a priority of 10,
- and finally, at least one entry in the `wmsclient_casestudy` table representing a case study with the name “Default”, containing a reference to at least one module and a valid setup of input data.

Every time the names or the set of available case studies changes, the IWAL server has to be restarted. After that, we can access IWAL on the Python test server by visiting `localhost:8080/login`.

F.1.3. Apache configuration

For deploying IWAL on an Apache server with mod_wsgi, we first need to create the entry point of the WSGI-application in form of an additional Python file. We could, for example, create the file `/var/www/IWAL/django.wsgi` with the following content:

```
import os
import sys

# The absolute path to the IWAL source, for example:
path = '/home/limbach/iwal'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'

import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

We have to ensure that Apache has proper read access to this configuration file, to all IWAL source files, and all static files to be served. Finally, we have to add the following line to the Apache configuration:

```
WSGIScriptAlias / /var/www/IWAL/django.wsgi
```

After a restart of the Apache server, IWAL is accessible at `<server-location>/login`, e.g. `localhost/login`.

The configuration of Apache and mod_wsgi can be further improved in order to obtain a better performance of IWAL. For example, mod_wsgi can be run in *daemon mode*, where the execution of the application is divided into distinct processes with a configurable number of threads. On the `iwal.ethz.ch` server, the corresponding part of the configuration file looks as follows:

```
WSGISocketPrefix run/wsgi
WSGIDaemonProcess iwal.ethz.ch user=apache group=apache ↩
   ↪processes=16 threads=5 display-name=%{GROUP}
WSGIProcessGroup iwal.ethz.ch
```

One additional advantage of the daemon mode is that a restart of the Django application can be forced without restarting the whole Apache server, simply by touching the `django.wsgi` file.

Another improvement of the performance can be achieved by letting Apache, and not the Django applications, serve all static files directly. For this, the respective lines in the `iwal/urls.py` need to be removed and the Apache server needs to be configured accordingly, for example by adding the following line to the configuration file:

```
Alias /static /home/limbach/iwal/static
```

F.2. Example database migration

In this section, we demonstrate how an existing database model can be migrated to an updated version using South. For the following example, we want to add the new attribute `favorite_case_study` to the `wmsclient.UserProfile` model. We use South for managing, creating, and applying the database migrations. First of all, we have to create the initial migration from the empty database scheme to the current state of our database. This initial migration has to be performed once for every distinct Django application of which we want to change the database model. Here, we apply it only to the `wmsclient` application:

```
:~/iwal$ ./manage.py convert_to_south wmsclient
```

The next step is to change the actual model. For this, we add the following line to the file `iwal/wmsclient/models.py`

```
    favorite_case_study = models.ForeignKey(Casestudy, ↩
        ↪null=True, blank=True, related_name='+')
```

Next, we let South create a database migration automatically:

```
:~/iwal$ ./manage.py schemamigration wmsclient --auto
```

Finally, we are ready to apply this migration:

