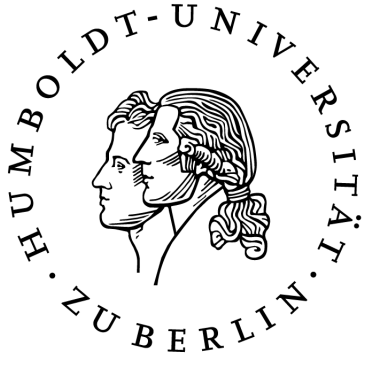
CrusDe: A plug-in based simulation framework for composable CRUStal DEformation simulations using Green's functions



Diploma Thesis
in partial fulfillment of the requirements for the degree
Diplominformatiker (Dipl.-Inf.)

submitted to the Department of Computer Science
of the Humboldt-University Berlin

by Ronni Grapenthin
born on 09.01.1979
in Jena

Advisors: Prof. Dr. Joachim Fischer, Humboldt-Universität zu Berlin
Dr. Freysteinn Sigmundsson, Nordic Volcanological Center, Reykjavík

Date: July 31, 2007

Kurzfassung

Ein bewährtes Werkzeug der Geowissenschaften zur Analyse der Dynamik der Lithosphäre in Reaktion auf eine Massenkraft ist die Greensche Methode. Hierbei wird eine spezielle Lösung einer inhomogenen partiellen Differentialgleichung mit Randbedingungen, die so genannte Greensche Funktion, die vom dynamischen System abhängt, mit einer in Einheitsmassenimpulsen vorliegenden Auflast gefaltet. Verschiedene Abstraktionen vom Erdinneren und den Effekten, die eine Auflast erzeugt, können mit Hilfe Greenscher Funktionen formuliert werden. Die Form der Auflast wird entweder durch ein empirisches oder ein analytisches Modell beschrieben. Alle Auflastbeschreibungen und Greensche Funktionen könnten innerhalb einer Softwareinfrastruktur benutzt werden, wäre die Greensche Methode derart implementiert, dass sie von der Semantik der Faltungsoperanden abstrahiert.

Die Diplomarbeit untersucht die Möglichkeit, die Greensche Methode so zu implementieren, dass Realisierungen ihrer Bausteine (Greensche Funktion, Auflastfunktion, Faltungsoperator) frei komponierbar werden. Hierzu wird die Softwarearchitektur eines Modell-Simulationsframeworks konzipiert und realisiert. Schnittstellen zur generischen Kommunikation zwischen den einzelnen Modellbausteinen werden definiert und damit die Austauschbarkeit spezifischer Implementationen ermöglicht.

Zum Test des entwickelten Plug-in basierten Simulationsframeworks CrusDe[1] wird eine Fallstudie durchgeführt. Regionale Deformationsmuster der Erdkruste, die von den Lavafeldern herrühren, die während des Ausbruchs des isländischen Vulkans Hekla im Jahr 2000 entstanden, demonstrieren den Nutzen des gewählten Ansatzes für das Simulationsframework. Probleme wie Nachnutzung von Softwarekomponenten und nutzerbestimmte Komposition von Simulationsmodellen sind in der vorgeschlagenen Architektur für die spezielle Domäne gelöst.

Die Arbeit liefert für das in der Programmiersprache C++ für Linux-Umgebungen entwickelte Simulationsframework CrusDe die folgenden Beiträge:

- die Konzeption und Implementation einer parametrisierbaren, durch Plug-ins erweiterbaren Infrastruktur zur Simulation der Effekte, die beliebige Auflasten in beliebigen geographischen Regionen auf die Erde ausüben,

- es wird eine unabhängige, nutzerbestimmte Auswahl der Modellbausteine für die Greenschen Methode ermöglicht,

- die Beschreibung von Simulationen erfolgt durch Experimentdefinitionen in externen XML-Dateien und

- Tests und Nachweis der Glaubwürdigkeit der Simulationsergebnisse durch Vergleich mit analytischen Lösungen und Resultaten früherer Studien.

[1] `http://www.grapenthin.org/projects/crusde`

Die Simulationsergebnisse der Fallstudie zeigen, dass durch Lavafelder hervorgerufene Effekte in der Umgebung von Vulkanen bei der Untersuchung von Deformationsmustern Beachtung finden müssen. Es ergeben sich folgende Erkenntnisse:

- Langzeitabsenkungen durch Auflast weisen Deformationsmuster auf, die den durch simulierten Druckabfall in Magmakammern hervorgerufenen ähneln; es kann zu Verwechslungen kommen,
- sollen Eigenschaften einer oberflächennahen Magmakammer ermittelt werden, müssen die Daten entsprechend der in näherer Umgebung vorhandenen Auflast korrigiert werden.

Abstract

Within geoscience, Green's method is an established mathematical tool to analyze the dynamics of the lithosphere in response to a mass force. A particular solution for an inhomegenous differential equation with boundary conditions, the so-called Green's function, is convolved with a surface load expressed by unit point masses. A Green's function depends on the specifics of the examined dynamic system. Different abstractions from the Earth's interior and from the effects caused by a load are expressed by means of Green's functions. Load shapes are defined by an either empirical or analytical model. All load and Green's functions could be utilized within a single software infrastructure if Green's method was implemented on a level that abstracts from the semantics of the convolution operands.

This thesis examines the possibility to implement Green's method in a way that realizations of its elements (Green's function, load function, convolution operator) are freely composable. A software architecture for a model simulation framework is proposed and implemented. This architecture defines interfaces for generic communication between the model elements and thus enables exchangeability of specific implementations.

A case study is conducted to test the developed plug-in based simulation framework CrusDe[2]. Regional crustal deformation patterns resulting from the lava flows that emerged during the year 2000 eruption of the Icelandic volcano Mt. Hekla demonstrate the usefulness and accuracy of the proposed approach. Problems such as software component reuse and user-driven simulation model composition are solved for the specific domain by the proposed architecture.

This work contributes the following to the development of CrusDe which is written for Linux environments in the programming languages C/C++:

- conception and implementation of a plug-in based infrastructure for modeling the surface displacements due to arbitrary loads in any region of the Earth,
- independent, user-driven selection of elements participating in Green's method,
- description of simulations in experiment definitions in external XML-files, and
- tests and proof of credibility by comparisons to analytical solutions and to the results of previous studies.

The model results of the case study show that surface loads cause considerable deformation around volcanoes and thus must be considered when deformation patterns in such areas are studied. The following conclusions are drawn:

- long term subsidence due to surface loads shows a deformation pattern similar to pressure decrease in shallow magma chambers; a mix-up is possible,
- if the characteristics of a shallow magma chamber are to be derived, the data must be adjusted to the effects caused by surface loads in the vicinity.

[2] `http://www.grapenthin.org/projects/crusde`

Contents

List of Figures

Listings

List of Tables

List of Abbreviations

API	application programming interface
COARDS	Cooperative Ocean/Atmosphere Research Data Service
DB	Database
DFT	discrete Fourier transform
DTD	Document Type Definition
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
IF	interface
InSAR	interferometric synthetic aperture radar
KISS	keep it small and simple
FFT	fast Fourier transform
FFTW	Fastest Fourier Transform in the West
I/O	input/output
LSI-system	linear space invariant system
MB	mega byte
PSF	plug-in based simulation framework
RAM	random access memory
SAR	synthetic aperture radar
SI-system	Système international d'unités
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

"Programming is legitimate and necessary academic endeavor."
(Donald E. Knuth)

1.1 Motivation & scientific context

Modeling and simulation play an important role in gaining a better understanding of processes that work hidden from the human eye in the interior of the Earth. Acquired data that might reveal the driving forces of our observations is usually punctual in either space, or time, or both. A Global Positioning System (GPS) receiver, for instance, might calculate its position at an arbitrarily high frequency, but it measures the movements of a single point only. Satellite imagery, on the other hand, provides a high spatial resolution of the Earth's surface, but observations are only repeated on the order of days or weeks. Furthermore, a multitude of (interacting) processes usually causes anomalies in geo-data. In words of *Oreskes et al.* (1994), geo-data are "inference-laden signifiers of natural phenomena to which we have incomplete access."

Modeling and simulation – carefully applied – provide helpful means to fill data gaps. With the help of mathematical expressions results are obtained that fit the recorded data and are continuous over time and space. These expressions represent a simplified reality and yet, the identification of processes that contribute to a signal might be possible. Thus, modeling and simulation may support theories which in turn help to explain observations.

In the discipline of crustal deformation which is concerned with responses of

the Earth's crust to endogen (e.g. magmatic) and exogen (e.g. glacial) processes, Green's method is a frequently used mathematical tool for analyzing the dynamics of the Earth's crust in response to mass forces (surface loads). In this method a particular solution of an inhomogeneous partial differential equation, the so-called Green's function, is convolved with a mass force that acts on the crust. This surface load is suspected to induce the observed effect on the Earth's surface. Since Green's functions that represent a simplified understanding of the Earth's inner structure can be constructed, they are used to mimic loading related effects. The result of Green's method is an estimate for the response of the Earth in a specified region to the applied force. Such a result could be, for instance, uplift due to glacial melting (with the surface load being a glacier).

Availability of models, not only in the form of mathematical formulations but also as executable software, gets more important the more scientific findings rely on them. This is also true for the Earth and surface load models that are employed in Green's method. Analytical solutions for this method are obtainable – if at all – only with great effort. As soon as numerical methods are used to compute a result it is important to provide the applied methods; especially as source code. Not only that programming errors are identified more quickly this way, but effortless reproduction of results and questioning of applied methods allow for fertile discussions about the findings and thus improve inferred theories.

Software infrastructures that allow for composition of simulation models and support external configuration of a particular simulation support the exchange of both simulation models and experiment descriptions between modelers. This is because simulation models are to be implemented as separate software components that can be transferred physically between simulators.

However, the initial preparation of such infrastructures demands an effort which dilutes later advantages and thus might as well be avoided in favor of quickly obtained results for the particular problem at hand. This is especially reasonable in the case of Green's method since it is a mere convolution which consists of only a few loop-statements in source code. This might explain the fact that so far no

such infrastructure seems to exist for Green's method although its inherent modular structure highly favors composability.

The situation the author focuses is different in the way that the objective from the very beginning is to implement two different Green's functions. Both are to be convolved with identical mass forces since a comparison of simulation results is aimed at. Thus, the perspective of two almost identical source code bases gave rise to the motivation to provide a software infrastructure that implements Green's method in a way that allows for external composition of the convolutions participants (Green's function, load function, and convolution operator).

1.2 Objectives and contribution

The main objective of this thesis is the development of a simulation framework which implements Green's method to study crustal loading effects, especially crustal deformation. Application scenarios of such a framework include the simulation of effects such as surface displacements, strain changes, and gravity changes of the Earth induced by any kind of surface load which could be a water reservoir, a glacier, as well as a lava flow. Comparable studies were recently conducted by, e.g., *Pinel et al.* (2007), *Grapenthin et al.* (2006), *Ófeigsson et al.* (2006), and *Barletta et al.* (2006) to name but a few. The general approach for finding a solution to any of the above scenarios would either be a novel implementation of Green's method or the alteration of existing source code which involves much redundant programming and testing.

To avoid these redundancies the developed simulation framework should support user-driven simulation model composition and simulation parameterization; at best by providing an infrastructure that enables the exchange of simulation models and experiment definitions between the users.

A part of this thesis is a case study that utilizes the developed simulation framework to demonstrate its applicability to questions regarding crustal deformation. The response of the Earth's crust to the lava of the year 2000 eruption of the

Icelandic volcano Mt. Hekla is simulated to achieve this objective.

Primarily the utilization of the Green's functions derived by *Pinel et al.* (2007) is aspired. The thesis uses these functions as published; their accuracy[1] is relied on. An in-depth discussion on the underlying theoretical details it not provided within this thesis; in doubt the original publication and references therein must be consulted.

The priority of the implementation of a composable Green's method is to allow for a steep learning curve[2]. The intention of this is to encourage modelers to implement newly derived formal models based on Green's method in a way compatible with the simulation framework developed here and furthermore their release to the public. Therefore, simulation models must be simple to compose and extensions to the framework should be acceptable in at least the programming language C; FORTRAN should be an additional option since both are widely used in the geosciences. Furthermore, the simulation framework should be easy to install with few or no requirements to its runtime environment.

From the above objectives of the thesis the demands on this work are evident and only briefly summarized in the following. Ranking first are studies on the means to transform phenomena of the real world into an executable and extendable simulation framework, as well as closely related concepts such as composition, reuse, and validation and verification of models. This follows the study of Green's method as used by *Pinel et al.* (2007). The general concept of this mathematical tool must be understood and set into a relation to the examined system, the Earth-load-system. Additionally the underlying theories must be looked at to identify an efficient algorithm for implementation. Finally, the simulation framework and necessary additional elements are to be implemented and applied to conduct the case study.

---

[1] This refers especially to the fact that no proof or validation of the semantics of these functions is given here.

[2] Here, a learning curve is understood following its original meaning which expresses a functional relationship between the duration of learning and the resulting progress (`http://en.wikipedia.org/wiki/Learning_curve`, 23.05.07). Hence, a steep learning curve displays much progress during the initial stages of learning something new.

1.3 Structure of this thesis

After the general introduction to the topic of the thesis in this chapter, **chapter 2** gives an introduction to simulation and modeling science. Concepts that guide the development process of the simulation framework through the remainder of this thesis are presented. The idea of a composable simulation model, as well as the terms verification, validation and testing are investigated in detail. **Chapter 3** introduces the system that is to be examined in depth and explains theoretical backgrounds necessary to transform the observed system into a mathematical formulation. **Chapter 4** forms the bulk of this thesis and describes the architecture and implementation of the simulation framework that is realized within this work. The case study that is conducted in **chapter 5** demonstrates the applicability of the simulation framework as an aid in scientific work regarding crustal deformation studies. The thesis closes with concluding remarks and an outlook in **chapter 6**.

The appendices provide information supplementary to the main text. **Appendix A** serves as a reference to the symbols used in equations in the thesis. **Appendix B** presents and describes the directory structure of the enclosed CD. The installation of the simulation framework and how simulations are to be invoked is described in **appendix C**. A sample experiment and therewith the use cases of the simulation framework are detailed in **appendix D**. The extensive **appendix E** contains implementation details on the interfaces of the simulation framework and on the implemented plug-ins. Furthermore, some instructions on implementation and compilation of new plug-ins are given.

Lists of **figures**, **listings**, and **tables** are given directly after the table of contents to aid the quick look up of the respective contents. All **abbreviations** used in this thesis are explained before chapter 1.

2 Modeling and simulation

"The best material model of a cat is another, or preferably the same, cat."
(Norbert Wiener)

A general introduction to modeling and simulation science is given in this chapter. Terms fundamental to this area and of importance to the general understanding of this thesis are defined in section 2.1. A common classification of simulation model paradigms is given in section 2.2. The subsequent section 2.3 introduces a sequence of techniques which support solving general simulation problems. This sequence is expanded by the author so that specifics of the problem at hand are accounted for. The several stages of this sequence are followed in a step-wise manner in chapters 3 and 4 to achieve the objectives of this work (see section 1.2). Two stages of this sequence, composable simulation models and model validation and verification, deserve closer attention. The concept of composable simulation models is discussed in section 2.4 since this concept proves to be valuable with regard to this works objectives. Thus, the necessary software engineering background and terminology must be introduced. Furthermore, advantages are presented that motivate this concepts application and reflect its value. A brief debate on the meaning of model validation and verification in section 2.5 clarifies how these techniques are to be understood and what they express with regard to a model.

2.1 Disambiguation

Due to its inherently interdisciplinary nature the terminology used in modeling and simulation science varies slightly throughout the literature depending on application, author, and language. For instance, a computer program that represents a simplified view on a part of the reality might be referred to as *model* (e.g., *Pidd* (2002)), *computer simulation* (*Fujimoto*, 2000), or *simulation model* (*Fischer and Ahrens*, 1996). This section introduces definitions of fundamental terms as a basis for a consistent nomenclature within this thesis. The set of definitions given here is complemented subsequently throughout the rest of this thesis.

Following a materialistic concept of the world, *Fischer and Ahrens* (1996) see the reality as a collection of phenomena denoting perceived appearances. At the very beginning of the modeling process such collections are described and structured in a process that abstracts from the single entity. Phenomena which by definition hold specific characteristics are categorized. The emerging groups are entitled umbrella terms. In this object-oriented approach each group is referred to as *model class*.

A *model*[1] in its most common meaning is an abstraction from reality. Ranging from simple drawings to complex mathematical descriptions, models are analogies to phenomena of the objective reality and their interactions. Hence, *modeling* describes the process of obtaining an abstraction from reality. The literature (e.g., *Fischer and Ahrens* (1996)) contains different specializations of this general understanding of models depending on, e.g., application and level of abstraction. A *simulation*[2] *model*, for instance, is an abstraction from a phenomenon specifically designed to be experimented with (other specializations are introduced in section 2.3). The allowance for a systematic change of parameters which supports the acquisition of knowledge about the behavior of a modeled phenomenon is usually a design criterion leading to a simulation model.

A *system* is a phenomenon that *Fischer and Ahrens* (1996) expect to possess three characteristics:

[1] *modellus*, diminutive of *modulus* (Latin): small measure

[2] *simulare*, (Latin): imitate

- It must have a **purpose** an observer can identify.

- The phenomenon is a **composition of** other mutually related phenomena, so-called **system elements**, which define its functionality with regard to the purpose.

- The phenomenon is **non-divisible**: its purpose is altered or unrealizable if a part is extracted from the system.

The computational realization of a simulation model is referred to as *simulator*. In terms of *Zeigler et al.* (2000), "a simulator is any computation system [...] capable of executing a model to generate its behavior." This is not limited to a computer in the technological sense. Due to its ability to perform thought experiments, the human mind is a good example for such a model behavior generating system.

An important principle in software engineering is the 'keep it small and simple' (KISS) principle which propagates a modularized design; meaning that complex software applications are constructed from simpler parts. The *plug-in* concept realized by a software application is one possible implementation of the KISS principle in which software parts[3] of an application can be added, replaced, or removed by the user (*Mayer et al.*, 2003). The plug-in concept can be extended as far as creating a program that obtains all of its functionality from plug-ins and their interaction. Such a program would merely implement a management mechanism to provides an infrastructure for, e.g., the plug-in communication.

A *simulation framework* can be thought of as an implementation of the KISS principle since it provides and connects building blocks of a simulation model. A simulation framework providing a plug-in mechanism is a possible realization of the *composable simulation model* concept which enables user-driven simulation model composition. A detailed description is given in section 2.4.

[3] The meaning of 'software parts' is intentionally this vague at this point. It is sufficient to get a general understanding of the term 'plug-in'. However, a detailed discussion about the nature of these software parts is given in sections 2.4 and 4.1.

2.2 Simulation model paradigms

In modeling exists a number of different approaches to obtain an abstract representation of the original system in the form of a simulation model. These *paradigms* can be classified depending on the characteristics of a simulation model that seem most significant. Possible classifications of simulation models include time treatment (e.g., real-time or static[4]), or the way simulator states change as simulation time[5] advances.

Figure 2.1 shows a classification proposed by *Fujimoto* (2000) with regard to state changes. The most general classification of simulation models is as *continuous* and *discrete*. If the state of a system is considered to change continuously over time, simulation models that utilize differential equations with regard to simulation time describe such systems in *continuous* simulation models. Weather simulation models are a popular example of this paradigm. In *discrete* simulation models the original system is thought of as changing states only at discrete points in simulation time. If the original system can be described this way, the advantage to continuous models are consistent snapshots of the models state variables at each point in simulation time. *Fujimoto* (2000) describes the view on such a system as "jumping from one state to the next, much like moving from one frame to another in a cartoon strip." If these jumps in simulation time are always of equal size, the simulation model is referred to as *time-stepped*. However, systems exist that cannot be realized using a time-stepped simulation model since state changes in simulation time might, for

[4] Static modeling refers to obtaining only an end result, i.e. no time treatment is included.

[5] Simulation time is an abstraction to simulate the time of the original system in the simulation model(*Fujimoto*, 2000).

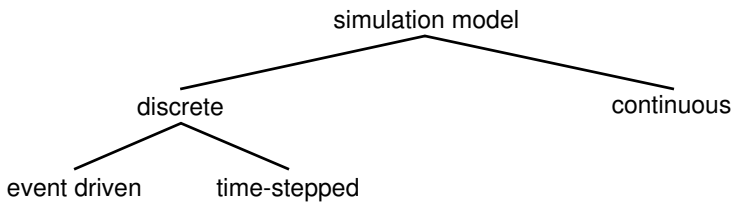


Figure 2.1: Classification of simulation model paradigms (*Fujimoto* (2000), changed)

instance, occur in a randomized way. When such a system is modeled, the smallest necessary time interval between two distinct system states might be unknown which requires the selection of the smallest possible step size to advance in simulation time. In this case time-stepped modeling leads to high frequency re-calculation of values that did not change at all. To avoid this overhead and (re-)calculate states only when something 'interesting' had happened, the concept of *event driven* simulation models evolved. In these models an *event* is referred to as a point in simulation time at which an instantaneous action of the original system is simulated.

The classification described above allows to draw conclusions regarding the method state variables are calculated. Later, this will aid the specification of the simulation framework and give insights into its design. Other classifications, e.g., classification of simulation models with regard to their time treatment, are not beneficial to this work since the key concepts of the system examined here are cannot be expressed in an adequate manner.

2.3 Computer simulation and problem solution

The stages of a computer simulation process which includes model development and implementation are identified by *Fischer and Ahrens* (1996); depicted in the scheme of figure 2.2. The different identified stages will aid the derivation of a composable simulation model in chapters 3 and 4 to achieve the objectives of this thesis.

In the beginning of the modeling process observation of the original system leads to a *conceptual model*. Here, the essence of a systems functionality, i.e. the system elements (model classes) and their relations, is represented in an informal, usually graphical manner. This helps to define the systems boundaries. The analysis objective, i.e. the question that is supposed to be answered with this model, must be formulated at this stage. The next step involves further specification which could be done, for instance, in either a rather declarative or a rather functional way. A *declarative model* depicts either state changes or events within the system. A *functional model*, however, presents the system as a black box and relates a particular

input to the respective output. This stage, though possibly depicting functional relations, is still located on an informal level. Analysis of this informal model and its translation into formulae lead eventually to a *formal model*; a mathematical representation of the system aligned to the analysis objective.

2.4 The composable simulation model

The focus of this work lies with the transition from the formal model to executable software for experimentation. Therefore, the stage of the *composable simulation model* in figure 2.2 is depicted in greater detail than originally intended by *Fischer and Ahrens* (1996).

In chapter 3 it will be shown that the characteristics of the formal model for the system examined within this work allow for decomposition into several simulation models. Figure 2.2 shows that an infrastructure must be provided in the form of

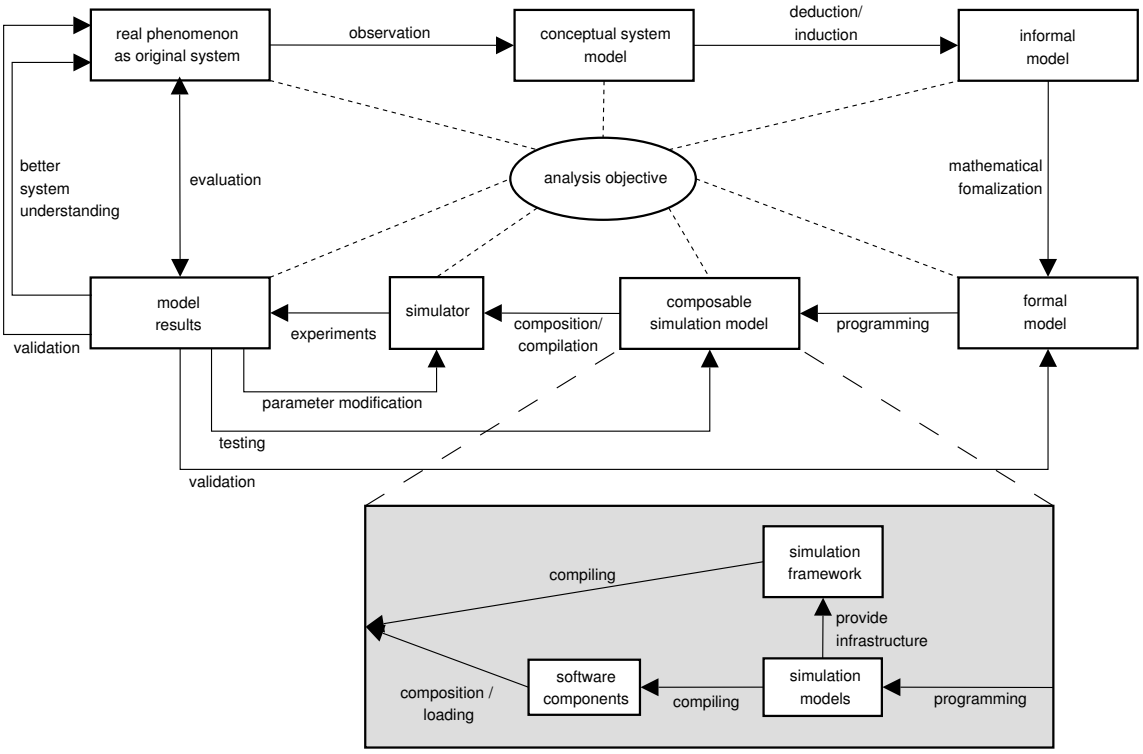


Figure 2.2: Scheme for solving a simulation problem using composable simulation models. See text (sections 2.3 to 2.5) for detailed explanations. (adapted from *Fischer and Ahrens* (1996))

a simulation framework which assures the organization of the simulation models in the way specified by the formal model. However, before the *composable simulation model* can be precisely defined and discussed, the key element to composability – the *software component* – deserves a little more attention.

Böhme (2007) lists three characteristics a system element needs to possess to be considered as *component*:

- Composability: A component is a building block of a system and therewith a part of a whole.
- Dependency on context: The context in which the term 'component' is used defines criteria to distinguish a component from the rest of a system.
- Component purpose: Much like considering a phenomenon as a system requires the phenomenon to allow for identification of its functionality (see section 2.1), a component is a subsystem with a self-contained functionality.