```
:~/iwal$ ./manage.py migrate wmsclient
```

Bibliography

Abrahams, D. and R. W. Grosse-Kunstleve. Building Hybrid Systems with Boost.Python. *C/C++ Users Journal*, July 2003. URL `http://www.boostpro.com/writing/bpl.html`. last accessed: 14-June-2013.

Aebi, C. *Climatological Analysis of the Interaction of PV-Streamer and Warm Conveyor Belt*. Master's thesis, Universität Bern, 2012.

Appenzeller, C. and H. C. Davies. Structure of stratospheric intrusions into the troposphere. *Nature*, 358:570–572, 1992.

Bachmann, J. *Untersuchung der Vorhersage von Zugbahn und Struktur außertropischer Tiefdruckgebiete*. Master's thesis, Johannes Gutenberg-Universität Mainz, 2010.

Bauer, D. and R. Peikert. Vortex tracking in scale-space. In *Proceedings of the symposium on Data Visualisation 2002*, VISSYM '02, pages 233–240. Eurographics Association, 2002.

de la Beaujardiere, J., editor. *Web Map Service Implementation Specification, Version 1.1.1*. Open Geospatial Consortium Inc., 2002. URL `http://www.opengeospatial.org/standards/wms`. last accessed: 14-June-2013.

de la Beaujardiere, J., editor. *OpenGIS Web Map Server*[sic!] *Implementation Specification, Version 1.3.0*. Open Geospatial Consortium Inc., 2006. URL `http://www.opengeospatial.org/standards/wms`. last accessed: 14-June-2013.

Bell, G. D. and D. Keyser. Shear and Curvature vorticity and Potential-Vorticity Interchanges: Interpretation and Application to a Cutoff Cyclone Event. *Mon. Wea. Rev.*, 121:102–121, 1993.

Berners-Lee, T. and D. Connolly. Hypertext Markup Language (HTML): A Representation of Textual Information and MetaInformation for Retrieval and Interchange. *Internet Draft*, June 1993. URL `http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt`. last accessed: 14-June-2013.

Bjerknes, J. On the structure of moving cyclones. *Mon. Wea. Rev.*, 47(2):95–99, 1919.

Blanchette, J. and M. Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall Press, 2008.

Blender, R. and M. Schubert. Cyclone Tracking in Different Spatial and Temporal Resolutions. *Mon. Wea. Rev.*, 128:377–384, February 2000.

Bray, T., J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, Recommendation REC-xml-20081126, 2008. URL `http://www.w3.org/TR/REC-xml/`. last accessed: 14-June-2013.

Browning, K. A. Organization of clouds and precipitation in extratropical cyclones. In Newton, C. and E. Holopainen, editors, *Extratropical Cyclones, The Erik H. Palmen Memorial Volume*, pages 129–153. American Meteorological Society, 1990.

Campa, J. *Potential vorticity and moisture in extratropical cyclones: climatology and sensitivity studies*. PhD thesis, Johannes Gutenberg-Universität Mainz, 2012.

Charney, J. G., R. Fjörtoft, and J. von Neumann. Numerical Integration of the Barotropic Vorticity Equation. *Tellus*, 2(4):237–254, 1950.

Chen, P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

Childs, H., E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A Contract Based System For Large Data Visualization. In *IEEE Visualization 2005*, pages 191–198, 2005.

Chin, N., C. Frazier, P. Ho, Z. Liu, and K. P. Smith. *The OpenGL Graphics System Utility Library (Version 1.3)*. Silicon Graphics, Inc., 1998. URL `http://www.opengl.org/registry/doc/glu1.3.pdf`. last accessed: 14-June-2013.

Clyne, J., P. Mininni, A. Norton, and M. Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New J. Phys.*, 9(8):301, 2007. URL `http://iopscience.iop.org/1367-2630/9/8/301/pdf/1367-2630_9_8_301.pdf`. last accessed: 14-June-2013.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, pages 568–572. The MIT Press, 2009.

Davies, E. R. *Machine Vision, Third Edition: Theory, Algorithms, Practicalities (Signal Processing and its Applications)*. Morgan Kaufmann, January 2005.

Dee, D. P., S. M. Uppala, A. J. Simmons, P. Berrisford, P. Poli, S. Kobayashi, U. Andrae, M. A. Balmaseda, G. Balsamo, P. Bauer, P. Bechtold, A. C. M. Beljaars, L. van de Berg, J. Bidlot, N. Bormann, C. Delsol, R. Dragani, M. Fuentes, A. J. Geer, L. Haimberger, S. B. Healy, H. Hersbach, E. V. Hólm, L. Isaksen, P. Kållberg, M. Köhler, M. Matricardi, A. P. McNally, B. M. Monge-Sanz, J.-J. Morcrette, B.-K. Park, C. Peubey, P. de Rosnay, C. Tavolato, J.-N. Thépaut, and F. Vitart. The ERA-Interim reanalysis: configuration and performance of the data assimilation system. *Quart. J. Roy. Meteor. Soc.*, 137(656):553–597, 2011.

Folk, M., G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47. ACM, 2011.

Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

Freeman, El., Er. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

Friedman, R. M. *Appropriating the Weather: Vilhelm Bjerknes and the Construction of a Modern Meteorology*. Cornell Paperbacks. Cornell University Press, 1993.

Fritz, S. Pictures From Meteorological Satellites and Their Interpretation. *Space Science Reviews*, 3 (4):541–580, 1964.

Fuchs, R., R. Peikert, H. Hauser, F. Sadlo, and P. Muigg. Parallel vectors criteria for unsteady flow vortices. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):615–626, May–June 2008.

Gamma, E., R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

Gregory, J. The CF metadata standard. *CLIVAR Exchanges No. 28*, 8(4), 2003. also available at URL `http://cf-pcmdi.llnl.gov/documents/other/cf_overview_article.pdf`, last accessed: 14-June-2013.

Henderson, A. *ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 2008.

van Herk, M. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recogn. Lett.*, 13(7):517–521, July 1992.

Hibbard, W. L. and D. A. Santek. The VIS-5D system for easy interactive visualization. In *IEEE Visualization 1990*, pages 28–35, 1990.

Holovaty, A. and J. Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Apress, 2009. released as *The Django Book* at URL `www.djangobook.com/en/2.0/index.html`, last accessed: 14-June-2013.

Ibáñez, L., W. Schroeder, L. Ng, J. Cates, and the Insight Software Consortium. *The ITK Software Guide, Second Edition, Updated for ITK version 2.4*. Kitware Inc., 2005. URL `http://www.itk.org/ItkSoftwareGuide.pdf`. last accessed: 14-June-2013.

Jain, R., R. Kasturi, and B. G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

Ji, G., H. Shen, and R. Wenger. Volume tracking using higher dimensional isosurfacing. In *IEEE Visualization 2003*, pages 209–216, October 2003.

Johnson, C. Top scientific visualization research problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, 2004.

Kleinschmidt, E. Über Aufbau und Entstehung von Zyklonen (1. Teil). *Meteor. Rundschau*, 3:1–6, 1950.

Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*, chapter 2.3.3, pages 354–355. Addison-Wesley Professional, July 1997.

Koch, P., H. Wernli, and H.C. Davies. An event-based jet-stream climatology and typology. *Int. J. Climatol.*, 26:283–301, 2006.

König, W., R. Sausen, and F. Sielmann. Objective identification of cyclones in GCM simulations. *J. Climate*, 6:2217–2231, 1993.

Lambert, S. J. A cyclone climatology of the canadian climate centre general circulation model. *J. Climate*, 1:109–115, 1988.

Lawrence, B. N., R. Drach, B. E. Eaton, J. M. Gregory, S. C. Hankin, R. K. Lowry, R. K. Rew, and K. E. Taylor. Maintaining and Advancing the CF Standard for Earth System Science Community Data, 2006. URL `http://cf-pcmdi.llnl.gov/documents/white-papers/cf2_whitepaper_final.pdf`. last accessed: 14-June-2013.

Limbach, S., M. Marto, P. Jöckel, E. Schömer, and H. Wernli. Die Analyse atmospärischer Strömungen. *Natur & Geist - Das Forschungsmagazin der Johannes Gutenberg-Universität Mainz*, 2:22–25, 2009.

Limbach, S., E. Schömer, and H. Wernli. Detection, tracking and event localization of jet stream features in 4-D atmospheric data. *Geosci. Model Dev.*, 5(2):457–470, 2012. URL `http://www.geosci-model-dev.net/5/457/2012/`. last accessed: 14-June-2013.

Mahrous, K., J. Bennett, G. Scheuermann, B. Hamann, and K.I. Joy. Topological segmentation of three-dimensional vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):198–205, 2004.

Manney, G. L., M. I. Hegglin, W. H. Daffer, M. L. Santee, E. A. Ray, S. Pawson, M. J. Schwartz, C. D. Boone, L. Froidevaux, N. J. Livesey, W. G. Read, and K. A. Walker. Jet characterization in the upper troposphere/lower stratosphere (UTLS): applications to climatology and transport studies. *Atmos. Chem. Phys.*, 11:6115–6137, 2011.