Adding claims about structure and behavior to the above characteristics, *Böhme* (2007) defines *software components* as templates which define an initial state and a potential behavior of software component instances in their physical representation as binary code (e.g., a dynamic (software) library[6]). A *software component instance* is a realization of a software component; it possesses a state and a behavior. *Interfaces* that describe the interaction of the software component instance with the rest of the software system are defined by the software component. Thus, a software component is much like a blueprint in engineering. It describes in detail how things are supposed to work, starting from an initial state. However, it is not able to execute any activity itself which is left to a realization of the blueprint – in engineering a physical workpiece, in software engineering the software component instance. *Böhme* (2007) suggests to avoid context-free usage of the term 'component' due to

[6] In computer science a library is a collection of subroutines. If loaded into an application at runtime, the library that contains the subroutines is called dynamic library; it remains as a separate file on the hard disk. The opposite of a dynamic library is a static library. Its subroutines are physically included (copied) into the application at compile time.

the significant semantic differences depending on the particular domain in which the term is used.

To realize software component instances the simulator needs to provide a special infrastructure which shall be referred to as *simulation framework* and should support the following functionality (*Böhme*, 2007):

- registration of software components (announcing the software components to the simulation framework),
- instantiation of software component instances from registered software components (create a copy of the template in memory which allows for representation of state and behavior of the very instance),
- composition of software component instances (binding of all interfaces), and
- interaction of the software component instances (communication via their interfaces).

Knowing what role software components play and what functionality a simulation framework must support, we can go back to figure 2.2 and detail the introduction to this section.

The formal model defines the context and therewith the criteria to distinguish the subsystems. As depicted in figure 2.2 the subsystems that are identified in the formal model are implemented as simulation models and then compiled into software components. The infrastructure that must be provided and thus the design of a particular simulation framework depends on the each of the simulation models, as well as their interaction as specified by the formal model. The binary representation of the simulation framework on the simulator (i.e., the programmed computer used to run a simulation) is responsible to load the software components. Furthermore, it must compose the created software component instances in a way that their interaction emulates the functionality of the formal model.

From this follows the definition for the composable simulation model:

Definition 1. *A* **composable simulation model** *is an abstraction from a phenomenom which can be divided into subsystems that are implemented as simulation models. It provides an infrastructure that allows for registration of software components (the binary representation of the simulation models), as well as instantiation, composition, and interaction of software component instances which represent the phenomenons subsystems in software. The composable simulation model is specifically designed to be experimented with. Thus, the infrastructure must provide further functionality for parameterization of the software component instances.*

The advantage gained from the efforts that are apparently necessary to realize a composable simulation model is software *reuse* on an advanced level. According to a definition by *Pidd* (2002), software reuse includes the "isolation, selection, and utilization of existing software parts" in the development of new software. Figure 2.3 shows an incomplete spectrum of reuse types which affect diverse kinds of software parts. They can be applied to reduce efforts that go into implementation and testing when software is written from scratch.

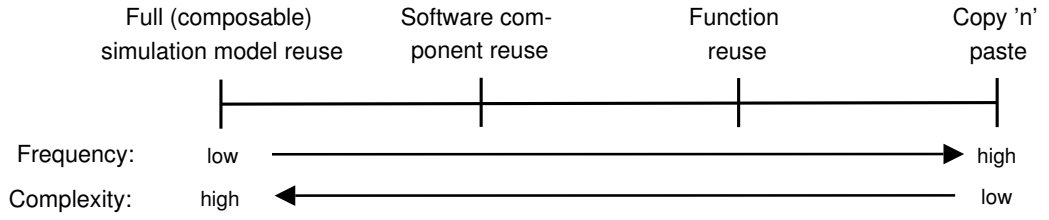


Figure 2.3: A (incomplete) software reuse spectrum. Since object-oriented concepts are usually not familiar in areas outside computer science concepts like class reuse, class collection reuse, and others which might, e.g., be located between function and software component reuse, are not included in this spectrum. (*Pidd* (2002), changed)

The overall message of figure 2.3 is that reuse frequency is inversely correlated to software complexity. Furthermore, it is implied that with complexity the level of abstraction raises. Source code fragments are certainly less abstract than concepts for composable simulation models as suggested by the definition above. Possible types of software reuse are described as follows (incomplete):

- *Copy 'n' paste* is the most common technique. Relatively small segments of code, usually slightly modified, are reused. Trust in the original programmer might be necessary, but is not crucial. The code is of a size manageable for reading and testing before reuse. Such segments often provide a good code basis for new projects.

- *Function reuse* includes, e.g., built-in functions of a programming language. Usually the purpose of the functions is quite specific which enables for simple tests to build up trust.

- *Software component reuse* describes the utilization of software components that offer functionality through interfaces to, e.g., third-party software. Both, complexity and necessary testing depend on the offered functionality, e.g. number of interfaces.

- *Full (composable) simulation model reuse* in the sense of using a complete (composable) simulation model as part of another, more complex (composable) simulation model has been the driving force for many developments in the simulation world (e.g. HLA[7]). However, problems to realize this type of reuse are manifold (e.g., how to deal with the fact that a model is designed having one specific objective in mind) (*Paul and Taylor*, 2002; *Overstreet et al.*, 2002).

Types of reuse not included in the spectrum are mainly from the field of object-oriented modeling and design and include concepts like class reuse and class collection reuse which might be located between function and software component reuse in figure 2.3. However, object-oriented concepts are usually not familiar to users of areas outside of computer science. Following one of the objectives of this work, independent extention of the proposed simulation framework shall be encouraged. This cannot be achieved if the world of object-oriented programming must be unlocked first. For this reason the presented spectrum of possible reuse scenarios spans concepts that are familiar to other disciplines or at least easy to grasp.

[7] High Level Architecture: for distributed simulations; desires to support universal and uniform interoperability (communication) of simulation models (e.g., *Pidd* (2002)).

The prospect of software component reuse or even full simulation model reuse made composable simulation models a vividly researched area in simulation and modeling science. Apart from the definition given here (see definition 1), several definitions for this term exist. They differ mainly in terms of the flexibility software components must provide to make up a composable simulation model. *Kasputis and Ng* (2000), for instance, propose in their position paper "... a [software] system with which simulations are created at runtime to meet the specific requirements of that run. The user specifies his needs to a system that in real time builds a simulation [model] ..." *Page and Opper* (1999), however, show that deciding whether an arbitrary collection of software components meets specified model requirements is NP-complete[8]. A more detailed discussion about issues regarding such a proposition is given by *Overstreet et al.* (2002). Apart from "automated solutions being computational intractable", the authors mention challenges in a reuse-enabling-design of composable simulation models and difficulties with the decision about whether existing models satisfy certain objectives. *Overstreet et al.* (2002) identify the capturing of objectives, assumptions, and constraints of simulation models as a key to their reuse.

Another definition for composable simulation models that being more software developer oriented is given by *Winnel and Ladbrook* (2003):

> "[A] Composable Simulation [Model] involves the selection of a series of existing modeling constructs, bringing them together in such a way as to model the real world situation at hand, in much the same way as existing modeling methodologies – but with much of the underlying coding already carried out."

Though closer to being realizable and pointing into the direction of concepts included in the comprehension of a composable simulation model, *Winnel and Ladbrook* (2003) still remain quite vague in their definition. Nothing is said about the

[8] NP-complete problems are the most difficult problems in complexity theory; they are to be computed in non-deterministic polynomial time (NP) (`http://en.wikipedia.org/wiki/NP_complete`, 19.07.07)

nature of the modeling constructs which, as shown in figure 2.3, could be just about anything. Furthermore, it remains unclear how the 'constructs' might be brought together and which parts of the software are affected by the coding that is already carried out. The definition of this thesis (see definition 1) is clear about these concerns and thus reflects the comprehension of composable simulation models in this work.

Although quite vague in their definition, *Winnel and Ladbrook* (2003) give a good overview of the benefits gained from the use of composable simulation models. They list, for instance, time saving due to lower system complexity and therewith money benefits, and quality benefits which come with the reuse of software components that are already tested and validated. *Overstreet et al.* (2002), however, imply that building a (composable) simulation model entirely from tested, validated, and accredited software components does not say much about the validity[9] of this newly composed simulation model. Thus, a certain effort must be made to validate this new simulation model.

Adding to the discussion that fully automated composition might be possible at best in very restricted domains with certain design criteria in mind, *Overstreet et al.* (2002) propose that providing tools for modelers' assistance is a more realistic goal. A calling that is taken up as an objective of this thesis in chapter 4 where an architecture enabling for composition of simulation models on a developer and modeler level is introduced.

2.5 Verification, validation, and testing

Once the composable simulation model is programmed it can be transformed into an executable program. This program and the computer it is installed on are the simulator which now can be used to obtain model results by performing experiments[10] However, figure 2.2 suggests this is not the final stage in the process of solving a

[9] see section 2.5 for a discussion about validity

[10] An experiment is understood as an instance of a (composable) simulation model using a specified set of parameter values.

simulation problem. Yet, nothing indicates that the model results can be trusted at all. Even if results 'appear' to be correct, this might be due to errors that cancel each other out depending on the test conditions.

To describe how much a model can be trusted two terms are widely used in modeling and simulation science:

- verification[11], and
- validation[12].

Both are applied in greatly varying situations; often erroneous (*Oreskes et al.*, 1994). Thus, an effort must be made to clarify both their meaning and their area of application. *Oreskes et al.* (1994) provide an insightful analysis of the philosophical basis of the two terms.

Following the argumentation of *Oreskes et al.* (1994), *verification* refers to an establishment of truth which makes it inapplicable to any model of natural (or other open[13]) systems. The problem starts with the observed data which are "inference-laden signifiers of natural phenomena to which we have incomplete access." (*Oreskes et al.*, 1994). Thus, even data to which model results are compared are not necessarily a 'truth' with respect to the analysis objective. Furthermore, models of natural systems are non-unique. Several models that abstract differently from reality might fit the same observed data which is a contradiction to the term 'verification'. The term 'software verification', however, refers to a discipline in software engineering which is concerned with the assurance that a software satisfies all defined requirements. Certainly this must be done as soon as the simulator produces model results to assure the results conform to the formal model. Yet, to avoid any terminological confusion this process is referred to as *testing* (see figure 2.2).

As for *validation*, *Oreskes et al.* (1994) state that this term is often misused

[11] from *verus* (Latin): true

[12] from *validus* (Latin): strong, effective; later: 'supported by facts or authority'

[13] In an open system the system elements can be influenced by elements outside of the examined system. Usually these influences are included in the model as assumptions which might not hold for any new study (*Oreskes et al.*, 1994).

interchangeably with 'verification' or even in the sense of a valid model being an accurate representation of reality. The literal meaning, however, refers to the establishment of legitimacy. *Oreskes et al.* (1994) translate this to a model being valid if it "does not contain known or detectable flaws and is internally consistent." Over the years more fine-grained concepts of 'validity' emerged in simulation and modeling science to allow for several 'stages' of confidence in the reliability of model results. Concepts of model validity include (weakest to strongest, stronger validity implies the weaker) (*Fischer and Ahrens*, 1996; *Zeigler et al.*, 2000):

- *Application validity*, proves that the model satisfies the analysis objective.
- *Empirical validity*, exists if, for all feasible experiments, the behavior of model and system agree within acceptable tolerance.
- *Behavior validity*, is gained if the model can predict yet unseen system behavior. The model is set to an initial state corresponding to the systems state.
- *Structural validity*, is affirmed if the model mimics in a step-by-step, element-by-element fashion the systems transitions.

To decide about a models validity the results should be evaluated by comparing them to the behavior of the real system. However, if no data from the original system are available, the results of several simulator experiments are to be compared among one another, or, as done in section 4.5, to results obtained from reference implementations. Furthermore, simple analytical solutions, i.e. results calculated directly by using mathematical methods (*Fischer and Ahrens*, 1996) could be used to show model correctness. At each step it might be necessary to go back as far as the observational stage in the modeling process to improve the model.

2.6 Summary

A solid foundation in both terminology and modeling concepts for this thesis was laid. Depending on the original authors, some differences in use and application of

terms exist in this field. An effort was made to agree on a terminology consistent within this thesis. Introduced modeling paradigms are used in later chapters to aid positioning the implemented simulation framework in the modeling world.

A sequence of technologies that applies to the process of solving an arbitrary simulation problem defines a road map through the following chapters. A refinement of that sequence allowed for the inclusion of composable simulation models which is the key concept to this work. The given definition for the term 'composable simulation model' outlines the efforts that must be made in the following chapters to realize this concept. The discussion about reuse strategies showed how the extra effort that goes into providing an infrastructure for the use of software components can ease future extentions of the (composable) simulation model. Finally, it is concluded that the models of concern to this work –models of natural systems– cannot be verified at all. To gain confidence in both, the software implementation and the models, thorough testing and a strong validity relation must be exerted.

3 Examining the deformation of the Earth's crust

"By relating the observed flexure or bending of the lithosphere to known surface loads, we can deduce the elastic properties and thicknesses of plates."
(Turcotte & Schubert)

Now that chapter 2 provided a setting in which arbitrary simulation problems can be solved, this chapter can introduce the system that is to be modeled and simulated. Theories necessary to solve the modeling problem at hand are presented. The sections are organized in a way that follows the steps suggested in figure 2.2 up to the point of deriving a formal load model. Additional background information is provided along the way to allow for comprehensible transformation between the different stages.

At first the physical problem and the analysis objectives are presented accompanied by the conceptual model in section 3.1. A general introduction to the Earth's structure and parts of its behavior is given in section 3.2. This gives a 'feeling' about the complexities and problems that may arise when it comes to simulate the behavior of this key part of the modeled system. In order to proceed applying the techniques outlined in figure 2.2 the author changes the perspective on the system to that of signal processing. This discipline provides several useful tools that are applied to derive an informal model for the original system in section 3.3. Section 3.4 is concerned with the introduction of the theoretical background essential for the formulation of two formal models that express the surface displacements of

the Earth due to loading which is done in section 3.5. The formal models that are derived in section 3.5 are formulae that generally can be implemented in two ways since they are based on the convolution operation. In section 3.6 a technique is presented that allows an efficient implementation of the formal models. In order to apply this technique correctly in the following chapter, its theoretical background is given and prerequisites that must be fulfilled are presented. This technique will serve as a blueprint to the implementation of the composable simulation model in the following chapter.

3.1 System description & conceptual model

The system examined in this thesis is concerned with the deformation of the Earth's surface in response to changes of the superimposed load and thus is referred to as *Earth-load-system*. Examples for load variations are changes in atmospheric pressure, melting of a glacier, emplacement of a lava flow, or various combinations of different mass forces that act on the Earth's surface. Mostly untraceable for the human eye, a means to measure the Earth's surface deformation is the Global Positioning System (GPS). Generally speaking, receivers on the ground calculate the changes of a points position by relating it to known positions of satellites.

Three system elements make up the Earth-load-system:

- a *load* is a body that applies a mass force to the surface it rests on,
- the *Earth* is a body which shows deformation at its surface in response to changes in the applied mass force, and
- a *load history* expresses the changes of a mass force over time, e.g., alternation between extrema.

Considering the definition of a *system* as given in section 2.1, it is obvious that extracting one of the above system elements from the Earth-load-system would make

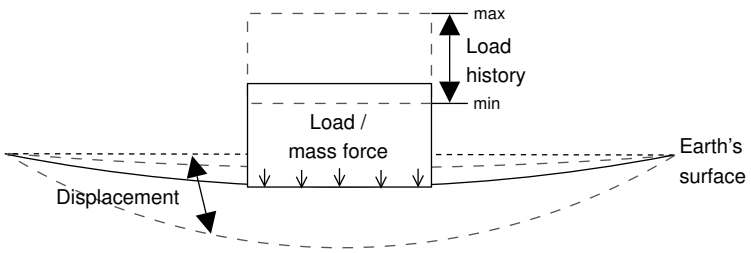


Figure 3.1: Conceptual model of the Earth-load-system. The load (solid black rectangle) is applied to the pointed black line which represents the initial state of the Earth's surface. The Earth responds to the mass force (little black arrows) with surface displacement, changing its state to the one denoted by the solid black line. The load, however, may vary over time which is denoted by a load history. At first it might raise to a maximum level which results in maximum displacement and then it could drop to a minimum represented by the dashed gray lines that limit or extend the load box. Depending on the load history, the displacement might alternate between the upper and the lower dashed, gray surface lines, linked to minimum and maximum load, respectively.

its purpose – to show a surface displacement due to a load change – unrealizable[1]. Figure 3.1 shows a possible conceptual model of the system.

The main objective when examining an Earth-load-system as depicted in figure 3.1 is the determination of the effects the mass force has on the body it is applied to. Such loading effects are not limited to surface displacements which this thesis focuses on. Examination of gravity variations or geocenter displacements (*van Dam et al.*, 2002) might also be of interest; all of which are referred to as *loading problems*. The **analysis objective** of the Earth-load-system is to find a solution for one particular loading problem; either as quantitative estimation of anticipated effects to a load change, or –when fit to real observations, e.g., GPS data– to infer knowledge about a system element (known as *inverse problem*). An inverse problem with regard to the Earth-load-system could be the derivation of deeper knowledge about the Earth's structure by fitting modeled surface displacements to those observed at GPS-stations (e.g., *Grapenthin et al.* (2006)).

When modeling the Earth-load-system it can be expressed as an aggregation of models for the following system elements:

- Earth model: a mathematical model that expresses geometrical considerations, and mechanical properties of the Earth,

[1] In this work static models are assumed to have a load history of step size 1 with the initial minimum load being zero. At step 1 the maximum load is applied to the Earth's surface. Thus, the load history cannot be extracted from the system.

- load model: either a mathematical model describing the extention and composition of a load, or data from load observations, and
- load history model: either a mathematical model describing the temporal evolution of a load or times series from observations (real, probably processed data).

Since a multitude of abstractions exists to express any of these elements in formulae, each is considered a *model class*. Although the Earth is spherical, mathematical expressions which describe it as a flat Earth approximation are considerably easier to handle. Thus, when modeling small segments of the Earth its spherical nature could be neglected. This results in simpler mathematical formulations which, as a downside, might add to the uncertainties of the model results. The load model class could contain data points describing the real extension of a load, or geometrical approximations of the load geometry such as disks, lines, or points. Each of these approximations is considered an instance of an Earth or load model, respectively.

Modeling the Earth-load-system as an aggregation of model classes provides the basis for composition of model class instances on the implementation level (in the form of simulation models) as shown in figure 2.2 and detailed in section 2.4.

The remainder of this chapter focuses on the derivation of two exemplary Earth models as well as the formal method that connects them to load models. Since the scope of this thesis is mainly to provide a simulation framework that makes any of the models that form the Earth-load-system exchangeable, the complexity of exemplary load and load history models is kept to a minimum. Explicit models for these are derived in chapter 4 and appendix E.2.

3.2 The Earth's inner structure

Before suitable models for the Earth can be introduced, an overview of the complexity of the Earth's inner structure should be given, because it accounts for the multitude of models that can be derived to approximate the Earth.

The Earth is basically composed of three distinct shells. The core with a radius of about $3500\,km$ consists of a solid inner and a liquid outer core and is enclosed by the mantle. The outermost shell is the thin crust; of all shells the least dense. Figure 3.2 shows estimated thickness and location of each shell, and illustrates the subdivision of the mantle.

The uppermost part of the mantle begins at a depth that depends on the thickness of the overlying crust. In areas of ocean basins this is at about $0 - 10\,km$ depth. Whereas, beneath huge mountain ranges the crust gains thicknesses of up to $80\,km$, e.g., in the area of the Andes or the Himalayas. The transition from crust to mantle marks a significant discontinuity in travel time of seismic waves[2]. The crust and the upper part of the mantle form the *lithosphere*[3]. Rocks of the lithosphere behave rigidly and are sufficiently cool to not deform on time scales less than 10^9 years (*Turcotte and Schubert*, 2002). The $1600\,K$ isotherm[4] marks the transition between lithosphere and asthenosphere[5]. The depth of this isotherm ranges from $100\,km$ depth beneath the oceans to about $200\,km$ depth beneath continental plates. Rocks of the asthenosphere are sufficiently hot to deform on short time scales. Beneath $370\,km$ depth lies a transition zone in which density steadily increases until at about $720\,km$ depth the even denser inner mantle begins.

To studies of crustal deformation the behavior of the lithosphere which is fragmented in tectonic plates is of major interest. Due to its rigidity the lithosphere bends when a load is applied to its surface. Examples for such loads are, e.g., atmosphere, glaciers, or volcanoes. Around the Hawaiian Islands, for instance, the bending of the lithosphere due to the Islands' load results in a region of greater water depth than farther from the islands (*Watts*, 2001).

The upper half of the lithosphere does not return to its initial state on time scales of 10^9 years. This part is identified as the elastic lithosphere. The lower and

[2] The so-called Moho or Mohorovičić-discontinuity

[3] *lithos* (Greek): stone + *sphere* (Greek): globe, ball

[4] surface of constant temperature given in Kelvin [K]

[5] *a* + *sthenos* (Greek): without strength, rigidity

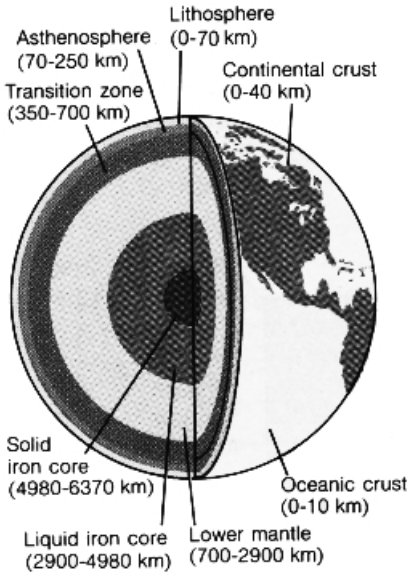


Figure 3.2: The Earth's inner structure as understood today (*Press and Siever*, 1978)

hotter part, referred to as the thermal lithosphere, bends when a load is applied, but relaxes on significantly shorter time scales than the elastic lithosphere (*Turcotte and Schubert*, 2002).

Depending on genesis and scale, the structure of the lithosphere can be of great complexity. Figure 3.3(a) shows a schematic section through oceanic crust. It is made of four distinct layers each having different physical properties. The tectonics of south-eastern Germany illustrated in Figure 3.3(b) give an example of the complexity lithospheric structure gains after several cycles of orogeny[6] and leveling over the course of about 570 million years. These examples are supposed to underline the necessity of a certain degree of abstraction in modeling crustal deformation over broader regions. It is simply impossible and unnecessary to capture the exact geology of a region in a model to conceive the processes that account for particular observations.

[6] *oro* (Greek): mountain + *genesis* (Greek): origin, creation, generation

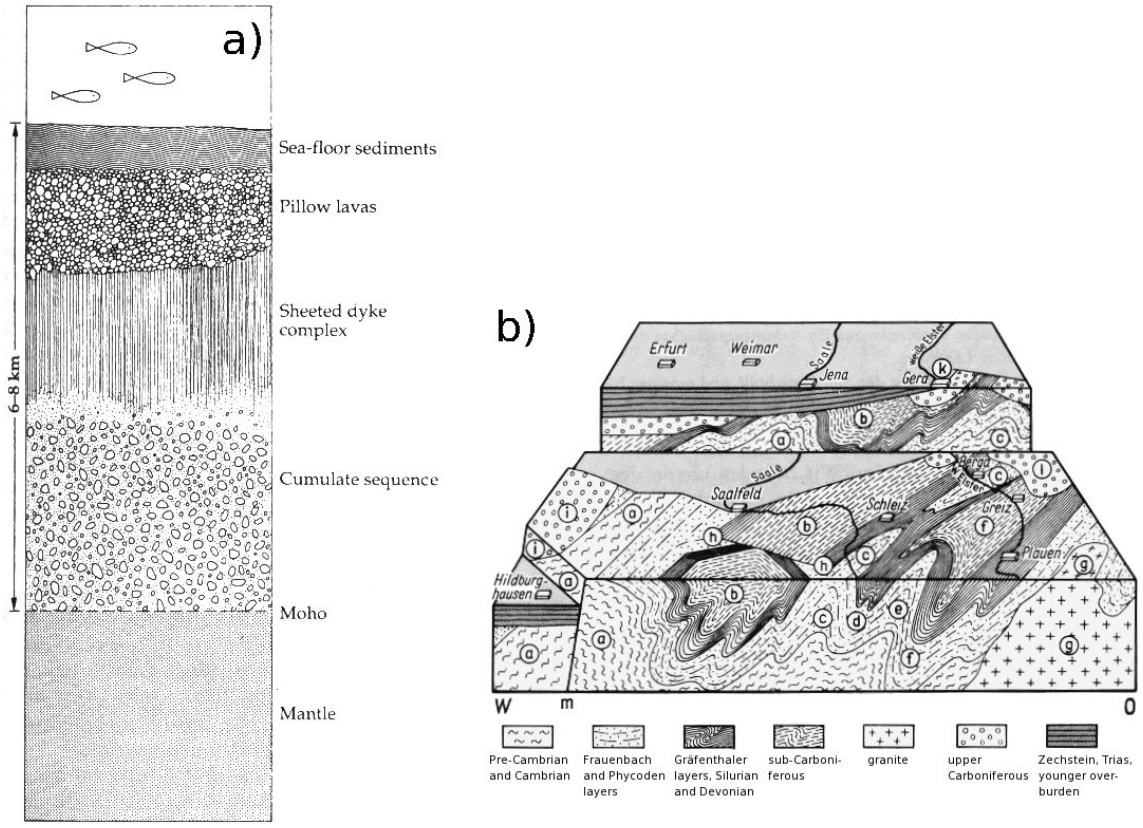


Figure 3.3: Schematic sections through the crust at two distinct parts of the Earth. Both depict about 6-8 km of depth. **(a)** Section through typical oceanic crust which consists of a good number of horizontal layers with different physical properties. (*Francis and Oppenheimer*, 2004) **(b)** Tectonics of south-eastern Germany showing folds which result from orogenesis. The result is breakup of the homogeneous horizontal layers that now even have vertical boundaries with layers formerly resting on top or below. Thus, the physical properties of such structures do not only depend on the sequence of the layers and their properties but also in the way they are folded. The circular labels mark distinct topographic features that are of interest in the original publication. (*Wagenbreth and Steiner*, 1990)

3.3 The lithosphere from a signal processing point of view

Section 3.2 gave an impression about the complexity of the lithosphere which implies that a variety of informal models can evolve depending on abstraction level and perspective on the lithosphere. To transform any of the potential informal models into mathematical expressions assistance of analogies to well-studied systems which are backed up by a sound theory is useful. An analogy for the lithosphere developed by, e.g., *Watts* (2001) is that of a *filter* (see figure 3.4). This analogy enables the

application of tools of filter theory to loading problems. However, before this analogy can be explained a general introduction to filters must be given.