Martius, O. and H. Wernli. A Trajectory-Based Investigation of Physical and Dynamical Processes That Govern the Temporal Evolution of the Subtropical Jet Streams over Africa. *J. Atmos. Sci.*, 69:1602–1616, 2012.

Martius, O., C. Schwierz, and H. C. Davies. Tropopause-level waveguides. *J. Atmos. Sci.*, 67:866–879, 2010.

Marto, M. Entwicklung einer Anwendung zur Visualisierung von Wetterdaten. *Bachelor's thesis, Johannes Gutenberg-Universität Mainz*, 2008.

Matula, J. Exploiting Web Services for Meteorological Applications. In *Second workshop on the use of OGC/GIS standards in meteorology*, November 2009.

McCormick, B. H., T. A. DeFanti, and M. D. Brown. *Visualization in Scientific Computing.* ACM, 1987.

Muelder, C. and Kwan-Liu Ma. Interactive feature extraction and tracking by utilizing region coherency. In *IEEE Pacific Visualization Symposium*, pages 17–24, April 2009.

Murray, D., J. McWhirter, S. Wier, and S. Emmerson. The Integrated Data Viewer: A Web-enabled Application for Scientific Analysis and Visualization. In *19th Conference on International Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology.* American Meteorological Society, 2003.

Murray, D., J. McWhirter, Y. Ho, and T. M. Whittaker. The IDV at 5: New Features and Future Plans. In *25th Conference on International Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology.* American Meteorological Society, 2009.

Murray, R. J. and I. Simmonds. A numerical scheme for tracking cyclone centres from digital data. part I: development and operation of the scheme. *Aust. Meteorol. Mag.*, 39:155–166, 1991.

Ng, A. Y., M. I. Jordan, and Y. Weiss. On Spectral Clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, pages 849–856. MIT Press, 2001. URL `http://ai.stanford.edu/~ang/papers/nips01-spectral.pdf`. last accessed: 14-June-2013.

Nickolls, J., I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

Nikhil, R. P. and K. P. Sankar. A review on image segmentation techniques. *Pattern Recognition*, 26 (9):1277–1294, 1993.

Pinto, J. G., T. Spangehl, U. Ulbrich, and P. Speth. Sensitivities of a cyclone detection and tracking algorithm: individual tracks and climatology. *Meteorologische Zeitschrift*, 14(6):823–838, 2005.

Post, F. H., B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22:775–792, 2003.

Raible, C. C., P. M. Della-Marta, C. Schwierz, H. Wernli, and R. Blender. Northern hemisphere extratropical cyclones: a comparison of detection and tracking methods and different reanalyses. *Mon. Weather Rev.*, 136(3):880–897, 2008.

Raoult, B., P. Bispham, A. Brady, J. L. Casado, R. Corresa, S. Lamy-Thepaut, T. Orford, D. Richardson, C. Sahin, S. Siemen, C. Valiente, and D. Varela. ECMWF Web re-engineering project "ecCharts". In *27th Conference on Interactive Information Processing Systems (IIPS).* American Meteorological Society, 2011. URL `https://ams.confex.com/ams/91Annual/webprogram/Paper180465.html`. last accessed: 14-June-2013.

Rautenhaus, M., G. Bauer, and A. Dörnbrack. A web service based tool to plan atmospheric research flights. *Geosci. Model Dev.*, 5(1):55–71, 2012. URL `http://www.geosci-model-dev.net/5/55/2012/`. last accessed: 14-June-2013.

Reas, C. and B. Fry. *Processing: A Programming Handbook for Visual Designers and Artists.* The MIT Press, 2007.

Reinders, F. *Feature-based visualization of time-dependent data.* PhD thesis, University of Delft, 2001.

Reutter, P. *Numerical simulations of microphysical processes in pyro-convective clouds.* PhD thesis, Max Planck Institut for Chemistry Mainz and University of Mainz, 2009.

Rew, R. and G. Davis. Netcdf: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10:76–82, 1990.

Roedel, W. and T. Wagner. *Physik unserer Umwelt: Die Atmosphäre.* Springer-Verlag, 2011.

van Rossum, G. and J. de Boer. Interactively Testing Remote Servers Using the Python Programming Language. *CWI Quarterly*, 4:283–303, 1991.

Rost, R. J. *OpenGL Shading Language.* Addison-Wesley Professional, 2004.

Samtaney, R., D. Silver, N. Zabusky, and J. Cao. Visualizing features and tracking their evolution. *Computer*, 27(7):20–27, July 1994.

Schemm, S., H. Wernli, and L. Papritz. Warm conveyor belts in idealized moist baroclinic wave simulations. *J. Atmos. Sci.*, 70:627–652, 2012.