A filter in its most common meaning, i.e. an element linking an input to an output (*Watts*, 2001), is shown in the block diagram of figure 3.4. Amplitude and phase of an input signal are weighted by a filter depending on the position in the spectrum before they are read out. The term *spectrum*[7] refers to a representation of a signal which depends on the frequencies of the signals constituents (e.g., *Meffert and Hochmuth* (2004)). *Meffert and Hochmuth* (2004) define a *signal* as a space-time-object being tied to a physical carrier whose parameter variations in time and space may contain information. For instance, a glaciers mass over a specific area with the parameters volume and density can be referred to as a signal.

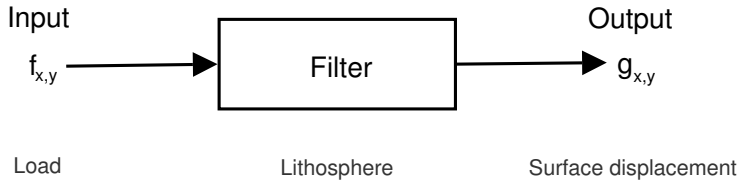


Figure 3.4: Block diagram of a filter with $f_{x,y}, g_{x,y}$ denoting discrete two-dimensional signals (value sequences) for input and output, respectively. The input is linked to an output depending on the characteristics of the filter. The most simple form of a filter is the all-pass which passes the input unchanged (e.g., a mere wire). Other filters pass only parts of the inputs frequency content. Assuming the lithosphere is a filter, its input is a load and its response is surface displacement on the output side occurs (displacement of the lithosphere, i.e. the filter, itself). The exact displacement is determined by the characteristics of the lithosphere. The simulated displacements depend on the chosen informal model for the lithosphere. In general, the lithosphere shows the characteristics of a low-pass filter. This means the output signal is smoother than the input (see text for details) (*Watts* (2001), changed).

Among other properties, filters can be distinguished by their spectral characteristics. A low-pass filter passes frequencies below a specific frequency, whereas higher frequencies are eliminated. The output signal, also called *response (function)*, appears smoother for it contains fewer inflection points since high frequency parts are eliminated. Other filter types with analog definitions are all-pass, high-pass, band-pass, and band-elimination filters.

Coming back to the analogy for the lithosphere, *Watts* (2001) states it can be attributed as filter, because a load is linked to a surface displacement depending on

[7] "apparition, specter", *spectrum*, (Latin): appearance, image, apparition

the characteristics of the lithosphere. With displacements extending into areas not covered by the load, it can be understood that short wavelength displacements are suppressed, e.g., wavelengths smaller than the dimensions of the load. However, long wavelength displacements associated with flexure are passed. According to *Watts* (2001) this behavior is related to the strength of the lithosphere. By responding with displacements as described here, the lithosphere behaves like a low-pass filter.

A special kind of filter is a *linear space-invariant* (LSI) filter. If the lithosphere responds to loads h_1, h_2 with surface displacements f_1, f_2, respectively, and a load $a\,h_1 + b\,h_2$ produces a surface displacement $a\,f_1 + b\,f_2$, the lithosphere is a *linear* filter. The arbitrary constants a and b are mere amplification factors. *Space-invariance* entitles a filter whose response does not depend on the inputs position. If a load h_1 at point (x, y) causes surface displacement f_1 at (x, y), then f_1 occurs at point $(x + r, y + s)$ when h_1 is shifted to $(x + r, y + s)$ for any values r and s. An approach to limit the complexity of crustal deformation studies is to assume the lithosphere is a LSI-filter (e.g., *Watts* (2001)). Its behavior is well known and includes (e.g., *Meffert and Hochmuth* (2004)):

- The principle of superposition is true (this comes along with linearity).
- The output contains only frequencies that are already existent in the input.
- A harmonic input has a harmonic output of the same frequency.

The following section introduces the mathematical background necessary to convert an informal model for the lithosphere to a formal model. The particular informal models for the lithosphere that are used as a basis in this thesis and their respective mathematical representations are presented in section 3.5.

3.4 Green's functions as a load response function

In his 1828 *Essay on the Application of Mathematical Analysis to the Theories of Electricity and Magnetism* George Green, the English mathematician and physicist, derived a powerful tool which is now called a *Green's function*. In general, such

a function represents a particular solution to an inhomogeneous partial differential equation with boundary conditions. With the background on linear filters given in section 3.3, Green's functions can also be described as a linear filters response to a delta function which can be imagined as a force acting on a very small entity, i.e. a short time (unit impulse), or a point (unit point mass) (see section 3.5). Since the lithosphere is assumed to be a linear filter (see section 3.3) Green's functions can be applied to solve, e.g., surface displacement problems due to loading.

Challis and Sheard (2003) describe the concept of a Green's function based on an unit impulse acting on a particle and its displacement. Adapted to loading problems of the lithosphere, the dynamics of a point $\vec{r}$ (cylindrical coordinates) on the Earth's surface that responds to an unit point mass with a displacement are considered (see figure 3.5). Applying an unit point mass $L(\vec{r'})$ to point $\vec{r'}$ results in a displacement $U(\vec{r'})$ at $\vec{r'}$. The displacement at $\vec{r}$ is determined by the Green's function $G(\vec{r}, \vec{r'})$. A real load L, however, covers an area instead of a point. It can be represented as a grid of unit point masses acting at (ideally) equidistant points on the lithosphere. Integration over the load covered area R subsumes the effects each unit point mass has on point $\vec{r}$:

$$U(\vec{r}) = \int_R G(\vec{r} - \vec{r'}) L(\vec{r'}) dS \tag{3.1}$$

Having in mind that unit point masses can be zero in R, equation 3.1 implies that geometries of loads are not restricted to any shape. The approach of equation 3.1 is referred to as *Green's method* (e.g., *Thornton and Marion* (2003)).

As *Challis and Sheard* (2003) point out: "a Green function depends on the dynamical system but not on the form of the applied force." In the case of loading problems the applied force is a loads mass. The dynamical system, however, is the lithosphere for which several conceptual models exist (see section 3.2 and section 3.5). Deducing these conceptual models to LSI-filters results in a number of informal models for the lithosphere. To describe a particular model in mathematical terms, a Green's function must be derived for each one of those. Detailed descriptions on how a particular Green's function is derived are given by, e.g., *Snieder* (2004).

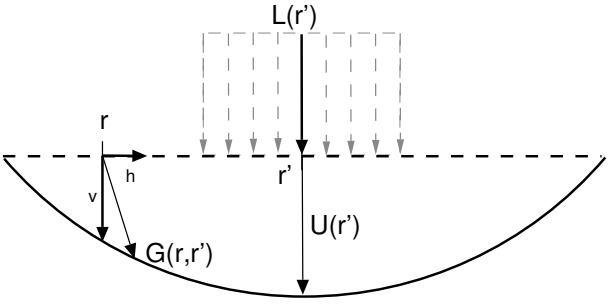


Figure 3.5: Green's function for the response to an unit point mass. The displacement in point r due to a load $L(r')$ that is applied at r' is expressed by the Green's function G(r,r'). Since real data, e.g. from GPS stations, comes with displacements split into spatial directions, instead of calculating the total displacement it might be useful to find separate Green's functions for horizontal h and vertical v displacements. This is furthermore necessary since Green's functions are solutions to partial differential equations. The gray arrows represent a load model that is an array composed of many unit point masses, each having an impact on r. Utilizing Green's method (equation 3.1) and thus convolving the load with a Green's function, the impact of all unit point masses that compose the load is attributed to the displacement at r.

3.5 Applying Green's method: formal models for surface displacement

Green's method (equation 3.1) provides a general frame to calculate the response of a dynamical system to an applied force. To apply this method to the calculation of surface displacements the integrands of equation 3.1 must be defined accordingly. The case is simple for the unit point mass as it is the product of the loads density, ρ, and and its height, h, at a point $\vec{r'}$:

$$L(\vec{r'}) = \rho(\vec{r'})h(\vec{r'}) \tag{3.2}$$

As implied in section 3.4 the respective Green's functions for the lithosphere are more complex to derive which is therefore not done here. In the following two candidate formal models for end-member cases of the lithospheric response to a load as derived by *Pinel et al.* (2007) are presented:

- the *instantaneous response*, the lithosphere is assumed to be an elastic half-space, and
- the *final relaxed response*, the lithosphere is assumed to be a thick elastic plate lying over an inviscid (see section 3.5.2) fluid.

Both informal models neglect the spherical nature of the Earth. Thus, caution is demanded when interpreting results in great distance from the load. When comparing a flat Earth approximation to a more realistic, spherical model of the Earth, *Grapenthin et al.* (2006) found that results agree quite good within a radius of 150 km around the loads center. The vertical instantaneous response of the flat Earth model agrees to within 1% in amplitude to the result obtained from the spherical model. For the instantaneous horizontal displacement, however, a difference in results of up to 10% is obtained.

The reference state before loading is the lithostatic stress field, i.e. pressure comes from the weight of the overlying rock. As defined by *Pinel et al.* (2007) the vertical axis (z) is always directed downwards. Hence, loading induces positive vertical displacement. Negative vertical displacement, however, is related to unloading.

3.5.1 Elastic half-space

A (three dimensional) *half-space* is limited in one spatial direction and ranges from positive to negative infinity in the other two directions, i.e.: $z \geq 0$, $-\infty < x < \infty$, and $-\infty < y < \infty$. *Elastic material* deforms when a force, e.g. a load, is applied. Once this force is removed the material returns to its initial state. To a certain extend all rocks behave elastically at low temperatures as long as the applied forces are not too large. In that case rock might deform plastically (i.e., not return to its initial state) or fracture.

To calculate the instantaneous or initial response of the lithosphere *Pinel et al.* (2007) use an informal model in which the lithosphere is a linear and space-invariant elastic half-space and derive Green's functions for vertical, $G_v(r)$, and horizontal displacement, $G_h(r)$:

$$G_v(r) = \frac{g}{\pi}\frac{(1-\nu^2)}{E}\frac{1}{r} \tag{3.3}$$

$$G_h(r) = -\frac{g}{2\pi}\frac{(1+\nu)(1-2\nu)}{E}\frac{1}{r} \tag{3.4}$$

Elastic parameters characterizing the lithosphere are the *Poisson's ratio*, ν, and

the *effective Young's modulus*, E. The former is a constant determining the compressibility of a material. It ranges from 0.0 to 0.5, where 0.5 describes a perfectly incompressible material. A common value for the lithosphere used in the literature is $\nu = 0.25$. The effective Young's modulus describes the force necessary to deform a composite material. Thus, it is a measure of a materials average stiffness. Typical values for the lithosphere range from $10\,GPa$ to $130\,GPa$.

Using Green's method (equation 3.1) with a load as defined in equation 3.2, and equations 3.3 and 3.4 as Green's functions, the surface displacement at point $\vec{r}$ is given by:

$$U_v(\vec{r}) = \int_R G_v(\vec{r} - \vec{r'})\rho(\vec{r'})h(\vec{r'})dS\vec{r'} \tag{3.5}$$

$$U_h(\vec{r}) = \int_R G_h(\vec{r} - \vec{r'})\rho(\vec{r'})h(\vec{r'})d\vec{S}\vec{r'} \tag{3.6}$$

where $U_v(\vec{r})$ and $U_h(\vec{r})$ are, respectively, vertical and horizontal displacement, $dS(\vec{r'})$ is the area around the point $\vec{r'}$, and R is the area of the load. Vectorial integration, i.e. the vectorial sum of the integrals in the two horizontal directions, is represented by $d\vec{S}\vec{r'}$. Arbitrarily shaped loads within R are created (e.g.) by setting $h = 0$ at points $\vec{r'}$ that do not belong to a load.

3.5.2 Thick plate over an inviscid fluid

To model the fully relaxed response (or final state) of the lithosphere after a load is applied *Pinel et al.* (2007) use an elastic layer of arbitrary thickness H which lies over an inviscid fluid of density ρ_f. *Inviscid* describes a material whose viscosity is neglected. Hence, fluid friction has no effect on the results. The fluid is assumed to be in equilibrium. The method is described in depth by *Pinel et al.* (2007). Therefore, only the Green's functions are given here. Vertical, $G_v^H(r)$, and horizontal, $G_h^H(r)$, displacements at the surface ($z = 0$) are expressed by:

$$G_v^H(r) = G_v(r) + \frac{(1-\nu^2)g}{E\pi}\int_0^\infty (\frac{B}{D} - 1)J_0(\epsilon r)d\epsilon \quad (3.7)$$

$$G_h^H(r) = G_h(r) + \frac{g}{2\pi E(1-2\nu)}\int_0^\infty \left[(1-\nu^2)(2\epsilon\frac{A}{D} + 4\nu)\right] J_1(\epsilon r)d\epsilon \quad (3.8)$$

where J_0 and J_1 are Bessel functions of the first kind of zeroth- and first-order, respectively. The coefficients A, B, and D are defined by:

$$A = \epsilon^2 H^2 - \nu(\cosh(2\epsilon H) - 1) - \frac{2(1-\nu^2)}{E} g\rho_f(\nu\frac{\sinh(2\epsilon H)}{\epsilon} + H) \quad (3.9)$$

$$B = \frac{1}{2}\epsilon\sinh(2\epsilon H) + \epsilon^2 H^2 + \frac{1-\nu^2}{E} g\rho_f(\cosh(2\epsilon H) - 1) \quad (3.10)$$

$$D = -\epsilon^3 H^2 + \frac{1}{2}\epsilon(\cosh(2\epsilon H) - 1) + \frac{1-\nu^2}{E} g\rho_f(\sinh(2\epsilon H) + 2\epsilon H) \quad (3.11)$$

where ϵ is the variable of integration. If $H \to \infty$, then $\frac{A}{D} \to -\frac{2\nu}{\epsilon}$ and $\frac{A}{D} \to 1$ which gives the solution of the elastic half-space of equations 3.3 and 3.4.

The displacements due to loading (see equation 3.2) are given by:

$$U_v^H(\vec{r}) = \int_R G_v^H(\vec{r} - \vec{r'})\rho(\vec{r'})h(\vec{r'})dS\vec{r'} \quad (3.12)$$

$$U_h^H(\vec{r}) = \int_R G_h^H(\vec{r} - \vec{r'})\rho(\vec{r'})h(\vec{r'})d\vec{S}\vec{r'} \quad (3.13)$$

where U_v^H and U_h^H are, respectively, vertical and horizontal displacement depending on plate thickness H.

Linking this section to the guiding figure 2.2, two informal models for the lithosphere, the elastic half-space and the thick plate over an inviscid fluid have been introduced. The analogy between the lithosphere and a filter as described in section 3.3 was utilized to transform the informal models into mathematical expressions for the system introduced in section 3.1. Green's method as introduced in section 3.4 represents the systems formal model. The identified subsystems are expressed by the load function and the Green's function within Green's method.

Mathematical expressions for both are given in this section. These can be utilized to infer the response of the lithosphere to a load.

3.6 Performance enhancement: fast convolution

The operation introduced in equation 3.1 (Green's method) defines the so-called two-dimensional (linear) *convolution*; an important tool for analyzing linear filters. To mark the convolutions character of an operator the double-asterisk notation is used (*Meffert and Hochmuth*, 2004):

$$\begin{aligned} U(\vec{r}) &= \int_R G(\vec{r}-\vec{r'})L(\vec{r'})dS \\ &= G**L \end{aligned} \tag{3.14}$$

A convolution is generally a computational expensive operation with demands in terms of time and memory that increase exponentially depending on signal size which is obvious from equation 3.1. This section will introduce a technique that is called *fast convolution* which under certain circumstances significantly decreases the time necessary to obtain the result of a convolution. Therefore, section 3.6.1 introduces the theory and circumstances under which a detour is possible that enables a faster convolution. The fast convolution itself is presented in section 3.6.2.

3.6.1 Theoretical background

Since unit point mass sequences are discrete signals by nature, equation 3.14 can be written as (for the sake of readability in Cartesian coordinates):

$$U_{x,y} = \sum_{x_1=0}^{R_x-1}\sum_{y_1=0}^{R_y-1} G_{x-x_1,y-y_1}L_{x,y} \tag{3.15}$$

where the load area, R, has width R_x and length R_y. The generalized definition of a convolution, however, allows for infinite functions. Thus, the bounds of the sums would range from $-\infty$ to ∞. It is obvious from equation 3.15 that in terms

of number of multiplications and additions a discrete convolution is an expensive operation. With L being composed of N unit mass points the complexity[8] is $O(N^2)$.

The amount of arithmetical operations can be reduced to $O(N \log N)$ when the convolution is performed in the spectral domain; an approach called *fast convolution*. Before the steps to achieve this performance improvement are explained (section 3.6.2), two fundamental tools must be introduced:

- the discrete Fourier transform (DFT), and
- the convolution theorem.

For the sake of simplicity, but without loss of generality, this is done for one-dimensional signals only. Given information is reduced to a minimum necessary for this thesis. An in-depth presentation of these subjects is found in the literature (e.g., *Stearns and Hush* (1999), *Meffert and Hochmuth* (2004)).

Discrete Fourier transform (DFT) In section 3.3 a *spectrum* was defined as a representation of a signal depending on the frequencies of its constituents. One function pair for transforming between spectral and original[9] representation of a signal is the discrete Fourier transform (DFT) and its inverse (IDFT, DFT^{-1}). The DFT supports decomposition of discrete and non-periodic functions of finite energy into their harmonic constituents which differ in amplitude, phase, and frequency. The frequency of the components is always a multiple of the fundamental frequency[10].

---

[8] The Landau-notation is used in computational complexity theory to express the maximum resource requirements of an algorithm. Depending on the input length the requirements are estimated asymptotically.

[9] Depending on the application and number of dimensions the particular naming for 'spectral' and 'original' domains differs. One-dimensional signals usually depend on time; thus, 'frequency' and 'temporal' domain would be used synonym. In two-dimensional applications identification with the 'wavenumber' and 'spatial' domain is common. For the sake of generality 'spectral' and 'original' domain are used throughout this thesis.

[10] Jean Baptiste Joseph Fourier suggested the possibility for such a decomposition for periodic and continuous functions in 1807. Due to much doubt in the scientific community about this fundamental idea enclosed in the work *Théorie analytique de la chaleur* (The Analytical Theory of Heat) it was not published until 1822 (*Meffert and Hochmuth*, 2004).

The one-dimensional transform pair is defined as (note opposite signs in the exponent):

$$\mathfrak{L}_m = \sum_{n=0}^{N-1} L_n e^{-j2\pi \frac{mn}{N}} \quad \text{with} \quad m = 0, 1, \ldots, N-1 \tag{3.16}$$

$$L_n = \frac{1}{N} \sum_{m=0}^{N-1} \mathfrak{L}_m e^{+j2\pi \frac{mn}{N}} \quad \text{with} \quad n = 0, 1, \ldots, N-1 \tag{3.17}$$

where $\mathfrak{L}_m$ is the m-th spectral component and L_n is the n-th element of the discrete signal which is of total length N. The transform and its inverse are linear and map values from the complex plane into the complex plane. Since both, the transform and its inverse, are calculated using the periodic[11] discrete exponential function, $\mathfrak{L}_m$ (equation 3.16) and L_n (equation 3.17) are periodic as well. This is an important property which uses to cause considerable confusion when not accounted for (e.g.) in fast convolution.

Figure 3.6 shows a signal discretized in time. The result of a one-dimensional DFT, $\mathfrak{L}_m$, is a vector of complex Fourier coefficients. Thus, real and imaginary part, or norm and phase have to be displayed separately (see figure 3.6(b-d)). The time signal is a discretized harmonic function consisting of a cosine and two sines (*Meffert and Hochmuth*, 2004). The real part of the spectrum contains the cosine (figure 3.6(b)), whereas the imaginary part contains both sines (figure 3.6(c)); one with a large amplitude and a small frequency and the other with a smaller amplitude and a twice the frequency.

An approach to perform a DFT more efficient than defined by equations 3.16 and 3.17 is called *fast Fourier transform* (FFT). Basically, symmetry and periodicity of the complex exponential function and aggregation of partial sums that have to be multiplied with the same factor are used to enhance performance. A precondition for the algorithm to operate most efficiently is that the input signals' dimensions are a power of 2 (see, e.g., *Meffert and Hochmuth* (2004)).

At this point it is important to mention that space-invariance is a fundamental

[11] An exponential function defined on the complex plane is periodic with the imaginary period $2\pi j$. It can be written as $e^{j\phi} = \cos(\phi) + j\sin(\phi)$.

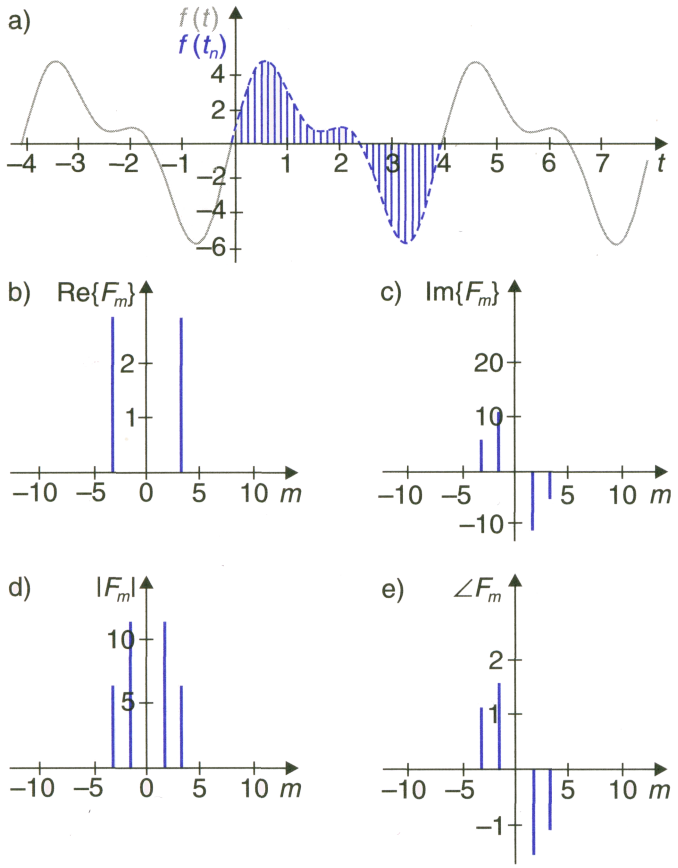


Figure 3.6: DFT example: **a)** original (time) signal discretized in interval [0,4). The spectrum of the result of the DFT is shown as: **b)** real part and **c)** imaginary part, and **b)** norm and **c)** phase, respectively (*Meffert and Hochmuth*, 2004)

prerequisite to this approach. Equations 3.16 and 3.17 show that *all* values of a signal are used to calculate the DFT and the IDFT, respectively. Thus, the result refers to the frequency content of the whole signal which makes it impossible to know anything about the spatial location of a single spectral element (e.g., *Meffert and Hochmuth* (2004)). In the original domain, however, it is possible to neglect spatial invariance since Green's functions are satisfied with the precondition of linearity. In modeling practice that means Earth models containing faults or other singularities have to be convolved with a load in the original domain utilizing the 'slow' convolution.

Convolution theorem If two signals are convolved in the original domain the *convolution theorem* states that this is equivalent to the multiplication of the Fourier

transformed signals (e.g., *Meffert and Hochmuth* (2004)). If

$$L_n \circ\!\!-\!\!\bullet\, \mathfrak{L}_m \quad \text{and} \quad G_n \circ\!\!-\!\!\bullet\, \mathfrak{G}_m$$

then

$$(L * G)_n \circ\!\!-\!\!\bullet\, \mathfrak{L}_m \cdot \mathfrak{G}_m \quad \text{with} \quad (L * G)_n = \sum_{m=-\infty}^{\infty} G_{n-m} L_m \tag{3.18}$$

The "$\circ\!\!-\!\!\bullet$" operator denotes the transform between the respective domains.

3.6.2 Convolving a Green's function and a load fast

Once the theory is introduced the fast convolution is a straightforward operation as shown in figure 3.7. However, two trapdoors deserve closer attention when using this approach:

- the fast convolution assumes periodical input, and
- the FFT takes only positive indices.

The first problem is due to the product of two DFTs always being periodic. This leads to an operation identified as circular convolution which induces an (mostly undesired) effect in the result which is referred to as *wrap around* effect. In contrast to linear convolution in the original domain the two signals *always* fully overlap in the spectral domain. To mimic a linear convolution in the spectral domain the signals have to be zero-padded to a size S greater or equal to $(x_1 + x_2 - 1)(y_1 + y_2 - 1)$, with x_i and y_i being, respectively, the width and length of both two-dimensional signals. If a FFT algorithm is used, S might be even greater since it must be a power of two (*Stearns and Hush*, 1999).

The second problem is due to the response function being centered around zero (as shown by the dotted region around G in figure 3.7). Here, the property of the DFT being periodic is beneficial: after zero-padding, the negative parts of the response function can be shifted to the far end of the signal (*Battle*, 1999). Then the signals are transformed using the FFT and the convolution is performed

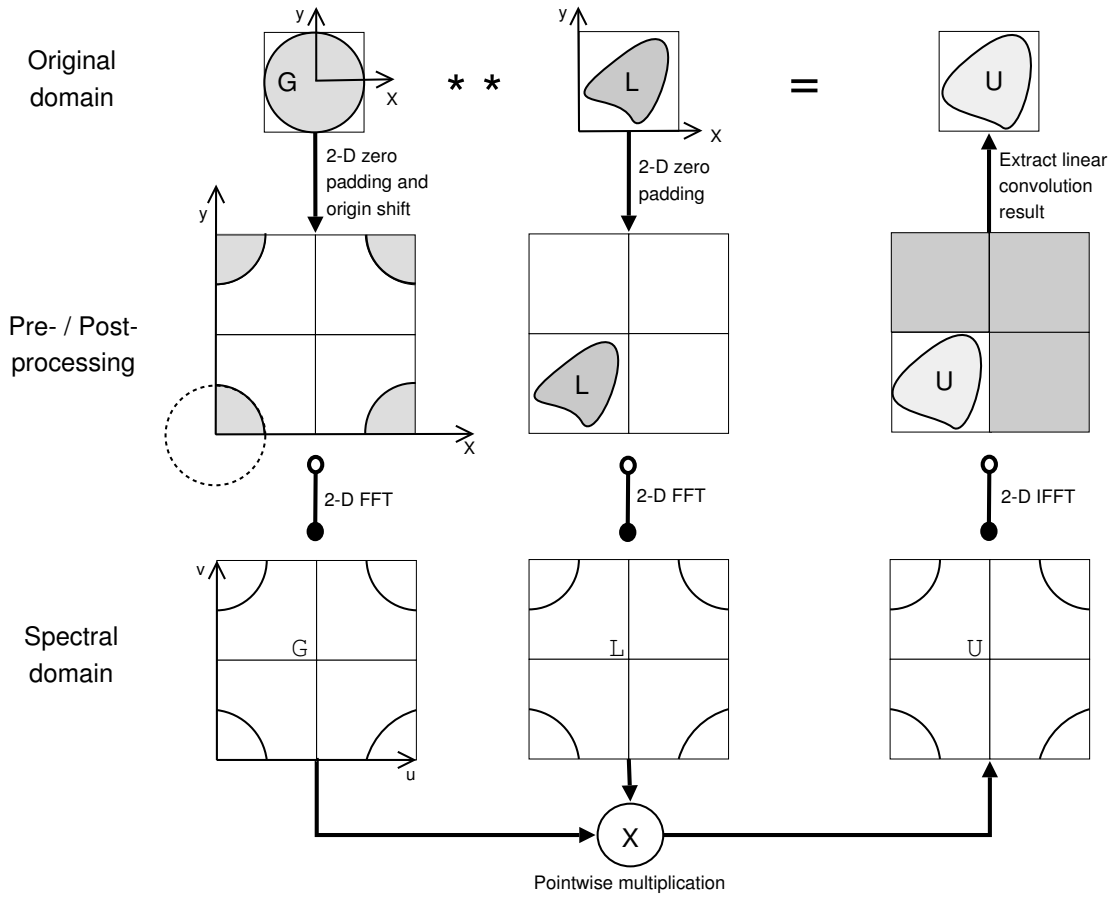


Figure 3.7: Block diagram for the formal model; Green's method. Convolution of a Green's function (G) with a load (L) depicted with two possibilities to obtain the convolution result (U) which are linked by the means of the convolution theorem. The top row of images illustrates the 2D convolution in the original domain and is analogous to equation 3.14. The columns refer to the steps necessary to utilize the fast convolution. In a pre-processing step G and L are zero-padded and G's negatively indexed signal parts (see dotted circle) are shifted to the far end of the zero-padded signal. After that, a 2D DFT (at best FFT) is applied to G and L. The bottom row illustrates the pointwise multiplication of the resulting spectra $\mathfrak{G}$ and $\mathfrak{L}$. In the right column post-processing takes place; basically performing the preprocessing in reverse order. First an inverse two-dimensional DFT is applied to the spectral product $\mathfrak{U}$. Following that, the region of interest is extracted from the result to remove the zero-pad. The convolution theorem states that the result U obtained by using the fast convolution is the same as obtained by convolving G and L in the original domain. (figure concept from *Battle* (1999), changed)

by pointwise multiplication of the spectra as suggested by the convolution theorem. The inverse transform is applied to the result and the zero-pad can be removed to obtain the linear convolution result.

3.7 Summary

We followed the scheme presented in figure 2.2 up to the point of deriving formal models for the examined system. This was only possible after the theory of filters adapted from signal processing was applied to the original system to derive an informal model. The author introduced the concept of Green's method as a formal model and thus a means to infer the lithospheric response to the mass force of a load. Then two particular formal models for the subsystem lithosphere, to be used in Green's method, were introduced: Green's functions for the elastic half-space and for a thick elastic plate lying over and inviscid fluid. These functions form the foundation of Green's method as applied here since they formulate the characteristics of the lithosphere to which a mass force (whose mathematical formulation is rather simple in comparison) is applied. The discrete Fourier transform (DFT) and the convolution theorem were introduced to enable performing a convolution as a multiplication in the spectral domain. This so-called fast convolution lowers the computational complexity of a convolution from $O(N^2)$ to $O(N \log N)$, with N being the length of the signal vector. The fast convolution therefore presents an efficient algorithm for implementation of the formal model (see objectives in section 1.2).

4 The composable simulation model: A plug-in based simulation framework

"And while those trays certainly didn't have much glamour they nevertheless had the hidden strength of a card catalog."
(Robert M. Pirsig)

With regard to figure 2.2 two major tasks are left to be tackled in this chapter: deriving the composable simulation model and its test and validation. Chapters 2 and 3 provide the necessary background to transform the formal model into a software system. In this chapter the author develops a plug-in based simulation framework (PSF) that supports the functionality needed to realize the formal model, Green's method, derived in chapter 3 on the basis of software components.

In section 4.1 the author specifies the simulation framework in a general way. Software components are identified and the logical data flow (input/output) between them is described. Based on this, the suggested software architecture of the PSF is detailed in section 4.2 which also precisely defines the terms 'plug-in' and 'plug-in based simulation framework'. The architecture consists of three logical layers: the functional, the interface, and the management layer. Their general function is described in section 4.2 as well. To ease the understanding of the subsequent sections, section 4.3 presents a brief overview of implementation specifics, e.g., implemented plug-ins and their handling in an Unix environment. The runtime scenarios illus-

trated by sequence diagrams in section 4.4 are utilized to explain the simulation frameworks general functioning (an in-depth presentation of the implementation which might help to gain a deeper understanding is given in appendix E). Since the author implemented both the PSF and the plug-ins, testing is necessary. Section 4.5 gives an overview on the testing that was carried out and discusses how the composable simulation model was validated. An evaluation of the PSF's capabilities is given in section 4.6.

4.1 Specification of the simulation framework

The idea behind the simulation frameworks architectural concept is the emulation of Green's method (see section 3.4) on which the formal models of section 3.5 are based (compare to equation 3.14)[1]:

$$U = G * *L \tag{4.1}$$

G and L are, respectively, a Green's function and a load function. U is a surface displacement resulting from a two-dimensional convolution of G and L. In this general form no dependencies on any model details exist. As for the architecture, a transfer of the modularized structure that is inherent to equation 4.1 into a simulation framework is intended. This is an implementation of the KISS principle. Hence, G, $**$, and L represent subsystems as defined in section 2.4. They are software components of the composable simulation model. Both, the operands (G and L) and the operator[2], are to be realized within the simulation framework in a way that represents their variable nature.

The intended modularized structure enables software component reuse (see

[1] This is basically equation 3.14 in an intentionally more simplified notation. Green's method is certainly not confined to a two-dimensional convolution. However, this operation applies to the problem in this thesis and is therefore used in all equations. The operator also indicates the dimensions of G, L, and U.

[2] As implied in section 3.6, a convolution can be performed in at least two different ways, namely in the original and spectral domain.

section 2.4 and figure 2.3) within the software components. The meaningfulness of this feature within the simulation framework is obvious from equations 3.7 and 3.8. The Green's functions for the elastic half-space (equations 3.3 and 3.4) are used in the formulation of the formal thick plate model for the lithosphere. Thus, it is reasonable to reuse the software component that implements the elastic half-space model in the software component that implements the thick plate model.

However, software component reuse / composability poses the problem of dependencies between software components. To break down the complexity of solving these dependencies (see NP-complete argument in section 2.4), reuse is restricted to software components of the same *category*. Categories, understood as a structuring element for the simulation frameworks software components, form proper subsets of the set of available software components. These subsets subsume different implementations of a particular task in the simulation framework. For instance, each of the terms G, $**$, and L of equation 4.1 is understood as such a category.

Given that the Green's functions meet the prerequisites for the fast convolution (see section 3.6), one software component of each category can be picked and combined freely to compose a simulation model following the structure of equation 4.1[3]. Figure 4.1 depicts a logical data flow chart of the simulation frameworks specification this far; reuse within categories is illustrated by reflexive arrows.

Figure 4.1 includes three additional software components which are not (an explicit) part of the formal model. It is, however, found useful to provide them as software components as well:

- The *load history* was already identified as a nonexchangeable system element of the Earth-load-system (section 3.1), but yet not included in the formal model since it is a mere alteration of the load characteristics depending on time. Therefore, figure 4.1 makes clear that the load history is transparent to the convolution operator. The load function retrieves the data directly from the load history.

[3] However, a Green's function does not need to fulfill the prerequisite of space-invariance. These functions can only be combined with operators that convolve in the original domain.

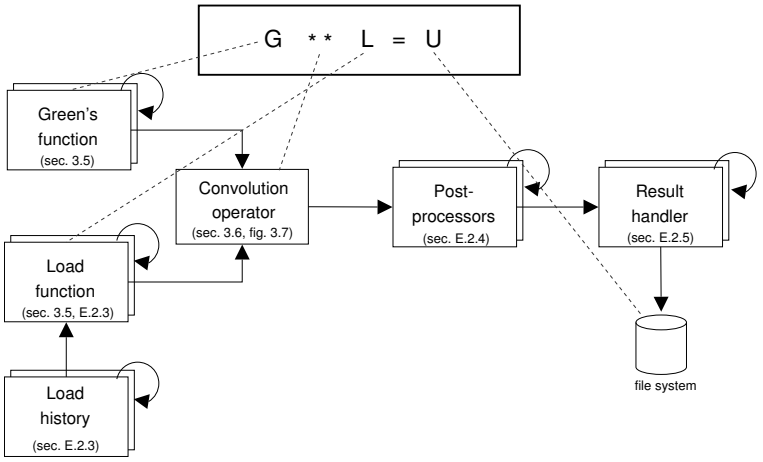


Figure 4.1: Data flow (arrows) between important software components (boxes) of the proposed simulation framework. Since the framework is supposed to resemble the formal model (Green's method) the gray dotted lines denote the analogy between terms of equation 4.1 and software components. Additional software components are added to increase functionality and flexibility. Reflexive arrows refer to a software component that accesses functionality of another software component of the same category. The convolution operator takes data from the Green's and load function. The latter might be affected by a load history depending on whether or not a time dependent load is to be simulated. Once a convolution result for the examined area is calculated, one or more post-processors can be applied to the result. Finally, the results are passed to a result handler that writes the modeling results in a particular format to the file system. The terms in brackets denote references to sections where the respective component is detailed.

- The *postprocessors* apply functions to the model results. Applicable operations are, for instance, simple coordinate system conversions or comparisons to real data.

- The *result handler* takes the obtained (and post-processed) modeling results and writes them to the file system. The intention for providing it as a software component lies with the allowance for a multitude of (user defined) output formats. Furthermore, the result handler can be 'misused' to provide an interface to third-party software. This way, the simulation framework could be utilized as a subsystem in a more complex software system that conforms to the analysis objective (compare to the 'full (composable) simulation model reuse' scenario in section 2.4).

Figure 4.1 shows that the data generated by a Green's function and a load function (possibly influenced by a load history) flow into a convolution operator. Both functions might depend on other software components of their category. Once the convolution operator possesses the necessary data it can produce a model result.

This can undergo alteration or extension by several postprocessors. Finally, a result handler will write the data to a file of arbitrary format.

Even though the formal model is mapped to software components and the scheme depicted in figure 4.1 could be implemented in some way, some thought must go into the paradigm the simulation framework shall implement.

Independent from the domain in which the convolution is performed, space will be treated in a discretized way. This is obvious since the load is a grid of unit point masses (see section 3.4). Due to their complex, apparently random character real load specifics (height, density distribution, see equation 3.2) cannot be described by continuous functions. Thus, values at discrete points are used as averages over small areas to model a real load. The formal models for the lithosphere introduced in section 3.5 do not include any singularities like fractures. Such cases might need adaptive gridding with a greater spatial resolution in areas of these singularities. The distance between load points in the simulation framework is in any case regarded as uniform over the modeled area. This grid-size must be a parameter determinable by the user. The temporal evolution of surface displacements, on the other hand, depends on two processes:

- changes of the load over time, i.e. the load history, and
- the relaxation of the lithosphere over time, i.e. the replacement of liquid mantle material.

With regard to nature both are simultaneously working, continuous processes. Currently only load histories are included in the simulation framework. The application of load histories should therefore be limited to simulations that employ Green's functions for an instantaneous response of the Earth since this gives a 'final' displacement at each point in time. *Pinel et al.* (2007) suggest a formal model for the relaxation process based on the convolution with time. Its implementation is, however, beyond the scope of this thesis.

The model results for Earth models that are more realistic than the instantaneous response depend on the behavior of the lithosphere over time. Therefore, model results are calculated at equidistant time steps of user-determined size. The option of implementing the event driven simulation model paradigm (see section 2.2) is neglected since an event could only be triggered by a load change. Although load histories have not been examined very deeply within this thesis (an example is equation E.1), it is clear that parts of the lithospheric response would be missing in the results if advance in time was modeled event based.

These considerations suggest a simulation framework that implements a discrete time- and space-stepped simulation model paradigm.

4.2 Architecture

To achieve the objectives of enabling users to:

- exchange software components,
- extend the simulation framework independently with new software components, and
- dynamically select software components that participate in a simulation,

a software architecture that supports the utilization of plug-ins is proposed in this section.

The vague phrase 'software parts', used in the definition of the term *plug-in* in section 2.1 which was supposed to give a mere intentional understanding of the topic of this work, is now replaced by the concept of a software component:

Definition 2. *A* **plug-in** *is a software component the user can add to or remove from a software application to alter its functionality. An application utilizing plug-ins must provide an infrastructure that, additionally to the creation of plug-in instances (see 'software component instances'), supports a mechanism that lets the user manage the plug-ins of a software application. An infrastructure that provides this functionality is referred to as* **plug-in based simulation framework** *(PSF).*

To convert the concept presented in figure 4.1 into a PSF the architecture shown in figure 4.2 is proposed. Due to the management effort that comes with the use of plug-ins, the proposed architecture is structured into three logical layers (figure 4.2):

- a *functional layer* that contains the plug-ins associated with the PSF,
- an *interface layer* which contains the provided[4]- (Framework API) and needed[5]-interfaces (plug-ins) of the simulation core (see below), and
- a *management layer* that provides the functionality the infrastructure must possess to realize plug-in instances (see section 2.4).

The software components whose logical data flow is depicted in figure 4.1 are included as plug-ins at the bottom of the scheme in figure 4.2. According to their nature of adding all simulation functionality to the PSF (the *complete* formal model is found in this layer), they form the functional layer. Interface and management layer implement the logical data flow between the software components as shown in figure 4.1 and therefore add composability to the simulation model (see section 2.4).

The central element of the PSF is the simulation core. It contains the main loop which defines the sequence in which plug-ins and management elements must operate to produce a model result. Furthermore, the simulation core manages the function calls that plug-ins invoke at the PSF's provided-interface, the framework API. The simulation core will route the call to the respective needed-interface which can access a plug-in instance that provides the requested functionality.

The input handler associated with the simulation core interprets the experiment definition (see listing D.1 for an example). On the basis of this document a simulation with the PSF is configured. The plug-ins that are wanted to participate in a simulation, their parameter settings, and additional information such as the coordinates of the region of interest and the spatial and temporal step sizes are defined in this file. The input handler is responsible to validate[6] the experiment definition

[4] Provided-interfaces are interfaces where a software component offers a functionality.

[5] Needed-interfaces are interfaces where a software component requests a functionality that is offered, e.g., by a provided-interface of another software component.

[6] In this context 'validation' refers to the confirmation that a document meets a defined format.

to a certain degree and make its contents accessible to the simulation core in an appropriate form (e.g. convert parameter values from text to numbers).

The plug-in manager associated with the simulation core has access to a plug-in repository and a plug-in database. Metadata that must be provided by each plug-in is stored here on a plug-ins registration with the PSF. This includes for example name, description of provided functionality, category, and location of the dynamic

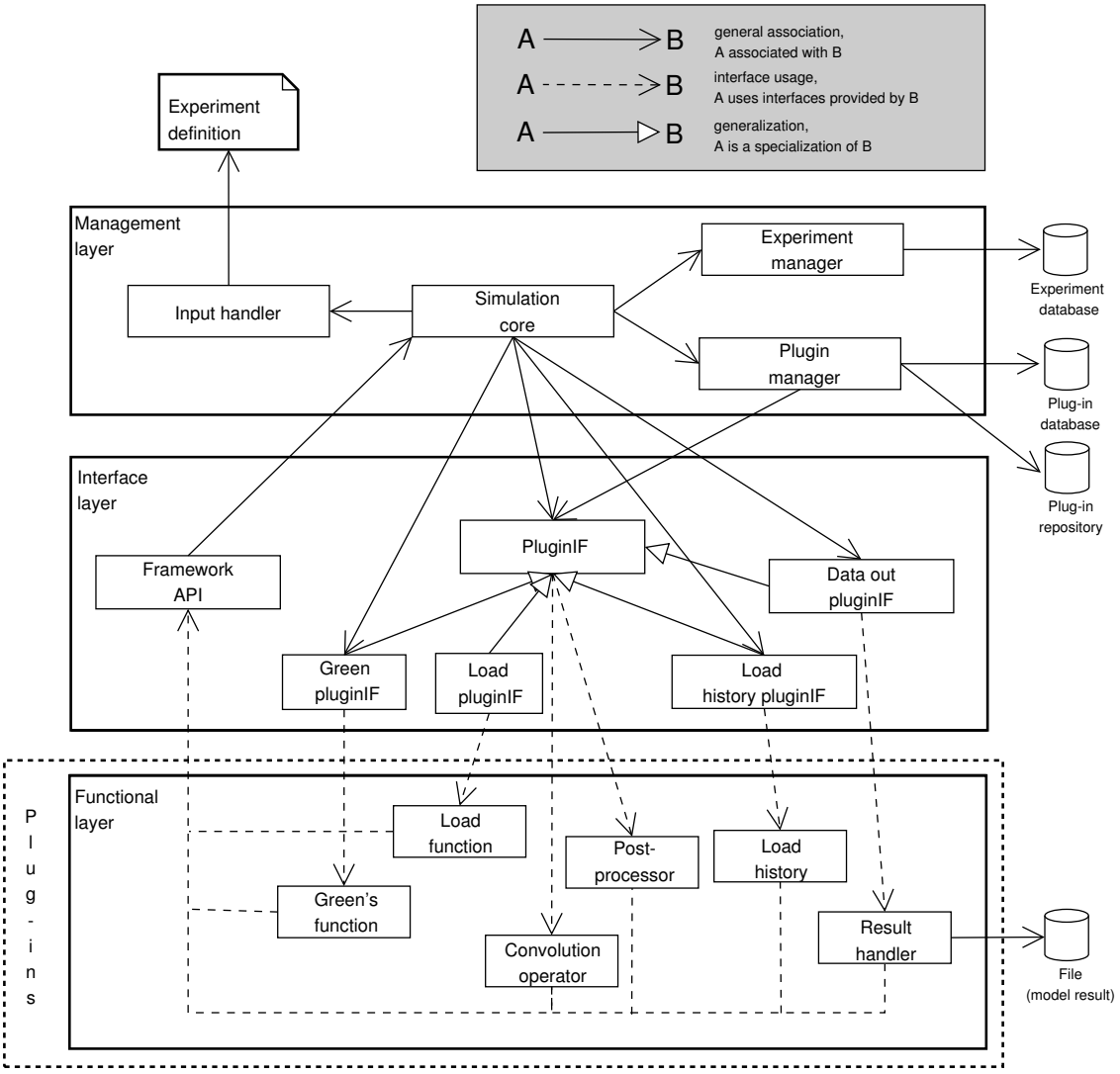


Figure 4.2: Architecture of the proposed plug-in based simulation framework. The software components of figure 4.1 are represented by plug-ins of the functional layer. The logical data flow depicted in figure 4.1 is realized by the interface and management layers which provide the infrastructure to support communication between the plug-ins. Input handler, experiment manager, and plug-in manager provide additional functionality to implement the plug-in concept. The experiment definition contains the configuration of a simulation.

library within the plug-in repository[7]. This is in accord to the demands of *Overstreet et al.* (2002) who identify the capturing of objectives, assumptions, and constraints as a key to reuse (see section 2.4).

Given the name of a plug-in, the plug-in manager can figure out whether this plug-in is registered with the PSF. The plug-in manager returns the plug-ins physical location in the repository (so-called path resolving) to the simulation core in case it is registered. Before being registered with the PSF, the plug-in manager checks whether or not a plug-in provides implementations of the frameworks needed-interfaces. In case it does, the plug-in is added to the plug-in repository.

Knowing the locations of the plug-ins, the simulation core can create instances of its needed-interfaces (e.g., `PluginIF`) and request each of them to bind an instance of a plug-in. Figure 4.1 shows which interface binds an instance of which plug-in (the provided-interfaces of the plug-in instances are connected to needed-interfaces of the PSF). The difference between the several needed-interfaces is mainly the parameterization of the 'run'-function which must provide the functionality of a plug-in. The 'run'-function represents an analogy to the `main` function known from several programming languages. The functionality of a 'run'-function of a load function plug-in would be, for instance, to calculate the unit point mass at a given point of the modeled area. For this, it would need the points coordinates as parameters. A post-processor, however, is simply told to work; without any parameters.

Functions that access a plug-ins metadata are inherited[8] from the `PluginIF`-interface. Loaded plug-ins use the frameworks API to request, e.g., data from other plug-ins or resources from the PSF.

The experiment manager associated with the simulation core keeps a history of the simulations (i.e. experiments) run by the PSF. Metadata, such as parameter settings, modeling result location, utilized plug-ins, user, time, and date are stored

[7] The plug-in repository is the location in the file system to store the dynamic libraries in a controlled and organized way.

[8] In object-oriented design an inheritance relation describes, here at the example of interfaces, that an interface A that inherits interface B contains B (e.g. all its functions) as a subset. Thus, at runtime instances of interface A can be interpreted as both interface A and (with the limited functionality of B) interface B.

in an experiment database to allow later matching of modeling results to, and reconstruction of experiments.

The architecture of the PSF must furthermore account for two additional, important features that are associated with the simulation core in the management layer (to limit the figures complexity, these features are not included in figure 4.1):

- The functionality of a plug-in may depend on an unknown number of parameters (defined by the plug-in) which get their values for each simulation assigned in the experiment definition. Thus, a *parameter registry* must be implemented and associated with the simulation core to link the definition and the initialization of the parameters. Every plug-in can register an arbitrary number of parameters with this registry. Before a simulation starts the input handler must assure that the user provided all necessary values and no parameters remain uninitialized in order to get each loaded plug-in to operate.

- A *plug-in registry* must be implemented to enable a plug-in to access the functionality provided by another plug-in of its category.

4.3 Implementation

The architecture for a PSF as proposed in section 4.2 is implemented by the simulation framework CrusDe[9] which the author developed within this thesis. Some specifics on CrusDe's implementation are given in this section. This includes information on some of the implemented plug-ins, how a plug-in concept is to be implemented in an Unix environment, and a short presentation of the files that are involved in a simulation. Since not all the implementation specifics can be included here further details are given in appendix E.

[9] `http://www.grapenthin.org/projects/crusde`

The plug-ins are physically represented by dynamic libraries in the plug-in repository (file system) of the simulator (see appendix E.3). The reference implementation of the plug-ins is carried out using the programming language C. This decision is not mandatory for future plug-ins as long as the symbols (e.g. function names) appear in the expected way in the dynamic library (see appendix E.1 for details on names of needed-interfaces). A reason to provide the reference implementation in the programming language C is that functions of a dynamic library are easiest accessed when defined in a C rather than a C++ manner (e.g., *Vandevoorde* (2006)). Both the management and interface layer are implemented in the programming language C++ which is necessary since object-oriented design techniques such as inheritance were found useful to apply.

4.3.1 Selected plug-ins

Within this thesis at least one plug-in per category has been implemented. This section briefly describes the implemented plug-ins for some 'key'-participants of a simulation to derive surface deformation. These include the convolution operator, the Green's functions, load functions, and a result handler:

Operator: fast 2d convolution. As shown in figure 4.2 a convolution operator has to implement the simulation cores `PluginIF`-interface. The particular plug-in is not only responsible to perform the convolution, but must also take care on the memory management of the convolution operands and the convolution result. This is mainly due to flexibility considerations: an operator might want to have the operands and result in a special form (see 3.6). Thus, providing operand and result memory, for instance, within the simulation core would cause unnecessary overhead and negative runtime effects.

The one convolution operator that is part of CrusDe implements a fast two-dimensional convolution as shown in figure 3.7. To realize this operation, the '`fast 2d convolution`' plug-in utilizes version 3 of the "Fastest Fourier

Transform in the West"[10] (FFTW) software library (*Frigo and Johnson*, 2005). FFTW provides the necessary transformations (DFT, IDFT) to transform the operands to the spectral domain and the result of the complex multiplication back to the original domain.

The '`fast 2d convolution`' plug-in does not register any output fields or parameters.

Green's functions: elastic half-space & thick plate. A Green's function plug-in must implement the simulation cores `Green PluginIF`-interface (see figure 4.2). The following plug-ins provide implementations of formal Earth models within CrusDe:

- '`elastic halfspace (pinel)`' implements the Green's functions given by equations 3.3 and 3.4, and
- '`thick plate (pinel)`' implements the Green's functions given by equations 3.7 and 3.8.

On loading, both plug-ins register three output fields (they calculate surface displacements in the x, y, and z direction). Furthermore, the '`elastic halfspace (pinel)`' plug-in registers parameters:

- `g`, the acceleration due to gravity with the SI-unit[11] $[m\,s^{-2}]$,
- `E`, the Young's modulus with the SI-unit $[GPa]$, and
- `nu`, the Poisson's ratio which has no unit.

Additionally, the '`thick plate (pinel)`' plug-in registers the parameters:

- `H`, the plate thickness with the SI-unit $[m]$, and
- `rho_f`, the density of the underlying fluid with the SI-unit $[kg\,m^{-3}]$.

[10] `http://www.fftw.org`

[11] Système international d'unités

Load functions: disk load & irregular load. To serve CrusDe as a load function, a plug-in must implement the simulation cores `Load PluginIF`-interface (see figure 4.2). Both implemented load function plug-ins expect the parameter `rho` in $[kg\,m^{-3}]$ to be provided in the experiment definition.