Schiemann, R., D. Lüthi, and C. Schär. Seasonality and interannual variability of the westerly jet in the Tibetan plateau region. *J. Climate*, 22:2940–2957, 2009.

Schroeder, W., K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Fourth Edition.* Kitware Inc., 2006.

Segal, M. and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.0).* Silicon Graphics, Inc., 1994. URL http://www.opengl.org/registry/doc/glspec10.pdf. last accessed: 14-June-2013.

Segal, M. and K. Akeley. *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004).* Silicon Graphics, Inc., 2004. URL http://www.opengl.org/registry/doc/glspec20.20041022.pdf. last accessed: 14-June-2013.

Segal, M. and K. Akeley. *The OpenGL Graphics System: A Specification (Version 3.0 - September 23, 2008).* The Khronos Group Inc., 2008. URL http://www.opengl.org/registry/doc/glspec30.20080923.pdf. last accessed: 14-June-2013.

Segal, M. and K. Akeley. *The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010).* The Khronos Group Inc., 2010. URL http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf. last accessed: 14-June-2013.

Serra, J. *Image Analysis and Mathematical Morphology.* Academic Press, Inc., 1982.

Severance, C. JavaScript: Designing a Language in 10 Days. *IEEE Computer*, 45(2):7–8, February 2012.

Shapiro, M. A. and S. Grønås, editors. *The Life Cycles of Extratropical Cyclones.* American Meteorological Society, 1999.

Siegesmund, M. *Visuell gestützte Mustererkennung zur Identifikation und Klassifikation von Ozonlöchern.* Master's thesis, Universität Rostock, 2006.

Silver, D. and X. Wang. Tracking and visualizing turbulent 3d features. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):129–141, April–June 1997.

Silver, D. and X. Wang. Visualizing evolving scalar phenomena. *Future Generation Computer Systems*, 15(1):99–108, 1999.

Silver, D. and N.J. Zabusky. Quantifying visualizations for reduced modeling in nonlinear science: Extracting structures from data sets. *J. Vis. Commun. Image Represent.*, 4(1):46–61, 1993.

Steppeler, J., G. Doms, U. Schättler, H. W. Bitzer, A. Gassmann, U. Damrath, and G. Gregoric. Meso-gamma scale forecasts using the nonhydrostatic model LM. *Meteorol. Atmos. Phys.*, 82: 75–96, 2003.

Stonebraker, M. and L. A. Rowe. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355. ACM, 1986.

Tarjan, R. E. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, April 1975.

Thompson, D., J. Braun, and R. Ford. *OpenDX: Paths to Visualization*. Visualization and Imagery Solutions, Inc., 2001.

Ulbrich, U., G. C. Leckebusch, and J. G. Pinto. Extra-tropical cyclones in the present and future climate: a review. *Theor Appl Climatol*, 96:117–131, 2009.

Wallace, J. M. and P. V. Hobbs. *Atmospheric Science: An Introductory Survey*. Academic Press, Inc., 1977.

van Walsum, T. *Selective visualization on curvilinear grids*. PhD thesis, Delft University of Technology, 1995.

Weigle, C. and D.C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *IEEE Symposium on Volume Visualization*, pages 103–110, October 1998.

Wernli, H. and H. C. Davies. A Lagrangian-based analysis of extratropical cyclones. I: The method and some applications. *Quart. J. Roy. Meteor. Soc.*, 123(538):467–489, 1997.

Wernli, H. and C. Schwierz. Surface cyclones in the ERA-40 dataset (1958-2001). part I: Novel identification method and global climatology. *J. Atmos. Sci.*, 63:2486–2507, 2006.

Wernli, H. and M. Sprenger. Identification and ERA15 climatology of potential vorticity streamers and cut-offs near the extratropical tropopause. *J. Atmos. Sci.*, 64:1569–1586, 2007.

Wilhelms, J. and A. van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11: 201–227, July 1992.

Winschall, A. *Die Herkunft des Wassers bei Starkniederschlagsereignissen*. Master's thesis, Johannes Gutenberg-Universität Mainz, 2009.

Woo, M., J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 3rd edition, 1999.

Würsten, F. Gewitter im Rechner. *ETH Globe*, 2:26–28, 2012. URL `http://www.ethz.ch/about/publications/globe/archive/highlights/2012_02/wetterprognosen`. last accessed: 14-June-2013.

Zucker, S. W. Region growing: Childhood and adolescence. *Computer Graphics and Image Processing*, 5:382–399, 1976.