The '`disk load`'-plug-in defines an uniform load in the shape of a disk in the examined area. Parameters that have to be defined in the experiment definition are (SI-unit $[m]$, see listing D.1):

- `center_x`, `center_y`, define the disks center,
- `radius`, defines the radius of the disk, and
- `height`, defines the height of the disk.

The '`irregular load`' plug-in differs from the `disk load` mainly in that it reads the load heights from a separate load file. The load file is expected to be in a tabular format with three columns separated by spaces and each row containing only numbers. The values of each row are interpreted as:

```
Longitude Latitude Height
```

Longitude and latitude are expected to be integers in Lambert coordinates $[m]$.

Result handler: netcdf writer A result handler plug-in must implement the `Data out PluginIF`-interface (see figure 4.2). One of the two result handlers that are implemented within this work is the '`netcdf writer`'-plug-in. It is described here, because it utilizes version 3.6.1 of the netCDF[12] C library which implements a standardized and machine independent format for writing and reading of array-type data (*Rew et al.*, 2006). Data of this type are widely used in geoscience and have a very good tool support[13].

[12] `http://www.unidata.ucar.edu/software/netcdf`

[13] E.g., `ncview` and `GMT` (Generic Mapping Tools) for data display and `NCO` (netCDF Operators) for data manipulation.

Depending on the domain in which netCDF files are used, several conventions[14] for variable-naming, data units, and other necessary arrangements to assure portability between programs are agreed upon. The '`netcdf writer`' plug-in implements the Cooperative Ocean/Atmosphere Research Data Service (COARDS) conventions which are implemented by many other geoscience tools that support the netCDF format. This frees CrusDe, for instance, from the need to implement a viewer for the modeling results.

This plug-in does neither register parameters nor output fields.

The interfaces each plug-in must implement are detailed in appendix E.1. More detailed descriptions for all implemented plug-ins are given in appendix E.2.

4.3.2 Plug-ins in Unix environments

As mentioned above, the physical representation of a plug-in within CrusDe is a dynamic library. This enables loading of a plug-in into the main application at any point in runtime. On most Unix-like environments such libraries are called 'shared objects' usually found in files with the suffix '`.so`', whereas in the Microsoft Windows world they are referred to as 'dynamic link library' carrying the suffix '`.dll`'. Basically, both represent an identical concept but are to be handled slightly different. This section gives a brief overview on how the plug-in concept is to be realized on source code level in a Linux environment. Instructions on how to build a dynamic library are found in appendix E.3.

Three steps are necessary to make use of a plug-in once the plug-in manager mapped the name of a plug-in given in the experiment definition to a position of the respective dynamic library in the plug-in repository. These steps are based on the functions `dlopen`, `dlsym`, and `dlclose` (*Linux manual*, 2003) of the dynamic linking loader. These functions are used to describe a load-access-unload cycle in CrusDe:

`void *dlopen(const char *filename, int flag)` – **load a plug-in:** Given the position of the dynamic library, the function `dlopen` loads the dynamic library

[14] `http://www.unidata.ucar.edu/software/netcdf/conventions.html`

and returns a 'handle' to it. The `flag` determines whether symbols[15] in the library are to be resolved when they are needed (`RTLD_LAZY`), or instantaneously upon loading of the library (`RTLD_NOW`). This function also increments a reference counter that expresses how many instances of the library are used.

`void *dlsym(void *handle, const char *symbol)` – **entity access:** This function returns the memory address to which a `symbol` of the dynamic library referenced by `handle` is loaded. Invoking a function pointer that is assigned the returned memory address will invoke the function of the dynamic library.

`int dlclose(void *handle)` – **unload plug-in:** This function decrements the counter of references to `handle`. The dynamic library is unloaded from main memory by `dlclose` as soon as the reference counter is zero.

These steps are implemented within the `PluginIF`-interface and therefore available to all inheriting interfaces. Calling the `load` function at the `PluginIF`-interface with the path to a dynamic library as a parameter, `PluginIF` will try to create an instance of the plug-in using `dlopen` and bind all interfaces using `dlsym`. The `unload` function of `PluginIF` invokes `dlclose` to destroy the plug-in instance.

4.3.3 Experiment files

Obvious from figure 4.2, CrusDe employs several files that have to meet specific formats. Sorted into input and output of a simulation – the most common use case of CrusDe; others are experiment or plug-in management – these files are:

- Input: experiment definition, plug-in database[16], experiment data (load), and
- Output: experiment database and model results.

The work on experiment data and model results is entirely done by plug-ins (see appendices E.2.3 and E.2.5). The format of these files depends on the particular plug-in chosen by the user.

[15] For example, names of variables or functions.

[16] The plug-in database is used for output when plug-ins are added. This is, however, a different use case.

All of the other files are in formats defined by the Extensible Markup Language (XML)[17]. Reasons to use XML include that it is a simple and flexible format for text documents standardized by the Wold Wide Web Consortium (W3C), and it is supported by many freely available tools.

A program that reads and writes XML documents is referred to as XML-processor or parser. Xerces-C++[18] is a XML-processor written in a portable subset of the C++ programming language and conforms to the XML 1.0 and associated standards. This parser has the advantage of validating XML documents, i.e., it automatically compares a document to a given Document Type Definition (DTD). A DTD specifies elements of an XML document and rules for its format. This way it can be assured that CrusDe's expectations on the format of a particular file are met.

4.4 Runtime scenarios

To give a better understanding of CrusDe's processing chronology this section utilizes three sequence diagrams to illustrate the functioning of the framework. At first a short introduction to the UML sequence diagram notation is given in section 4.4.1 which is necessary to fully understand the diagrams. The first sequence diagram in section 4.4.2 depicts an example of a full communication sequence between two plug-ins via the simulation core. The subsequent diagrams will display 'shortcuts' when it comes to plug-in communication to limit redundancy and complexity. Each of these 'shortcuts' can be replaced by a sequence similar to the one displayed in figure 4.3. Section 4.4.3 describes the initialization sequence of CrusDe. This gives a general idea of the work necessary to compose a simulation in accord to a given experiment definition. The execution sequence of CrusDe is presented in section 4.4.4. There it is shown how the logical data flow depicted by figure 4.1 is realized based on the architecture presented by figure 4.2.

[17] `http://www.w3.org/XML/`

[18] `http://xml.apache.org/xerces-c`

4.4.1 UML sequence diagrams

A sequence diagram allows the graphical representation of runtime scenarios of a software application. The notation of the presented sequence diagrams follows mainly the syntax of the Unified Modeling Language (UML) 2.0 (e.g., *Störrle* (2005), *Wikipedia contributors* (2007)). Without going too much into the details of this standardized graphical specification language, some of its symbols must be explained.

Participants in a sequence diagram are depicted by heads with attached parallel, vertical lifelines. A head contains the name of the participant ('`name:type`'). Horizonal arrows express messages exchanged between participants in the timely order in which they occur in the software application. Messages can have a name written above the arrow. If the arrow is solid with a filled triangle head, it represents a synchronous call. The response to a synchronous call is a dashed arrow with a lines head (omitted if the response is clear). The return value can be written over the arrow. Rectangles drawn on the lifeline represent activation or method call boxes, respectively. They illustrate that a participant is activated in order to respond to a message. Since CrusDe does not implement any parallel processing, only one participant is active at a time.

Rectangles with a little text area in the upper left corner represent combined fragments which denote a complex interaction. The text area contains an interaction operator (e.g. '`alt`' for an alternative, '`loop`' for a repeated interaction, '`ref`' for a reference to an external sequence diagram). The rest of the combined fragment contains the operands. In the sequence diagrams presented in this thesis an operand is usually a synchronous call.

4.4.2 Plug-in communication sequence

The plug-in communication is shown at the example of the convolution operator instance `f2dc` (fast 2-dimensional convolution) that requests the unit point mass of a load at point `(x,y)` from the instance of a load function plug-in. The time `t` is irrelevant, because it is assumed that no load history function is modeled. Figure 4.3

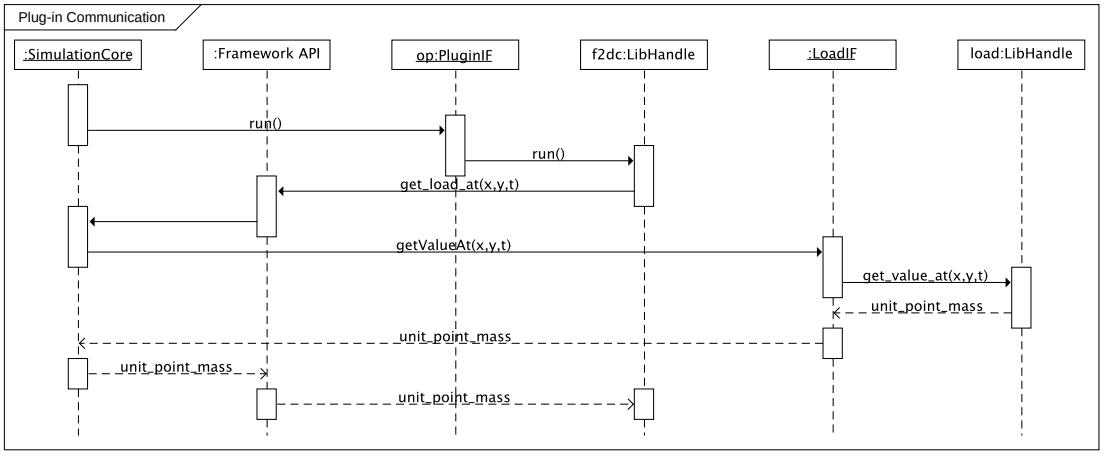


Figure 4.3: CrusDe plug-in communication sequence

shows the respective communication sequence.

The first two function calls in figure 4.3 (`run`) mark the entry point to the sequence which makes `f2dc` send the request for a load value. The request for a value at point `(x,y)` and time `t` is sent to the simulation core via the frameworks API function `get_load_at`. The simulation core routes this request to an instance of the load interface `LoadIF`. This interface instance bound the needed-interface `getValueAt` to `get_value_at` during initialization. The function `get_value_at` is a provided-interface of the load function plug-in instance `load`. Since no load history is modeled the unit point mass of the load at point `(x,y)` can be calculated directly by `load`. The result `unit_point_mass` is returned to `f2dc` via all instances that were involved in forwarding the request to `load`.

4.4.3 Initialization sequence

An excerpt of the initialization sequence of CrusDe is shown in figure 4.4. The `main` function calls the simulation cores `init` function to enter this sequence. At first the simulation core initializes the associated objects of the management layer. For the sake of space preservation only the call of the `init` function at the input handler instance is shown in figure 4.4. The input handler instance reads the experiment definition on initialization (`readExpDef`). Its contents is now available to CrusDe.

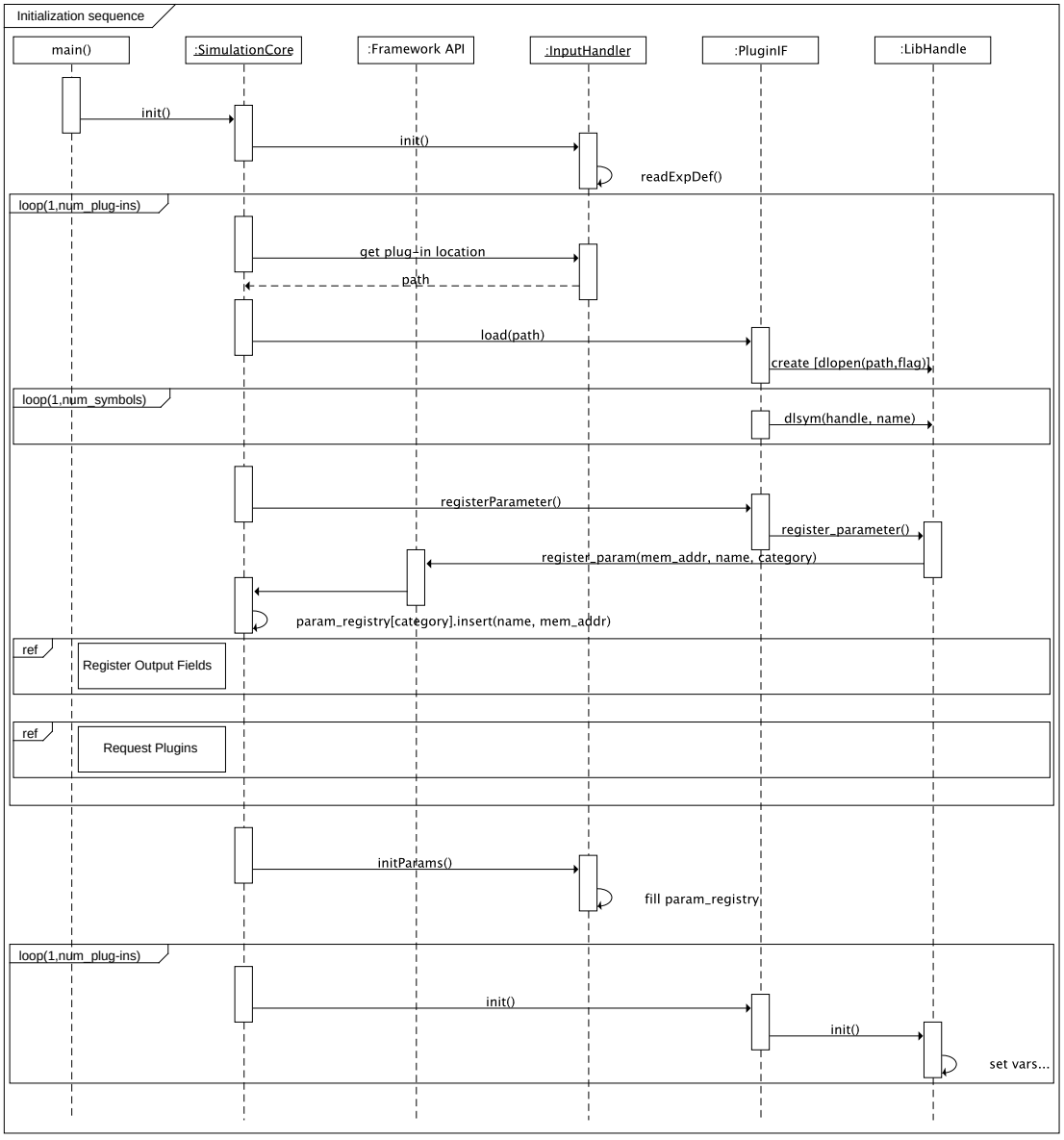


Figure 4.4: CrusDe initialization sequence

In a giant loop[19] follows loading and initialization of all the plug-in instances requested in the experiment definition. At first, the simulation core gets the name of a plug-in from the input handler. With some assistance from the plug-in manager the plug-ins name is mapped to a location in the plug-in registry. This `path` is then given to an instance of `PluginIF` using its `load` function. The interface instance creates an instance of the plug-in by loading the shared library found at `path` using the `dlopen` interface to the systems dynamic linking loader. The interface instance then binds all the needed-interfaces it defines by loading the plug-ins provided-interfaces using `dlsym`.

If successfully created, the simulation core will request the plug-in instance to register its parameters. A plug-in must implement the function `register_parameter` in order to make this possible. Within this function a plug-in instance can call the `register_param` function at the frameworks API. The parameters address in memory, its name and the plug-ins category must be provided with this call. The simulation core saves the name/address pair to the field of the parameter registry that is reserved for `category`. If different plug-in instances of the same category will register a parameter with an identical name, both are assigned the same value that is assigned first in the experiment definition.

The registration of output fields and plug-ins is analogous to the registration of parameters and therefore only indicated by references to external sequence diagrams in figure 4.4. When requesting the functionality of an existing plug-in, the name of the requested plug-in must be given to the frameworks API in the same format as it would be defined in the experiment definition. The simulation core will create an instance of the requested plug-in just as described above and – upon success – return a pointer to the requested plug-ins 'run'-function (see appendix E.1 for details on both).

Once all plug-ins are created and have their parameters registered the simulation core calls the input handlers `initParams` function. The experiment definition is

[19] In the program sources this is only to some extent true. Plug-in instances not directly bound by `PluginIF` are loaded and initialized by the respective interface instances outside the loop. For simplicity, however, it is depicted in this more general way.

searched for parameter names as they occur in the parameter registry and – in case they are found – the parameter values are written to the memory address associated with the parameter names. In case a parameter is not found in the experiment definition, the simulation is aborted and CrusDe exits. After that, all plug-in instances' parameters are initialized and the `init` function of each plug-in instance can be called. This way the plug-in instances can, for example, compute constants that are used during the simulation on the basis of the parameter values.

4.4.4 Execution sequence

Figure 4.5 depicts CrusDe's realization of the logical data flow as shown in figure 4.1. For clarity, the interface layer is completely omitted in the sequence diagram since chronological order and the actors are of major interest (see section 4.4.2 for communication details).

Most of the sequence in figure 4.5 is enclosed in a loop that repeats at least once and at most `time steps` times which is a value that can be defined by the user in the experiment definition. At each time step the simulation core will invoke the `run` function of a convolution operator instance which in the depicted case is the fast two-dimensional convolution `f2dc`. The operator will then iterate over the simulation area (`R`) and request values from the Green's function (for all spatial directions the particular Green's function calculates) and from the load function to fill the inputs for the DFT (see figure 3.7). Therefore, `f2dc` invokes `get_array_at` which has a pointer to an array the size of displacement directions as parameter. This function calculates the defined Green's function at point `(x,y)` and writes the results into the given array. Upon that the load function is invoked by calling its `get_value_at` function. If a load history instance exists, the load function instance requests the load height at time `t` by calling `constrain_height`. A `unit_point_mass` for point `(x,y)` (at time `t`) is returned to `f2dc`.

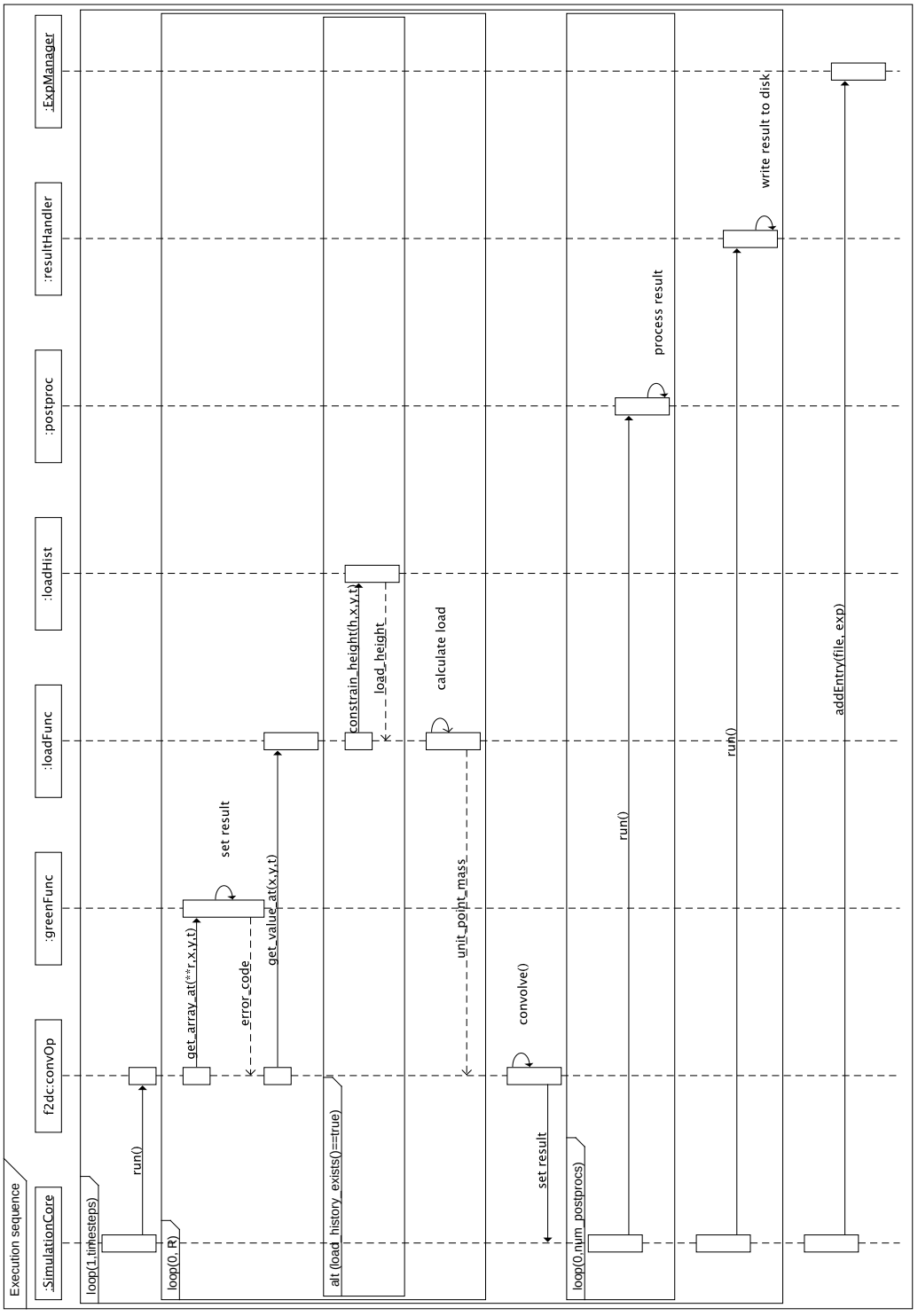


Figure 4.5: CrusDe execution sequence

Once values for each point in `R` are obtained, `f2dc` can perform the fast convolution (see section 3.6) and inform the simulation core about the location of the convolution result in memory using `set_result`. The simulation core will then loop over all defined postprocessor plug-in instances and have them execute their `run` function on the obtained result. The model result for the actual time step is then written to the file system by a result handler plug-in instance which defines the output format of the data. Finally, after the processing is done for all time steps, the experiment manager gets to archive this simulation in the experiment database.

Comparing the approach described above to the logical data flow as outlined in figure 4.1, it is obvious that differences are only about who is pulling and who is pushing the data.

4.5 Testing and validation

The introduction to this chapter mentioned two tasks that are to be worked on in this chapter. With the implementation of the plug-in based simulation framework (PSF) CrusDe that is described in the previous sections the first task is fulfilled. With regard to figure 2.2 a stage is reached where the simulator can be used to conduct experiments. However, when referring to section 2.5 apparently nothing is known about how the modeling results can be trusted. To build up trust in the newly implemented PSF the second task of this chapter, test and validation of CrusDe, is carried out in this section.

The testing of CrusDe can be subdivided into:

- *infrastructural tests* which refer to the management and interface layer, and
- *functional tests* concerning the plug-ins or the functional layer, respectively.

Problems that can arise in the infrastructure from programming mistakes are relatively simple and can be solved straightforwardly. Most of those errors are obvious when the interaction with the environment (experiment definition, databases)

and plug-in communication are examined. Examinations of these interactions were carried out for correct as well as a range of wrong user inputs (only then the case study in chapter 5 is possible). However, CrusDe could still end up in an undefined state and abort its service ungracefully when fed unspecified input.

The real problem comes with mistakes in the functional layer since as soon as results are obtained it is uncertain whether they are right or wrong. A simple test case was constructed and compared to an analytical result and results of a similar simulation model that implements the same load models and Green's functions for the elastic half-space, but performs the convolution in the original domain (*Grapenthin and Sigmundsson*, 2006).

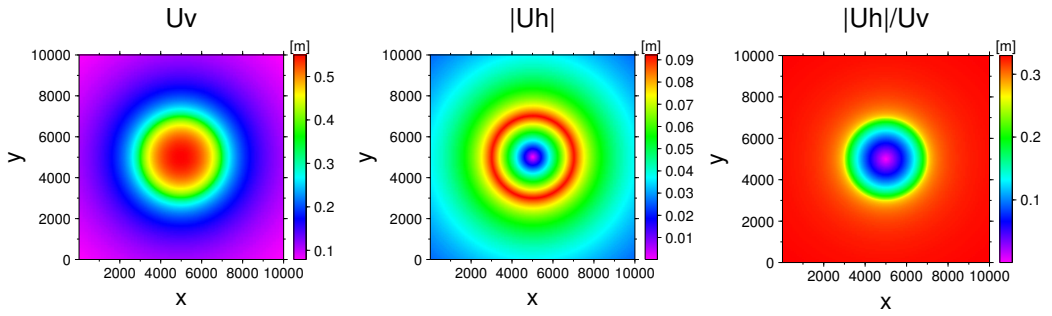
As a test case the response of an elastic half-space to a disk load is examined. Maximum vertical and minimum horizontal displacement are obtained under the center of a disk. The results depicted in figures 4.6(a) and 4.6(b) correspond to this. An analytic solution for the vertical displacement at this point can be derived from equation 3.5 as given by *Geirsson et al.* (2006):

$$U_{v,center} = 2\,\rho\,h\,R_0\,g\,\frac{1-\nu^2}{E} \tag{4.2}$$

where $U_{v,center}$ is the vertical displacement under the center of a disk, ρ is the density of the load, h is the loads height, R_0 is the radius of the disk, g is the acceleration due to gravity, and ν and E are, respectively, Poisson's ratio and Young's modulus.

Table 4.1 shows the results for a disk load with the parameters height $h = 150\,m$, radius $R_0 = 2\,km$, and density $\rho = 1000\,kg\,m^{-3}$ applied to an elastic half-space with a Young's modulus of $E = 10\,GPa$ and a Poisson's ratio of $\nu = 0.25$.

Table 4.1 shows that CrusDe misses the analytical solution by $2.2\,mm$ in the vertical and $0.1\,mm$ in the horizontal displacement. The differences to the reference implementation by *Grapenthin and Sigmundsson* (2006) can be explained by different techniques for the convolution. With respect to the difference in vertical displacement table 2.1 in *Grapenthin and Sigmundsson* (2006) is referenced. There the correlation between the spatial discretization and the error of the numerical solution is shown. This is analog for the horizontal displacement. The impact of



(a) vertical (b) horizontal (c) ratio

Figure 4.6: Response of an elastic half-space to a disk load. The Young's modulus and Poisson's ratio of the half-space are set to $10\,GPa$ and 0.25, respectively. The disks parameters are set to height $h = 150\,m$, radius $R_0 = 2\,km$, and density $\rho = 1000\,kg\,m^{-3}$. Its center is at $x = 5000\,m$ and $y = 5000\,m$.

spatial discretization when using the fast convolution is obvious from CrusDe's non-zero horizontal displacement in table 4.1 which will only be zero for infinitely small grid sizes, e.g. dS approaches 0. However, comparing the horizontal displacement shown in figure 4.6(b) to the solution of *Grapenthin and Sigmundsson* (2006) identical patterns are observed. The horizontal displacement is maximum in the area of the disks border and approaches zero under its center which is obvious from vector arithmetics.

Conforming to *Pinel et al.* (2007) the ratio between horizontal and vertical

	analytical solution	CrusDe's solution	reference implementation[a]
$U_{v,center}$ [m]	0.5518[b]	0.5496	0.5500
$\lvert U_{h,center}\rvert$ [m]	0.0	0.0001	0
$\lvert U_h\rvert / U_z$	$\leq 1/3$[c]	$\leq 1/3$[d]	$\leq 1/3$

Table 4.1: Comparison of CrusDe's solution for the response under the center of a disc load to a reference implementation and an analytical solution. The disk is characterized by height $h = 150\,m$, radius $R_0 = 2\,km$, and density $\rho = 1000\,kg\,m^{-3}$ and its center is at $x = y = 5000\,m$. The elastic half-space has a Young's modulus of $E = 10\,GPa$ and a Poisson's ratio of $\nu = 0.25$. The grid size is $10 \times 10\,m$ for a $10000 \times 10000\,m$ region of interest.

[a] see *Grapenthin and Sigmundsson* (2006)

[b] from equation 4.2

[c] Note that this must be true for the whole modeled area, not just the center, see *Pinel et al.* (2007)

[d] see figure 4.6(c)

displacements does not exceed 1/3 for an elastic half-space with $\nu = 0.25$ as depicted in figure 4.6(c) (see table 4.1).

Simulations that were run with different experiment definitions utilizing the irregular load and sinusoidal load history plug-ins (see appendix E.2.3) produce results that conform to *Grapenthin et al.* (2006). Results obtained by *Pinel et al.* (2007) could be reproduced using the thick plate plug-in for the Green's function. As stated by *Pinel et al.* (2007) it is observed that with increasing plate thickness the response of the thick plate model equals the response of the elastic half-space.

These observations testify that:

- the infrastructure does indeed work,
- the fast convolution plug-in produces results with a tolerable error (with respect to measurement errors in the *mm*-range) compared to the analytical solution and the reference implementation,
- the Green's functions for the elastic half-space (equations 3.3 and 3.4) and the thick plate (equations 3.7 and 3.8) are implemented correctly,
- the load related plug-ins (disk load, irregular load, sinusoidal load history (see appendix E.2.3)) are implemented correctly, and
- the post-processors (coordinate conversion and ratio plug-in (see appendix E.2.4)) and the result handler (table writer, netCDF writer (see appendix E.2.5)) are also implemented correctly.

As for model validity, not much more than application validity (see section 2.5) can be attested at this point. The satisfaction of the analysis objective is inherent to the formal models presented in section 3.5. Therefore, no further proof for this claim would be necessary. Anyway, the Green's functions have been used in previous studies where model results are confirmed by GPS measurements (*Pinel et al.*, 2007; *Grapenthin et al.*, 2006) which should build up additional trust.

To attest behavior validity future studies have to show how well predictions fit real data. An example that presents model forecasts which await confirmation by measurements is the study by *Ófeigsson et al.* (2006). They examine subsidence due to the new water reservoir Hálslón in the east of Iceland. Structural validity, however, is not possible at all. The formal model results are mere surface displacements; no activity within the lithosphere that would be represented by model states is included.

It must, however, be said that model validity is mainly inherent to the (Green's function) plug-ins and can change with each simulation depending on the participating plug-ins. Therefore, nothing absolute can be said when stating about model validity in connection with CrusDe. The framework merely provides an infrastructure for 'more or less' valid simulation models.

4.6 Evaluation of the plug-in based simulation framework

The implemented PSF provides the full infrastructure necessary to add and remove plug-ins and have them communicate via defined framework interfaces. A plug-in can rely on the framework which guarantees that fellow plug-ins will exist to provide required data. For each of the defined categories exists at least one reference implementation. Composed via an experiment definition, the plug-ins implement Green's method as described in this work. The implementation of the reference plug-ins follows standards and conventions (see section 4.3.1 and appendix E.2) for data storage. Yet, integration of plug-ins written in programming languages other than C, especially FORTRAN, was not studied.

Although CrusDe is in a productive state, several difficulties exist:

- **main memory:** The current implementation of the fast convolution consumes much main memory since both operands and the result are stored in arrays about four times the size of the modeled area (see figure 3.7). This sets a lower limit to the grid size depending on the available main memory of the simulator. This problem can be solved by implementing, for example, a two-dimensional

overlap-add method (e.g, *Smith* (1997)) in which small chunks of the load are subsequently convolved with a Green's function. However, the simulator used for this thesis has a total of 512 MB RAM and did not run into problems with experiment definitions that required as many as 1000×1000 grid points.

- **error messages:** As mentioned above, CrusDe is tested for several, also erroneous inputs, but cases exist where it will refuse to work and exit without any useful hints regarding the problem.

- **plug-in categories:** Currently the framework API implements the category distinctions for additional plug-in requests by a mere naming convention which a plug-in developer can easily violate – accidently or on purpose. Since the plug-ins are functionally without any intersection, doing this would not be of much use. However, during registration of a plug-in with CrusDe a comparison of its category to the categories of all plug-ins it requests could solve this problem.

- **experiment manager:** The experiment management is currently the weakest component of CrusDe since it is not much but a mere recorder and viewer for past experiments. Useful functions, such as search and retrieval of old experiment definitions are missing. Furthermore, the experiment manager should not be as deeply woven into the architecture as it is right now. The definition of an experiment management interface for third-party solutions would be of good use since a number of such systems exists. However, the evaluated systems (e.g., *Hawick and James* (2004)) were designed to manage much bigger simulation models and are not included for this reason. Most experiment management systems also require the user to install database management systems such as MySQL. This collides with the objective to require as little functionality as possible from the pre-existing working environment.

- **parameter units & ranges** CrusDe does not support the evaluation of physical units for plug-in parameters. Although the SI-system is widely used in science, other systems of units exist which may cause problems. The result

units might be clear from the original publications of the formal models or the plug-in documentation, but the modeling results remain dimensionless which is problematic when they are to be stored independent from CrusDe. Furthermore, the definition and evaluation of parameter ranges would be useful to avoid trouble by, e.g., errors in input magnitudes.

4.7 Summary

The scheme (figure 2.2, see section 2.3) that outlines the stages from an original system to model results obtained from a simulation was followed to the very end in this chapter by implementing a plug-in based simulation framework, presenting tests, and a validation attempt. The suggestion of *Overstreet et al.* (2002) (see section 2.4) to provide tools for modelers' assistance rather than aiming for fully automated composition was followed throughout this chapter.

Based on the formal model, Green's method, that was introduced in section 3.4 the author specified a simulation framework. After re-defining the term 'plug-in' on the basis of a software component the frameworks specification was transformed into a layered software architecture for a plug-in based simulation framework. The proposed architecture is mainly influenced by the idea of composable simulation models as introduced in section 2.4. The management, interface, and functional layers were described followed by a description of the implemented PSF which is called CrusDe. Several runtime scenarios in the form of sequence diagrams illustrated the way CrusDe operates. A short description of the performed tests that utilize the fast convolution as described in section 3.6 was given. The model was placed on the lowest position in the validity hierarchy introduced in section 2.5. However, a brief discussion showed that the plug-ins determine the validity of a composed simulation model. A brief evaluation of CrusDe showed that it is in a productive state. Improvements are, however, possible.

5 Case study: The Hekla 2000 lava

"If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them."
(Henry David Thoreau)

The purpose of this chapter is to place the developed PSF in the context of active research that is concerned with understanding of deformation of the Earth's crust. Therefore, CrusDe is utilized to simulate both the instantaneous and the final relaxed response to the (preliminary) lava flows that formed in the year 2000 during the eruption to the Icelandic volcano Mt. Hekla. Following a general introduction of the volcano and a brief motivation of this study in section 5.1, the specifics of the study site are presented in section 5.2. The model results and a comparison to estimates obtained from a model that relates subsidence to magma chamber deflation are presented in section 5.3. Conclusions from this case study are drawn in section 5.4.

5.1 Introduction

The Hekla volcano is located in the south-west of Iceland about $100\,km$ east of the capital Reykjavík (see figure 5.1). Being one of the most active volcanoes in Iceland, it erupted 18 times during the last 11 centuries. From the 1970s on the frequency of eruptions increased from about two eruptions each century to about one

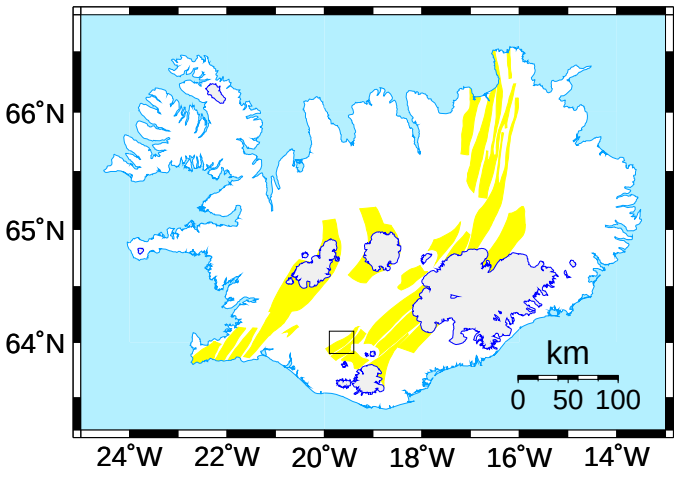


Figure 5.1: Map of Iceland with the study site marked by the black rectangle (see figure 5.3 for blow-up)

every 10 years. In February 2000 the latest eruption covered an area of $17.93\,km^2$ with $0.189\,km^3$ lava (*Höskuldsson et al.*, 2007). Assuming an average density of $2900\,kg\,m^{-3}$ for basalt this eruption product weights about $54.81\,10^{10}\,kg$ which is a considerable load being applied to the Earth's surface in this area. The eruptions of 1970, 1980-81, and 1990 added similar lava flows to the volcanic edifice.

Many volcanoes in the world show a more or less regular pattern in eruption frequency over a certain time. Extrapolation of future eruption cycles based on past behavior does not work well since behavior might change suddenly. Hekla is an example where the time between eruptions shortened significantly from approximately 50 to 10 years. The opposite seems to happen, for instance, at the Katla volcano about $60\,km$ south-east of Hekla. Until 1918 major eruptions at Katla occurred in a cycle of about 50 years which changed for an unknown reason. An obvious product of long term volcanic activity is the volcano edifice itself which applies a significant load to the lithosphere and might even affect its own behavior due stress changes induced in the ground.

As a step towards understanding the lithospheric structure around Hekla, surface displacements due to the (preliminary) February 2000 lava can be estimated using the plug-in based simulation framework developed in this thesis. The com-

parison of model results to observations might reveal information about the characteristics of the lithosphere and confirm areas suspected of irregular behavior (e.g., fractures) since these are not included in the model.

The interferogram in figure 5.2 shows an example observation of deformation that took place in a period of about 4 years between 08/1993 and 06/1997 with a maximum displacement of approximately 3 cm at the south-west and northern edge of the volcano. *Pagli* (2006) describes an interferogram as a product of satellite radar interferometry (InSAR), a geodetic technique to measure deformation. The basic functioning is that synthetic aperture radar (SAR) satellites transmit a radio signal to the Earth and measure the distance from the antenna to the ground and back. They preserve amplitude and phase of the returning signal for each recorded 'pixel'. The phase difference between two SAR images that are acquired at different times, but from about the same position in space is a measure of the change in distance between the ground and the radar antenna. With changes in satellite positions removed the phase difference can be attributed to deformation of the ground that

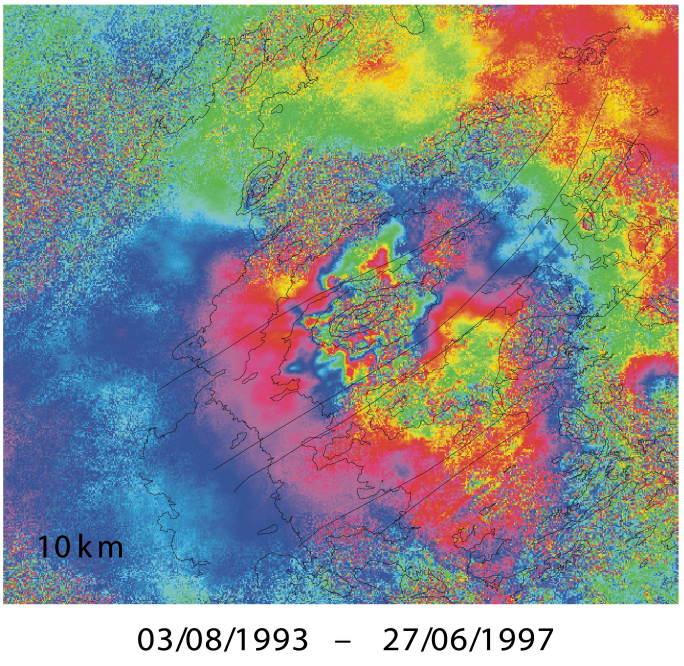


Figure 5.2: Interferogram showing subsidence at Hekla between 03/08/1993 and 27/06/1997. One color fringe represents about 3 cm of vertical displacement seen at south-western and northern edge of the volcano edifice (courtesy of Rikke Pedersen, Nordic Volcanological Center).

occurred between the acquisition of the two SAR images. Displacements as small as $3\,mm$ are detectable. They show up in the interferogram as so-called 'fringes'; a color band in which no color repeats. One fringe represents about $3\,cm$ displacement (see figure 5.2).

5.2 The study site: Experiment definition

Figure 5.3 shows a blow-up of the area enclosed by the rectangle in figure 5.1. The preliminary outlines of the 4 lava flows of the Hekla 2000 eruption and the topography of the area around the volcano are depicted.

The outlines of the lava flows describe the area of the load which is composed of an array of unit point masses in a $100 \times 100\,m$ grid. The loads density is assumed to be uniform over the area with a value of $\rho = 2900\,kg\,m^{-3}$; a typical value for fresh basalt. The average heights of the flows differ significantly and are given in table 5.1.

Results are obtained for both the instantaneous and the final relaxed response utilizing the Green's functions given by equations 3.3, 3.4 and 3.7, 3.8, respectively.

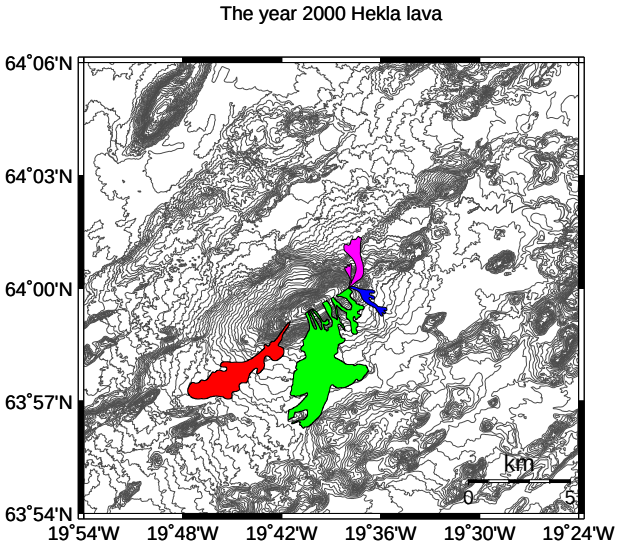


Figure 5.3: Map showing the topography of the study site and the preliminary Hekla 2000 lava flows — pink: north, blue: north-south, green: south, red: south-west (compare to table 5.1).

The elastic parameters of the lithosphere are set to $E = 40\,GPa$ for the Young's modulus (*Grapenthin et al.*, 2006) and $\nu = 0.25$ for the Poisson's ratio. The acceleration due to gravity of the Earth is $g = 9.81\,m\,s^{-2}$. The parameters for the thick plate model are with $H = 5\,km$ for the plate thickness and $\rho_f = 3100\,kg\,m^{-3}$ for the upper mantle density the same as used by *Pinel et al.* (2007) who studied the nearby Katla volcano. Listing 5.1 shows the complete experiment definition utilized to simulate the final relaxed response of the surface to the Hekla 2000 lava. The experiment definition for the instantaneous response is the same; only the Green's function is changed to '`elastic halfspace (pinel)`'.

The model result obtained by a fast convolution is post-processed by adding horizontal radial displacement using the '`xy2r`' plug-in. Furthermore the ratio between horizontal and vertical displacement $|U_h|/U_v$ is added to the result array using the '`hori2vert-ratio`' plug-in.

5.3 Model results

5.3.1 Instantaneous and final relaxed deformation due to the Hekla 2000 lava

The results obtained from the two experiments are presented in figure 5.4 with figure 5.4(a) – 5.4(c) showing the instantaneous and figure 5.4(d) – 5.4(f) showing the final relaxed responses, respectively. An estimate of the time necessary for the final relaxed response to be established would be on the order of tens to hundreds of years (*Pinel et al.*, 2007).

Lava field	Mean thickness (m)	Fraction of total area (%)
North	3	3.6
North-south	3	7.3
South	12	61.6
South-west	10	27.9
Mean	9.82	100

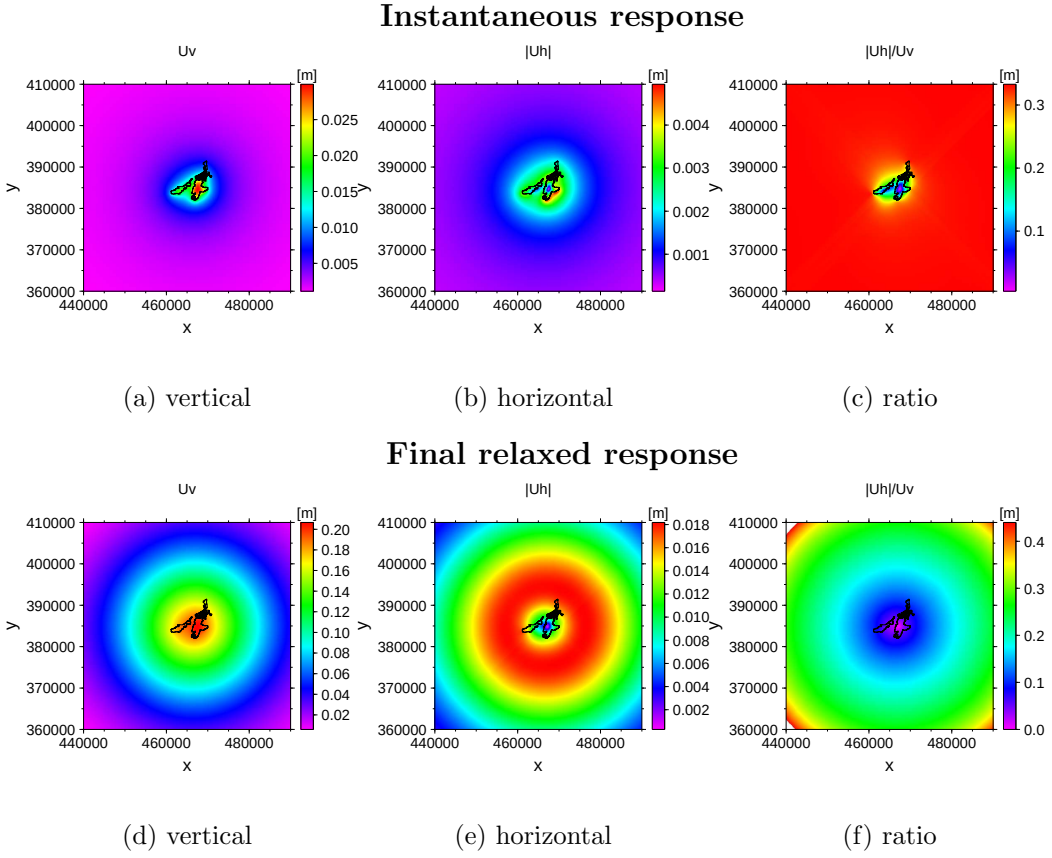
Table 5.1: Characteristics of the Hekla 2000 lava flows (*Höskuldsson et al.*, 2007).

```
<?xml version="1.0" encoding="UTF-8"?>

<experiment name="hekla 2000 lava">

  <file name="load" value="./hekla_load.100.xyz" />
  <file name="result" value="./result_thickplate.nc" />

  <region name="west" value="420000" />
  <region name="east" value="520000" />
  <region name="south" value="320000" />
  <region name="north" value="420000" />

  <parameter name="gridsize" value="100" />

  <greens_function>
    <plugin name="thick plate (pinel)"/>
    <parameter name="g" value="9.81" />   <!-- acc. gravity [m/s^2] -->
    <parameter name="nu" value="0.25" /> <!--  poissons ratio -->
    <parameter name="rho_f" value="3100" /> <!-- density fluid [kg/m^3] -->
    <parameter name="H" value="5000" />   <!-- plate thickness [m] -->
    <parameter name="E" value="40"/> <!-- youngs modulus [GPa] -->
 </greens_function>

  <load_function>
    <plugin name="irregular load" />
    <parameter name="rho" value="2900" /> <!-- lava desity-->
  </load_function>

  <postprocessor>
    <plugin name="xy2r" />
    <plugin name="hori2vert-ratio" />
  </postprocessor>

  <output> <plugin name="netcdf writer" /> </output>

  <kernel> <plugin name="fast 2d convolution" /> </kernel>

</experiment>
```

Listing 5.1: Hekla 2000 experiment definition

Unfortunately, at the time of this study no real deformation data has been available to compare the modeled displacements to observations. However, there is something striking about the almost circular final relaxed response. The regularity is surprising given the irregularity of the input load which is still preserved in the instantaneous response (though already low-pass filtered). Only small irregularities are visible at the tip of the south-western lava flow for the vertical response and under the center of gravity for the horizontal response.

The pattern emerging in figures 5.4(d–e)is interesting since it can easily be mistaken with the response to a deflating shallow magma chamber. Magma chamber pressure changes are often modeled as a point source of pressure in an elastic half-space; the so-called 'Mogi model' (*Mogi*, 1958). The surface deformation in response to a Mogi source is radially symmetric which accounts for the possibility of a mix-up.



(a) vertical (b) horizontal (c) ratio

(d) vertical (e) horizontal (f) ratio

Figure 5.4: **a-c)** Instantaneous response of an elastic half-space characterized by a Young's modulus, $E = 40\,GPa$ and a Poisson's ratio $\nu = 0.25$ to the Hekla 2000 lava. **d-e)** Final relaxed response of a thick plate ($H = 5\,km$, $E = 40\,GPa$, $\nu = 0.25$) over an inviscid fluid ($\rho = 3100\,kg/m{-}3$) to the Hekla 2000 lava

```
<load_function>
  <plugin name="disk load" />
  <parameter name="height" value="9.82"/>
  <parameter name="radius" value="2514" />
  <parameter name="center_x" value="467400" />
  <parameter name="center_y" value="384700" />
  <parameter name="rho" value="2900" />
</load_function>
```

Listing 5.2: Disk load experiment definition excerpt. Changes to the original listing 5.1 are underlined.

In the following the final relaxed response of a disk is compared to the response induced by a point source defined by the Mogi model. The Mogi model is introduced in the following section 5.3.2. The disk is modeled with an uniform height of $h = 9.82\,m$ which is the weighted mean of the lava flow thicknesses (see table 5.1) and a radius $r = 2514\,m$ which is obtained from the lava flow volume $V = 0.195\,km^3$ (calculated from gridded load). Listing 5.2 displays the respective changes of the original experiment definition (listing 5.1).

5.3.2 Modeling a magma chamber: The Mogi model

The response to a point source in an elastic half-space with Poisson's ratio $\nu = 0.25$ in cylindrical coordinates is (*Sigmundsson*, 2006):

$$U_v = C\,\frac{d}{(d^2+r^2)^{\frac{3}{2}}} \tag{5.1}$$

$$U_h = C\,\frac{r}{(d^2+r^2)^{\frac{3}{2}}} \tag{5.2}$$

where d is the depth of the point source, r is the horizontal distance from the source, and C is the source strength. Maximum uplift occurs directly above the source. Thus, when setting $r = 0$ in equation 5.1 the source strength is (*Sigmundsson*, 2006):

$$C = h_0\,d^2 \tag{5.3}$$

where h_0 is maximum vertical displacement.

The volume of the surface change due to pressure change of the point source is given by *Sigmundsson* (2006):

$$\Delta V_{edifice} \quad = \quad 2\,\pi\,C \tag{5.4}$$

Obvious from the above equations, 4 free parameters must be fixed when using the Mogi model: latitude, longitude, source depth, and source strength. Latitude and longitude are the coordinates of the maximum final relaxed vertical displacement (figure 5.4(d)) which is at $x = 467400\,m$ (longitude) and $y = 383700\,m$ (latitude). The maximum height, $h_0 = 21.42\,cm$, is the maximum vertical response to a disk load (see figure 5.5(a)). To obtain the source depth equation 5.3 is inserted in equation 5.4 and solved for d:

$$d \quad = \quad \sqrt{\frac{\Delta V_{edifice}}{2\,\pi\,h_0}} \tag{5.5}$$

Using the volume of the lava flow, V, as volume of surface change, $\Delta V_{edifice}$, the source depth is at $d \approx 12\,km$. Using this in equation 5.3 gives a source strength of $C = 0.031\,km^3$.

5.3.3 Response of a deflating magma chamber vs. final relaxed response to a disk load

Since all free parameters of the Mogi model are fixed to reasonable values, the surface displacements can be calculated for the area which is shown in figure 5.4. The model results for both the final relaxed response to a disk load and the response to a deflating magma chamber are shown in figure 5.5 (z-axis being directed downwards).

The first two rows in figure 5.5 show vertical displacements due to the disk, the point source deflation, and the residual ($R_v = U_{v,disk} - U_{v,mogi}$). The first row uses the same color scale for the three images, whereas the second row presents each image with an individual scale. The vertical residual shows good correlation between the responses directly underneath the area covered by the disk. Farther

away from the source the differences increase up to 1/6 of underestimation by the Mogi response.

The last two rows in figure 5.5 show horizontal displacement with the same scale settings as for the vertical response. The third row shows a huge difference in the horizontal response which is to be quantified from the scales in the fourth row.

5.4 Discussion & conclusions

Since the model results are obtained with different underlying Earth models (Mogi model: elastic half-space, disk load response: thick elastic plate / viscoelastic[1]), a comparison of their results might appear like comparing apples and oranges. This is only to some extent true. It is important to refer to the respective time scale.

A sudden eruption that lasts only a few weeks is very likely to cause only elastic deformation due to pressure reduction in the magma chamber and due to the lava flows. In this case the results obtained from simulations with the Mogi model can be compared to the simulated instantaneous elastic response of the Earth induced by the lava flows. Since at least the vertical elastic displacement due to a lava flow poses a significant signal source, great care is necessary when GPS signals are analyzed recorded in the vicinity of fresh lava flows; especially when source depths shall be inferred from such data. If not avoidable at all, such surveys should be conducted at a point in the region of interest that is farthest away from the center of gravity of the fresh lava flow (which equals, for instance, the location of highest vertical displacement in figure 5.4(a)).

When studying subsidence due to a volcanic eruption on time scales on the order of years the comparison in figure 5.5 shows that the response to surface loading and a deflating magma chamber might show the same circular pattern. The resulting subsidence, however, differs significantly. Due to the low-pass characteristics of the

[1] A viscoelastic substance has an elastic component and a viscous component. The viscosity of a viscoelastic substance gives the substance a strain rate, i.e. rate of deformation, dependent on time (`http://en.wikipedia.org/wiki/Viscoelasticity`, 19.07.07).

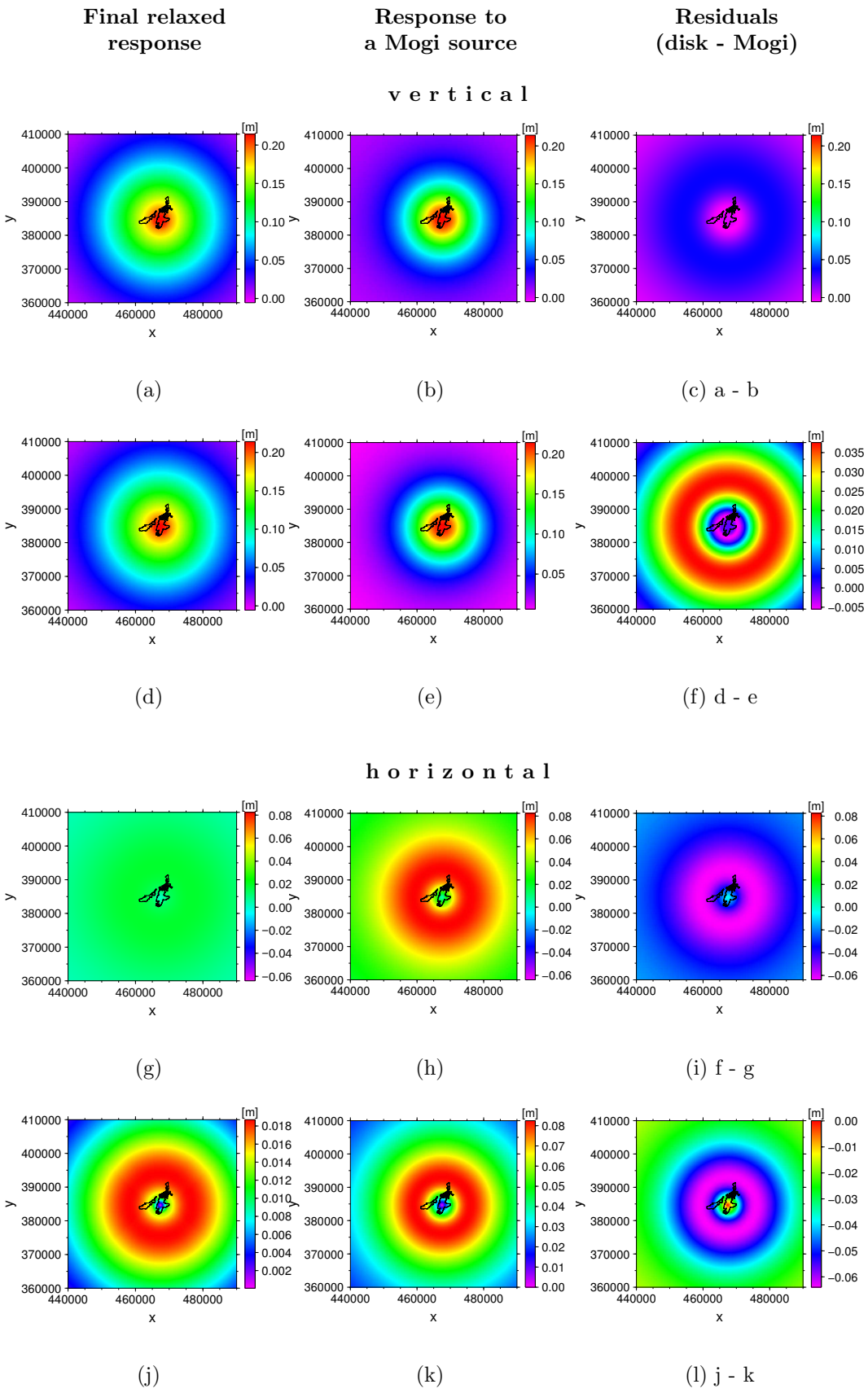


Figure 5.5: Disk load response (left column), Mogi source response (center column), and residuals (right column), **a-f)** vertical response, **g-j)** horizontal response, the lava outlines are included for orientation

lithosphere (see section 3.3) and the observations in figure 5.4, it can be concluded that the initially radially symmetric response to a point source would not change its shape over time, only increase in amplitude. Therefore, the comparison in figure 5.5 holds. It is obvious from figure 5.5 that gradual subsidence due to the load on the surface can superimpose or even mimic the signals of a Mogi type magma chamber.

This comparison is even more allowable when a response to sudden inflation or deflation of a magma chamber might be superimposed by ongoing subsidence due to 'old' surface loading. This is would involve both an instantaneous elastic and a gradual viscoelastic response which is exactly what figure 5.5 compares. Thus, if surface loading is ignored when inferring deformation sources and the deformation is attributed to a deflating magma chamber, moderate errors in the vertical and significant errors in the horizontal displacements are to be expected.

Seismic studies conducted by *Soosalu and Einarsson* (2004) do not support the existence of a magma chamber between $4\,km$ to $14\,km$ underneath the Hekla volcano which makes the assumption of a point source at $12\,km$ highly questionable. Thus, when too much, e.g. all, of the deformation signal is attributed to magma chamber deflation the source will be estimated at too shallow depth which is clear from equation 5.5 (assuming constant volume). The source depth d is inversely correlated to the square root of the maximum vertical displacement. Thus, especially when the estimated maximum vertical displacement is 'small', e.g. $h_0 < 1\,m$, small variations of h_0 will have a significant effect on source depth variations.

A suggestion that can be inferred from the simulations conducted in this chapter is that if magma chamber constraints are to be estimated directly after an eruption or during the time span of ongoing, expected viscoelastic response induced by eruption products at the surface, care is required when applying the Mogi model (the same might hold for viscoelastic responses to glacial dynamics if a volcano is covered by an ice cap). Additional factors posed by the surface load must be considered and constrained by careful measurements and interpretations of observations and eruption histories to correct the recorded data for such signal sources.

A good indicator whether observed displacement patterns are the (elastic)

response to a deflating magma chamber or (viscoelastic) subsidence of the crust due to loading might be the ratio between horizontal and vertical displacement.

Future studies must look deeper into that. A suggestion would be to simulate the ongoing subsidence at Hekla (convolution with time (*Pinel et al.*, 2007)) in response to the lava flows (1970-2000) and compare the emerging pattern to observations.

5.5 Summary

Using the example of a recent Icelandic lava flow, it was demonstrated that the PSF is in a development state mature enough to actually aid answering questions in crustal deformation science. CrusDe's allowance for painless simulation model composition and straightforward parameterization, all within a clear experiment definition, fosters structured simulations. It could be shown that it is possible to adjust simulations at ease to questions thrown up by obtained results – provided the necessary functionality is already implemented as a plug-in compatible with CrusDe.

6 Summary, conclusions & outlook

"Everything is interaction."
(Alexander von Humboldt)

The plug-in based, discrete time- and space-stepped **Crus**tal **De**formation simulation framework CrusDe is developed in this thesis. A layered architecture consisting of management, interface, and functional layers is proposed to realize Green's method for the Earth-load-system as a composable simulation model. CrusDe implements this architecture in the C/C++ programming language for Linux environments.

The interaction between the architectural layers allows for a decomposition of Green's method into its elements Green's function, load function, and convolution operator. Each represents a model category that can hold a multitude of different implementations. These are treated as plug-ins in the sense that users select a particular implementation from each category to compose a simulation model. CrusDe (on the management layer) in turn plugs the user selections into the underlying simulation model (functional layer) for each specific simulation. Supplemental categories to the ones given above are defined by the framework and include the postprocessor and the result handler category. They allow, respectively, for user-defined alteration of a modeling result and arbitrary data storage formats.

Users can individually add new plug-ins to each category. The framework supports reuse of functionality provided by existing plug-ins for new implementations within the same category. In an experiment definition formulated in XML, users define the plug-ins and provide values for parameters to configure a particular simulation. An embedded experiment manager stores both experiment definition and

additional metadata in a XML database after a completed simulation.

To describe the stepwise transformation of the observed phenomenon – the displacement of the Earth's surface due to load changes – into a program architecture that can be implemented to execute simulations, a general sequence of techniques for finding solutions for simulation problems is utilized. The observed phenomenon is first identified as emerging from the Earth-load-system for which, due to the complexity of the lithosphere, a variety of informal models exist. To transform informal models into formal mathematical expressions the analogy of the lithosphere as a linear space-invariant (LSI) filter is used. This enables utilizing Green's method as a formal model since the LSI analogy fulfills the necessary preconditions (linearity) for its use. Based on this formal model a program architecture that utilizes the modularity of Green's method to realize user-driven composition of the simulation model is developed. Two existing Green's functions are provided as plug-ins which represent the lithosphere as an elastic half-space and a thick elastic plate over an inviscid fluid. Since these functions are not only linear but also space-invariant, a convolution operator plug-in could be provided that implements the fast convolution.

Trust is the model that CrusDe implements comes from previously conducted studies that utilized the same model to examine effects of glacial loading in Iceland on different scales. These studies confirm the model results with GPS measurements. Another study uses the model results to forecast subsidence due to a new water reservoir in the east of Iceland which awaits confirmation by GPS data.

As for trust in the implementation of the developed simulation framework, CrusDe's results for the simple test case of a disk load correspond to that of an analytical solution for displacements under the center of the disk. Furthermore, the results correspond to a reference implementation in which Green's method is executed in the original domain. Additionally, the results of the following studies could be reproduced:

- elastic and final relaxed response to a load at the Mýrdalsjökull ice cap conducted by *Pinel et al.* (2007),
- elastic response to annual load cycles (involves load history) at the four largest

Icelandic ice caps conducted by *Grapenthin et al.* (2006), and

- elastic response to minimum and maximum water levels of the new water reservoir Hálslón in the east of Iceland conducted by *Ófeigsson et al.* (2006).

Moreover, a case study conducted in this thesis utilizes CrusDe to examine the lithospheric response to the lava of the year 2000 eruption of the Icelandic volcano Mt. Hekla (see section 5.4 for conclusions drawn from this study).

Applicability of CrusDe to all these surveys underlines its flexibility. It is concluded that:

- CrusDe supports all types of load, since all loads can be expressed by height and (average) density,
- CrusDe supports all geographical settings when the provided instantaneous and final relaxed response are applicable or models of the lithosphere in the form of Green's functions exist,
- adjusting the simulation framework to new study sites requires only few settings in a XML file,
- the simulation framework is designed to ease user-driven extensions, and
- CrusDe tries its best to test the consistency of new plug-ins, but can by no means test the semantics a plug-in intends to provide.

Application scenarios for CrusDe in scientific work can be derived from its analysis objective (see section 3.1) and include:

- solving a particular loading problem (surface deformation) in the form of a quantitative estimation of anticipated effects,
- solving a particular inverse loading problem (surface deformation) by fitting results to real observations (e.g., GPS or InSAR data) and thus infer knowledge about a system element (e.g., parameters of the lithosphere), and

- assistance in planning of GPS and InSAR campaigns to locate deformation patterns that are either of interest to a measurement campaign or might interfere with the actual study.

During testing and the case study the following possible improvements and extentions for CrusDe surfaced:

- A convolution operator plug-in that supports convolution with time to allow for the simulation of a transition between instantaneous and final relaxed response.
- A convolution operator plug-in that supports the overlap-add method (*Smith*, 1997). This would enable more efficient use of memory and arbitrarily small grids.
- Support of plug-ins in FORTRAN is very desirable since much scientific programming uses this language. Adjustments on the interface layer would be necessary.
- Support of composite surface loads. This requires an additional load plug-in that calculates average densities of the unit point masses depending on densities and respective heights given for a particular point.
- Direct application of postprocessors to existing modeling results.
- An improved simulation management featuring searches, editing, and export of experiment definitions.
- Definition and evaluation of parameter units and ranges to assure a correct parameterization and a more convenient interpretation of the modeling results.
- Useful error messages for cases in which CrusDe fails.
- A GUI that aids the creation of an experiment definition (possible in any (domain specific) graphical language) and writes them to XML files.
- Green's functions should notify CrusDe whether they are space-invariant.

Direct comparisons of model results obtained using different Green's functions that formalize an identical informal Earth model will be possible at some point. Minding the respective boundary conditions of the Green's functions, a discussion about the impact of both complexity of the informal model *and* the respective formal model is possible when model results are compared among each other and to real data. In a sense this would liberate findings obtained by simulation from depending on a particular numerical expression.

In the long term some thought must be given to requirements that CrusDe has to fulfill to be compatible to multi-framework environments (e.g., *Argent and Rizzoli* (2004)). Due to the its specific functionality reuse of the full composable simulation model within a greater simulation context is an option. Of course, the including multi-framework must assure that CrusDe's analysis objective is met.

Bibliography

Argent, R. M., and A. E. Rizzoli (2004), Development of Multi-Framework Model Components, in *Integrated Assessment and Decision Support, Proceedings of the 2nd Biennial Meeting of the International Environmental Modelling and Software Society*, edited by Claudia Pahl-Wostl and Sonja Schmidt and Andrea E. Rizzoli and Anthony J. Jakeman, iEMSs.

Barletta, V. R., C. Ferrari, G. Diolaiuti, T. Carnielli, R. Sabadini, and C. Smiraglia (2006), Glacier shrinkage and modeled uplift of the Alps, *Geophys. Res. Lett.*, *33*, L14,307, doi:10.1029/2006GL026490.

Battle, D. J. (1999), Maximum Entropy Regularisation Applied to Ultrasonic Image Reconstruction, Ph.D. thesis, University of Sydney.

Böhme, H. (2007), Softwarekomponenten mit eODL und SDL für verteilte Systeme, Ph.D. thesis, Humboldt-University Berlin.

Bronstein, I., K. Semendjajew, G. Musiol, and H. Mühlig (2001), *Taschenbuch der Mathematik*, 5 ed., 1191 pp., Verlag Harri Deutsch.

Challis, L., and F. Sheard (2003), The Green of Green's Functions, *Physics Today*, *56*(12), 41–46.

Fischer, J., and K. Ahrens (1996), *Objektorientierte Prozeßsimulation in C++*, 360 pp., Addison-Wesley.

Francis, P., and C. Oppenheimer (2004), *Volcanoes*, 521 pp., Oxford University Press.

Frigo, M., and S. G. Johnson (2005), The Design and Implementation of FFTW3, *Proc. IEEE*, *93*(2), 216–231.

Fujimoto, R. M. (2000), *Parallel and Distributed Simulation Systems*, 320 pp., John Wiley & Sons, Inc.

Galassi, M., J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi (2007), *GNU Scientific Library – Reference Manual*, Free Software Foundation, 1.9 ed.

Geirsson, H., T. Árnadóttir, C. Völksen, W. Jiang, E. Sturkell, T. Villemin, P. Einarsson, F. Sigmundsson, and R. Stefánsson (2006), Current plate movements across the Mid-Atlantic Ridge determined from 5 years of continuous GPS measurements in Iceland, *J. Geophys. Res.*, *111*, B09,407, doi:10.1029/2005JB003717.

Grapenthin, R., and F. Sigmundsson (2006), The Green's functions technique in crustal deformation and its applications (WT), *Tech. Rep. 0602*, The Nordic Volcanological Center, Reykjavík, Iceland.

Grapenthin, R., F. Sigmundsson, H. Geirsson, T. Árnadóttir, and V. Pinel (2006), Icelandic rhythmics: Annual modulation of land elevation and plate spreading by snow load, *Geophys. Res. Lett.*, *33*, L24,305, doi:10.1029/2006GL028081.

Hawick, K., and H. James (2004), Distributed Scientific Simulation Data Management, *Tech. Rep. CSTN-008*, Institute of Information and Mathematical Sciences, Massey University.

Höskuldsson, A., N. Óskarsson, R. Pedersen, K. Grönvold, K. Vogfjörd, and R. Ólafsdóttir (2007), The millennium eruption of Hekla in February 2000, *Bull. Volcanol.*, doi:10.1007/s00445-007-0128-3.

Kasputis, S., and H. C. Ng (2000), Composable simulations, *Proceedings of the 2000 Winter Simulation Conference*, pp. 1577–1584.

Linux manual (2003), *Linux Programmer's Manual:dlopen(3)*.

Mayer, J., I. Melzer, and F. Schweiggert (2003), Lightweight plug-in-based application development, *Lecture Notes in Computer Science; Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002. Revised Papers*, *2591/2003*, 87–102.

Meffert, B., and O. Hochmuth (2004), *Werkzeuge der Signalverarbeitung*, 274 pp., Pearson Studium.

Mogi, K. (1958), Relations between eruptions of various volcanoes and the deformations of the ground surface around them., *Bull. Earthquake Res. Inst. Univ. Tokyo*, *36*, 99–134.

Ófeigsson, B., P. Einarsson, F. Sigmundsson, E. Sturkell, H. Ólafsson, R. Grapenthin, and H. Geirsson (2006), Expected Crustal Movements due to the Planned Halslón Reservoir in Iceland, in *Eos Trans. AGU , Fall Meet. Suppl.*, vol. 87(52), pp. Abstract T13A–0495.

Oreskes, N., K. Shrader-Frechette, and K. Belitz (1994), Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences, *Science*, *263*, 641–646.

Overstreet, C. M., R. E. Nance, and O. Balci (2002), Issues in Enhancing Model Reuse, in *International Conference on Grand Challenges for Modeling and Simulation, Jan. 27-31, San Antonio, Texas, USA*, San Antonio, Texas, USA.

Page, E., and J. Opper (1999), Observations on the Complexity of Composable Simulation, *Proceedings of the 1999 Winter Simulation Conference*, pp. 553–560.

Pagli, C. (2006), Crustal Deformation Associated with Volcano Processes in Central Iceland, 1992-2000, and Glacio-isostatic Deformation Around Vatnajökull, Observed by Space Geodesy, Ph.D. thesis, University of Iceland.

Paul, R. J., and S. J. Taylor (2002), What use is model reuse: is there a crook at the end of the rainbow?, *Proceedings of the 2002 Winter Simulation Conference*, pp. 648–652.

Pidd, M. (2002), Simulation software and model reuse: A polemic, *Proceedings of the 2002 Winter Simulation Conference*, pp. 772–775.

Pinel, V., F. Sigmundsson, E. Sturkell, H. Geirsson, P. Einarsson, M. T. Gudmundsson, and T. Högnadóttir (2007), Discriminating volcano deformation due to magma movements and variable surface loads: Application to Katla subglacial volcano, Iceland, *Geophys. J. Int.*, *169*(1), 325–338.

Press, F., and R. Siever (1978), *Earth*, 2nd ed., 649 pp., W.H. Freeman and company, San Francisco.

Rew, R., G. Davis, S. Emmerson, H. Davies, and E. Hartnett (2006), *The NetCDF Users Guide*, Unidata Program Center, University Corporation for Atmospheric Research, netCDF Version 3.6.1.

Sigmundsson, F. (2006), *Iceland Geodynamics, Crustal Deformation and Divergent Plate Tectonics*, 228 pp., Springer-Praxis, Chichester, UK.

Smith, S. W. (1997), *The Scientist and Engineer's Guide to Digital Signal Processing*, 626 pp., California Technical Publishing.

Snieder, R. (2004), *A Guided Tour of Mathematical Methods for the Physical Sciences*, 2nd ed., 507 pp., Cambridge University Press.

Soosalu, H., and P. Einarsson (2004), Seismic constraints on magma chambers at hekla and torfajökull volcanoes, iceland, *Bull. Volc.*, *66*(3), 276–286, doi:10.1007/s00445-003-0310-1.

Stearns, S. D., and D. R. Hush (1999), *Digitale Verarbeitung analoger Signale*, 7th ed., 571 pp., R. Oldenburg.

Störrle, H. (2005), *UML 2 für Studenten*, 320 pp., Pearson Studium.

Thornton, S. T., and J. B. Marion (2003), *Classical Dynamics of Particles and Systems*, 5th ed., 672 pp., Brooks Cole.

Turcotte, D. L., and G. Schubert (2002), *Geodynamics*, 2nd ed., 528 pp., Cambridge University Press.

van Dam, T., H.-P. Plag, O. Francis, and P. Gegout (2002), GGFC Special Bureau for Loading: Current Status and Plans, in *Proceedings of the IERS Workshop on Combination Research and Global Geophysical Fluids, IERS Technical Note No. 30*, pp. 180–198.

Vandevoorde, D. (2006), Plugins in C++, *Tech. rep.*

Wagenbreth, O., and W. Steiner (1990), *Geologische Streifzüge, Landschaft und Erdgeschichte zwischen Kap Arkona und Fichtelberg*, 4th ed., 204 pp., Deutscher Verlag für Grundstoffindustrie, Leipzig.

Watts, A. B. (2001), *Isostasy and Flexure of the Lithosphere*, 478 pp., Cambridge University Press.

Wikipedia contributors (2007), Wikipedia: Unified modelling language, online (last checked: 07/2007).

Winnel, A., and J. Ladbrook (2003), Towards composable simulation: Supporting the design of engine assembly lines, in *17th European Simulation Multiconference ESM2003, June 9-11, 2003 Nottingham, UK Foundations for Successful Modelling & Simulation*, edited by D. Al-Dabass.

Zeigler, B. P., H. Praehofer, and T. G. Kim (2000), *Theory of Modeling and Simulation*, 2nd ed., 510 pp., Academic Press.

A Symbols

Symbol	Quantity	SI unit (where applicable)
A	coefficient for thick plate model	
B	coefficient for thick plate model	
C	coefficient for thick plate model	
C	source strength (sec 5.3.2)	m^3
d	source depth	m
D	coefficient for thick plate model	
e^x	exponential function	
E	Young's modulus	Pa
$f(t)$	continuous time signal	
$f(t_n)$	discrete time signal	
$f_{x,y}$	discrete, two-dimensional input signal (filter)	
F_m	spectral coefficients of a DFT	
g	acceleration due to gravity	$m\,s^{-2}$
$g_{x,y}$	discrete, two-dimensional output signal (filter)	
G	Green's function	$m\,kg{-}1$
$G(\vec{r},\vec{r'})$	Green's function that determines displacement at point $\vec{r}$ in response to a load at point $\vec{r'}$	$m\,kg{-}1$
$G_h(r)$	Green's function that determines horizontal displacement at a point r	$m\,kg{-}1$
$G_h^H(r)$	Green's function that determines horizontal displacement at a point r dependent on a plate thickness H	$m\,kg{-}1$
$G_v(r)$	Green's function that determines vertical displacement at a point r	$m\,kg{-}1$
$G_v^H(r)$	Green's function that determines vertical displacement at a point r dependent on a plate thickness H	$m\,kg{-}1$
$G_{x,y}$	Green's function that determines displacement at point (x,y)	$m\,kg{-}1$
h	height	m
h_0	maximal vertical displacement	m

$h(\vec{r'})$	height at point $\vec{r'}$	m
H	plate thickness	m
J_0	Bessel function of zeroth order	
J_1	Bessel function of first order	
L	load, load function	kg
L_n	n-th value of a discrete (1D) signal	
$L(\vec{r'})$	unit point mass at point $\vec{r'}$	kg
$L_{x,y}$	unit point mass at point (x, y)	kg
$\mathfrak{L}_m$	m-th spectral component of a signal	
N	number of elements in a discrete signal	m
r	horizontal distance	m
R	area, region of interest	m^2
R_0	radius of a disk	m
R_h	residual of horizontal displacement	m
R_v	residual of vertical displacement	m
R_x	width (of region of interest)	m
R_y	length (of region of interest)	m
$\vec{r}, \vec{r'}$	point in cylindrical coordinates	
t	time	s
$U(\vec{r'})$	directionally unspecified displacement at point $\vec{r'}$	m
$U_{x,y}$	directionally unspecified displacement at point (x, y)	m
U_h	horizontal displacement	m
$U_h(\vec{r'})$	horizontal displacement at point $\vec{r'}$	m
$U_h^H(\vec{r'})$	horizontal displacement at point $\vec{r'}$ depending on plate thickness H	m
U_v	vertical displacement	m
$U_{v,center}$	vertical displacement under the center of a disk	m
$U_{v,disk}$	vertical displacement due to a disk load	m
$U_{v,mogi}$	vertical displacement due Mogi source	m
$U_v(\vec{r'})$	vertical displacement at point $\vec{r'}$	m
$U_v^H(\vec{r'})$	vertical displacement at point $\vec{r'}$ depending on plate thickness H	m
V	volume (of a lava flow)	m^3
(x, y)	Cartesian coordinates of a point	
$\Delta V_{edifice}$	volume of surface change	m^3
ν	Poisson's ratio	—
ρ	density	$kg\,m^{-3}$
$\rho(\vec{r'})$	density at point $\vec{r'}$	$kg\,m^{-3}$

B Contents of the CD

```
.
|-- crusde                  - CrusDe source code, incl. Makefile
|   |-- doc_html            - HTML documentation
|   |-- experiment-db       - home directory of experiment database
|   |-- plugin_src          - plug-in source code, Makefiles in subdirs
|   |   |-- data_handler    - sources of result handlers
|   |   |-- green           - sources of Green's functions
|   |   |-- load            - sources of load functions
|   |   |-- load_history    - sources of load history functions
|   |   |-- operator        - sources of convolution operator
|   |   +-- postprocess     - sources of post-processors
|   +-- plugins             - plug-in repository, subdirectories as
|                             in 'plugin_src', plug-ins are installed
|                             to this folder
|-- examples                - sample experiments definitions
|   |                         and GMT scripts ...
|   |-- disk                - for a disk case
|   +-- hekla               - for case study (load + mogi data)
|-- gmt                     - mapping data (used in examples)
|   +-- hekla               - ... for hekla
|-- images                  - figures of the thesis
|-- latex                   - latex sources of the thesis
|-- libs                    - libraries needed for CrusDe
|   |-- fftw                - install FFTW here (see contents)
|   |-- netcdf              - install netCDF here (see contents)
|   +-- xerces-c            - install Xerces-C here (see contents)
|-- listings                - source code listings used in thesis
+-- thesis_grapenthin.pdf   - digital version of this document
```

C Installation and simulation

To install CrusDe the complete `crusde` directory of the enclosed CD must be copied to a position in the file system that is writable (all subdirectories and files of must be changed to writable after being copied from the CD). Before the simulation framework can be compiled and linked, the libraries in the `libs` directory must be installed to the prepared directories. The instructions of the particular library (included in the prepared directory) are to be followed for installation. Furthermore, the Qt-library (`http://www.trolltech.com`) must be downloaded and installed (CrusDe is tested for version 4.1.1). The `Makefile` in the `crusde` directory expects Qt to be installed at `/usr/local/Trolltech/Qt` (change the variable `QT_DIR` in the `Makefile` if another place is preferred).

Changing into the `crusde` directory and executing: `$> make all` will compile the sources and link them into the binary `crusde`. This will work only if all libraries are installed correctly. All provided plug-ins are created in the `plugin_src` directory.

Before any of the examples in the `examples/*` directories can be simulated, every plug-in to be used with the framework must be registered. Appendix D.2 explains this step in detail. Additionally, the environment variable `CRUSDE_HOME` must be set to the path where the framework is installed.

Changing, for instance, to the `examples/hekla` directory and invoking:

```
$> ../../crusde/crusde hekla_disk_elastic.xml
```

will run an example simulation. The output is created in a file as specified in the experiment definition (path must be given there as well). The installation of `ncview` and the Generic Mapping Tools (GMT) for viewing the results, especially netCDF files, is recommended.

D Sample experiment & use cases

In this chapter a sample experiment and the three use cases of CrusDe are explained. The first use case of CrusDe is simulation of loading effects; especially surface deformation. The study area of the sample experiment is the Icelandic volcano Mt. Hekla which is familiar from chapter 5. The experiment definition that complements the ones already given in chapter 5 with an instantaneous response to a disk load that builds up over time is explained in section D.1. The second use case is the management of plug-ins which is described in section D.2. Management of completed experiments represents the third of CrusDe's use cases which is briefly presented in section D.3.

D.1 Experiment definition

Listing D.1 shows a complete experiment definition that can be used to simulate the response of an elastic half-space to a disk shaped lava buildup of the 12-day millennium eruption at Mt. Hekla.

From listing D.1 it is obvious that an experiment definition is divided into two parts. The global part from line 4–15 defines parameters important to the simulation core such as the output filename and the region of interest. The `timesteps` parameter is optional and only necessary when a load history is to be simulated.

The rest of the file from line 17–51 defines all the plug-ins that are composing the simulation model of the experiment. The `load_history` definition is optional; all the others are mandatory. The only place to define more than one plug-in is within the `postprocessor` definition (see lines 44–47).

```
<?xml version="1.0" encoding="UTF-8"?>

<experiment name="hekla, disk">
 <file   name="result" value="./hekla_disk_elastic.nc" />
                                    <!--result file-->

 <region name="west"   value="420000"/><!--region of interest-->
 <region name="east"   value="520000"/><!--Lambert coordinates-->
 <region name="south"  value="320000"/>
 <region name="north"  value="420000"/>

 <parameter name="timesteps" value="12"/>
                             <!-- 12 day eruption-->
 <parameter name="gridsize"  value="500"/>
                             <!--side length of cells-->

<!-- EARTH MODEL -->
 <greens_function>
  <plugin    name="elastic halfspace (pinel)"/>
  <parameter name="g"  value="9.81"/>  <!--acc. due to gravity-->
  <parameter name="nu" value="0.25" /> <!--Poisson ratio-->
  <parameter name="E"  value="40"/>    <!--Young modulus-->
 </greens_function>

<!-- SPATIAL LOAD MODEL -->
 <load_function>
  <plugin    name="disk load" />            <!--disk geometry-->
  <parameter name="height"   value="9.82"/> <!--parameters describe-->
  <parameter name="radius"   value="2514"/> <!--density and-->
  <parameter name="center_x" value="467400"/><!--dimensions of the-->
  <parameter name="center_y" value="384700"/><!--disk load.-->
  <parameter name="rho"      value="2900"/>
 </load_function>

<!-- TEMPORAL LOAD MODEL -->
 <load_history>
  <plugin    name="sinusoidal" />
  <parameter name="period_length" value="24" />
             <!--twice the timesteps to simulate only load build-up -->
  <parameter name="peak" value="12" />
             <!--maximum is on the last simulated day-->
</load_history>

 <postprocessor>
   <plugin name="xy2r" />
   <plugin name="hori2vert-ratio" />
 </postprocessor>

<!-- postprocessor, result handler & convolution operator -->
 <output> <plugin name="netcdf writer"/> </output>
 <kernel> <plugin name="fast 2d convolution" /> </kernel>

</experiment>
```

Listing D.1: Example experiment definition.

The experiment definition is enclosed by the `<experiment />` tag whose attribute `name` serves as an identifier in the experiment database (see section D.3).

Given the experiment definition is saved to `examples/hekla/disk_elastic.xml`, it can be executed by invoking the following command at the command line:

```
$> ../../crusde/crusde disk_elastic.xml
```

D.2 Using the plug-in manager

In case parameters or the name of a plug-in to be used in an experiment are forgotten, or a new plug-in is to be added, the plug-in manager can be started from the command line using the `-P` parameter:

```
$> ./crusde -P
```

Figure D.1 shows a screenshot of the plug-in manager's graphical user interface (GUI). The left column presents the names of all installed plug-ins sorted into a tree of categories. Selecting one of the plug-ins, e.g., '`elastic halfspace (pinel)`' as depicted in the figure, will load all details about the plug-in that are found in the database into the right frame. Dependencies and parameters are extracted from the plug-in upon registration with the framework. Most of the other information must be given by the plug-in developer as textual information in the plug-in (see appendix E.3).

The first of the three buttons to the right allows adding a new plug-in. A window will pop up which asks for the shared library (`*.so`) that is to be added. The new plug-in is automatically sorted into the category it defines and saved to the plug-in repository. It will show up in the plug-in tree when it passed the tests the plug-in manager performs.

The delete button will remove the selected plug-in from the database. If a category is empty, i.e. no plug-ins are installed, it will not show up in the tree view. The close button will shut down the plug-in manager.

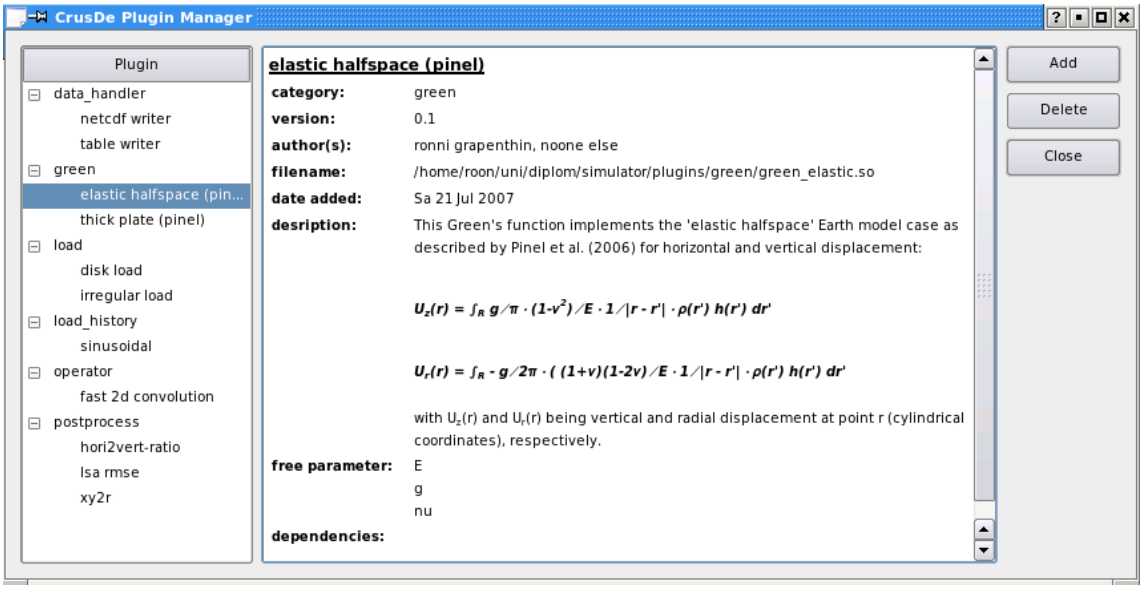


Figure D.1: Screenshot plug-in manager GUI

D.3 Using the experiment manager

The experiment manager is started from the command line using the `-M` parameter:

```
$> ./crusde -M
```

Figure D.2 shows a screenshot of the experiment managers GUI depicting the record of the experiment that is defined in listing D.1. The interface is quite similar to that of the plug-in manager. The leftmost frame shows the names (as defined in the experiment definition) of all performed experiments which create categories for the particular experiments. The experiment manager shows single experiments identified by date, time and user as item of the experiment name. Selecting one of the experiments, e.g. the Hekla experiment from Sun 22 July 2007, will load all data from the experiment definition into the right frame. Hitting the delete button will remove the selected experiment from the database. The model results are, however, **not** touched by this operation. The close button closes the experiment manager.

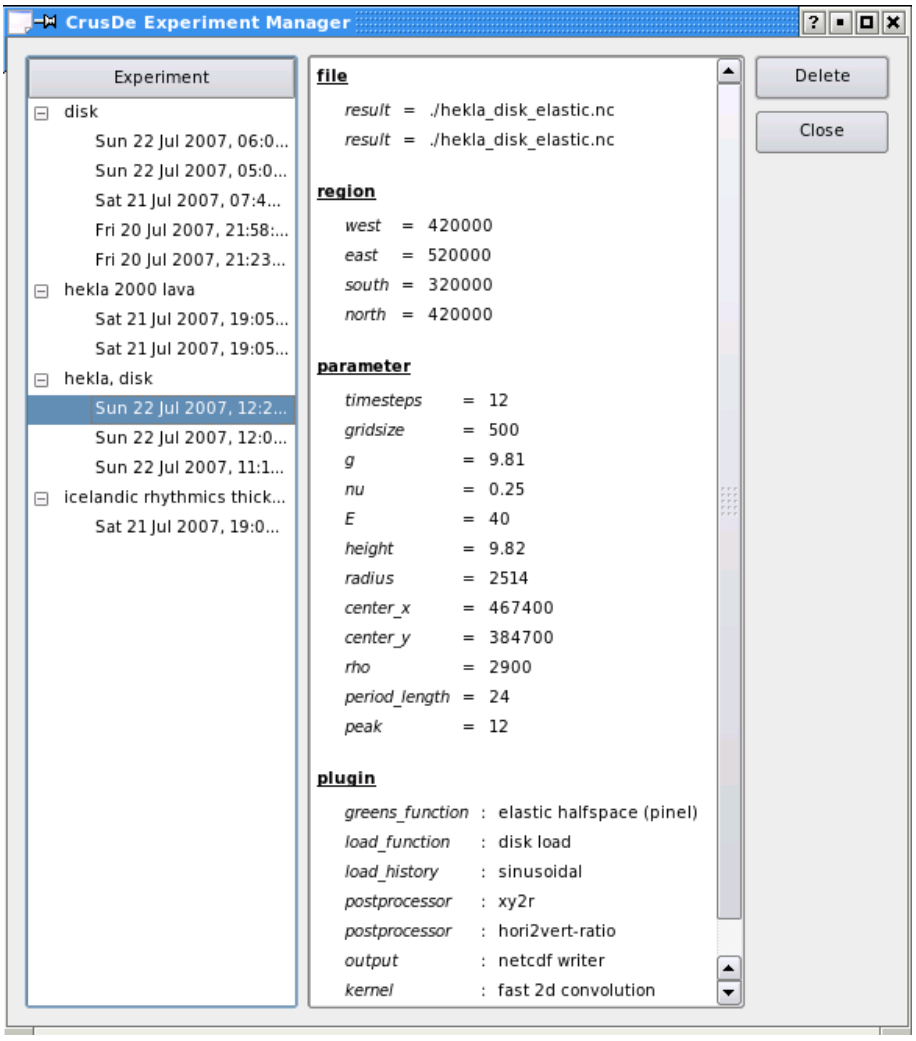


Figure D.2: Screenshot experiment manager GUI

E Implementation details

E.1 Interfaces of the simulation framework

This section intends to give a short overview of the 'language' in which the plug-ins of the functional layer and the rest of the simulation framework communicate. The communication works over several interfaces of the interface layer. From the simulation cores point of view they can be divided into two categories:

- needed-interfaces, and
- provided-interfaces.

Needed-interfaces are interfaces a plug-in must implement in order to be of any use to the simulation core. The simulation core uses these interfaces to retrieve information about the plug-ins and to get them to do work. *Provided-interfaces* (API) offer functionality to the plug-ins. Via this channel, plug-ins can request information or make announcements to the simulation framework. The description given here focuses on the most important interfaces and is not complete. A complete documentation can be found on the enclosed CD in the folder `/crusde/doc_html` (see appendix B).

E.1.1 Needed interfaces

A needed-interface of the simulation core is a function a plug-in must implement, because it is called at runtime. If a needed-interface is not implemented, CrusDe cannot operate. Thus, the plug-in manager will test a candidate plug-in for existing implementations of the needed-interfaces.

Before any of the necessary interface functions is bound, instances of the plug-ins have to be created. The simulation core delegates this responsibility to the interfaces / interface classes (`PluginIF`, `Load pluginIF`, `Green pluginIF`, `Load history pluginIF`, and `Data out pluginIF`). For each plug-in that is utilized within CrusDe an instance of the respective interface class is created. This instance will load a shared library from the plug-in repository which is then accessible for interface binding. The plug-ins are loaded (and unloaded after the results are obtained) in this sequence:

1. Green's function,
2. load function,
3. convolution operator,
4. result handler,
5. load history, and
6. all postprocessors (as found in the experiment definition).

Figure 4.2 shows the inheritance relation that makes *needed-interfaces* defined by `PluginIF` a subset within the inheriting interfaces. Thus, all plug-ins have to provide the following functions to implement the `PluginIF`-interface (ordered in accord to the calling sequence in the simulation core):

`void register_parameter()` The function `register_parameter()` tells a plug-in to announce the parameter names it requests from the experiment definition and the respective addresses for the values in memory. The simulation core will take care on copying the value that matches the parameter name from the experiment definition to the given address. Since this is the first function to be called at a plug-in, the values can be used during initialization (see below).

`void register_plugins()` If a plug-in intends to use the functionality of other plug-ins, it must call the respective API function within `register_plugins()` (see listing E.1).

`void init()` The function `init()` is intended to allow for resource acquisition, setting of internal variables, and calculation of constants of a plug-in to achieve a defined state.

`void register_output_fields()` Since the modeling result is stored in an array, a plug-in can call the respective API functions (see listing E.1) to have CrusDe allocate enough memory for the results within the function `register_output_fields()`. A Green's function, for instance, will probably want to have memory allocated for the displacement directions (vertical, horizontal$_x$, and horizontal$_y$). This interface is currently only called at Green's functions and postprocessors).

`'run'-functions` The 'run' functions differ depending on category and are explained below. However, they are called at this position in the sequence.

`void clear()` The function `clear()` is called before unloading of the plug-in to allow freeing of resources that were allocated during initialization / runtime.

The major difference between the 5 needed plug-in interfaces shown in figure 4.2 is the 'run'-function which holds the execution functionality, i.e. the purpose of a plug-in. Some plug-ins need parameters at runtime to fulfill their duty. These differ among categories. Thus, depending on a plug-ins category different 'run'-functionality must be implemented:

`void run()` Both `PluginIF` and `Data out pluginIF` expect plug-ins to provide functionality via the function `run()`.

`double constrain_load_height(double max_load, int x, int y, int t)` A load history plug-in must implement this function of `Load history pluginIF` and return a load height that depends on the maximum load, `max_load`, at point `(x,y)` at time `t`.

`double get_value_at(int x, int y, int t)` `Load pluginIF` expects this function to be implemented by load function plug-ins. The function must return the load at point `(x,y)` and time `t` in the region of interest.

`int get_value_at(double **result, int x, int y, int t)` Green pluginIF expects a Green's function plug-in to implement this function. It must determine the Earth's response at point `(x,y)` and time `t`. The result for each spatial dimension must be written to the respective index of the array `result` retrieved from `crusde_register_output_field()` (see below).

E.1.2 Provided interfaces: Framework API

The simulation framework does provide an interface which the plug-ins can use mainly to retrieve information. Many of the needed-interfaces are in a sense complemented by the provided-interfaces. As section E.1.1 showed, many needed-interfaces simply trigger an action in a plug-in which often results in a response sent via the API. Listing E.1 gives a full overview of the provided-interfaces. The following list describes some of the more important API functions:

`void crusde_register_param(double* p, const char* name, PluginCategory cat)` The plug-in of category `cat` registers the memory address of parameter `p` with name `name` at the simulation framework. The value for `name` comes from the experiment definition wherein it must be found with `name` as identifier.

`void crusde_register_output_field(int* position, FieldName f)` The plug-in adds an additional field to the output array. The fields position in the array is stored at the memory address `position` is pointing to. The second parameter denotes whether a spatial `(x,y,z)` any other field `(add)` is added.

`int crusde_get_green_at(double** result, int x, int y)` The value of the Green's function at point `(x,y)` is written to the address of the array `result`. The size of `result` depends on the number of spatial directions the for which Green's function calculates values (e.g., vertical, horizontal$_x$, horizontal$_y$).

`double crusde_get_load_at(int x, int y, int t)` The load at point `(x,y)` and time `t` is returned.

`double crusde_constrain_load_height(double max_load, int x, int, y, int t)`
The load height at point `(x,y)` and time `t` depending on the maximum load `max_load` is computed.

`double** crusde_get_result()` Returns a pointer to the array that keeps the convolution result.

`green_exec_function crusde_request_green_plugin(char* plugin)` Using this function, any Green's function plug-in can request a pointer to the 'run'-function of the fellow Green's function that is named `plugin`. Analog functions exist for the other plug-in categories.

E.2 Implemented plug-ins: details

E.2.1 Convolution operator: 'fast 2d convolution'

As mentioned in section 4.3.1 the '`fast 2d convolution`' plug-in uses version 3 of the FFTW library to transform its operands to the spectral domain and back. When using FFTW a basic sequence must be kept (see listing E.2):

- allocate memory for DFT input and output,
- create a *plan* for the transform,
- execute the plan to perform the DFT (as often as desired), and
- free resources: destroy plan, and free allocated memory.

A *plan* is an executable data structure that accepts input data and calculates the respective DFT. The planner, given the size of the input and the data type, automatically measures the run time of different plans and selects the fastest. This usually consumes time which can be limited by a flag that indicates how long a user is willing to wait for a reasonable plan (e.g. `FFTW_ESTIMATE` takes short planning time). Once we have a plan it can be applied to many data. Thus, sometimes

```
,
1  void crusde_register_param(double* param,
2                             const char* param_name, PluginCategory);
3              /*register parameter with CrusDe*/
4  void crusde_register_output_field(int* position, FieldName);
5              /*register output field with CrusDe*/
6
7  int crusde_get_size_x();      /*length of the region of interest*/
8  int crusde_get_size_y();      /*width of the region of interest*/
9  int crusde_get_size_t();      /*total number of time steps*/
10 int crusde_get_gridsize();    /*side length of a grid cell*/
11 int crusde_get_min_x();       /*westernmost coordinate of ROI*/
12 int crusde_get_min_y();       /*southernmost coordinate of ROI*/
13 int crusde_get_dimensions(); /*total output fields*/
14 int crusde_get_displacement_dimensions();
15                             /*total spatial output fields*/
16 int crusde_model_time();      /*current time step*/
17 int crusde_stepsize();        /*time increment with each model step*/
18 int crusde_get_x_index();     /*index of x-deform in result array*/
19 int crusde_get_y_index();     /*index of y-deform in result array*/
20 int crusde_get_z_index();     /*index of z-deform in result array*/
21
22 const char* crusde_get_observation_file();
23             /*filename of points to be observed (unused)*/
24 const char* crusde_get_load_file();
25             /*file that contains load definition*/
26 const char* crusde_get_out_file();
27             /*filename for result output*/
28
29 /*green's function coefficients at x,y*/
30 int crusde_get_green_at(double** res, int x, int y);
31 /*load at x,y,t*/
32 double crusde_get_load_at(int x, int y, int t);
33 /*constrain load h at x,y,t*/
34 double crusde_constrain_load_height(double h, int x, int y, int z);
35
36 void crusde_set_result(double**); /*return pointer to model results*/
37 double** crusde_get_result();     /*pointer to model results*/
38 void crusde_set_quadrant(int);    /*get/set quadrant in coord. sys...*/
39 int  crusde_get_quadrant();       /*where green's func. is calculated*/
40 void crusde_exit(int exitcode);   /*have CrusDe terminate gracefully*/
41 boolean  crusde_load_history_exists();
42             /*a load history was defined, returns {true|false}*/
43
44 /*functions to request pointers to the run time function of other
45   plugins of the same category (p is plugin name)*/
46 green_exec_function       crusde_request_green_plugin(char* p);
47 load_exec_function        crusde_request_load_plugin(char* p);
48 run_function              crusde_request_kernel_plugin(char* p);
49 run_function              crusde_request_postprocessor_plugin(char* p);
50 loadhistory_exec_function crusde_request_loadhistory_plugin(char* p);
```

Listing E.1: CrusDe API.

time invested into planning might pay off when extensive reuse of a plan is expected (*Frigo and Johnson*, 2005).

The implementation of FFTW's processing structure in the provided '`fast 2d convolution`' plug-in is distributed among the interface functions `init()`, `run()`, and `clear()`:

`init()` According to the considerations of section 3.6, the `init` function first calculates a size S for input and output of the fast two-dimensional convolution that is a power of 2 and large enough to avoid wrap around effects. Following this, memory for the transform input and output is allocated[1]. After having allocated six arrays (input and output for load function, Green's function and convolution result, respectively), three plans for a two-dimensional DFT are created. Two plans to transform load function and Green's function to the spectral domain (mapping from the real to the complex plane), and one plan for the inverse transform of the convolution result (mapping from the complex to the real plane).

`run()` The `run` function iterates over the region that is defined in the experiment definition and initializes the Green's function array for each displacement direction the Green's function plug-in offers (e.g., x, y, z) using the API function `crusde_get_green_at`. The load array is initialized at time step 0 using the API function `crusde_get_load_at` and then re-calculated every following time step if a load history function was defined. For each displacement direction a convolution result is obtained by:

- executing the DFT,
- complex multiplication of the transform results in the spectral domain, and
- execution of an inverse DFT to transform the result to the original domain.

[1] FFTW expects input arrays to be one-dimensional. Higher dimensionality is to be realized via offsets. For instance, to retrieve the value of the (i,j,k)-th element of an array M of size $I \times J \times K$, the expression $M[k + K * (j + J * i)]$ would be used.

```
,
#include <fftw3.h>
#include <complex.h>
/* use fftw_allocate if memory is allocated dynamically*/
double       load_in[S],  green_in[S],  conv_out[S];
fftw_complex load_out[S], green_out[S], conv_in[S];
fftw_plan    plan_load,   plan_green,   plan_conv;
int i = -1;

/* plan real to complex 2-dimensional DFT */
plan_load  = fftw_plan_dft_r2c_2d( S_x, S_y, load_in,
                                   load_out,
                           FFTW_ESTIMATE );

plan_green = fftw_plan_dft_r2c_2d( S_x, S_y,  green_in,
                                   green_out,
                                   FFTW_ESTIMATE );

/* plan a complex to real inverse transform */
plan_conv  = fftw_plan_dft_c2r_2d( S_x, S_y, conv_in,
                                   conv_out,
                           FFTW_ESTIMATE);

/* Initialize green_in and load_in with data ... */

fftw_execute(plan_load);    /* compute load  DFT */
fftw_execute(plan_green);   /* compute green DFT */

/* point-wise complex multiplication and normalization*/
while(++i < S){
   conv_in[i] = (green_out[i] * load_out[i]) / S;
}

/* inverse DFT, conv_out now holds the result */
fftw_execute(plan_conv);

/*free ressources*/
fftw_destroy_plan(plan_load);  /* use fftw_free to free */
fftw_destroy_plan(plan_green); /* memory allocated with */
fftw_destroy_plan(plan_conv);  /* fftw_allocate         */
```

Listing E.2: Example showing how a fast convolution could be implemented using FFTW

Invoking the API function `crusde_set_result` the '`fast_2d_convolution`' plug-in notifies the simulation core about the memory address of the convolution result.

`clear()` This function destroys all plans created during initialization and frees all other memory allocated upon initialization.

E.2.2 Green's functions

`elastic halfspace (pinel)`

In this plug-ins `init()` function constant parts of the Green's functions are calculated to minimize the number of operations at runtime. The `get_value_at` function computes the value of the Green's function at point (x, y) which is converted to cylindrical coordinates before computation. The result, however, is in Cartesian coordinates (see appendix E.2.4).

`thick plate (pinel)`

In the implementation of the function `request_plugins` this plug-in asks the simulation core to load the '`elastic halfspace (pinel)`' plug-in and return a pointer to its `get_value_at` function.

Apart from initialization of constants the `init` function also prepares the integral functions and the integration workspace of the GNU Scientific Library[2] (GSL) (*Galassi et al.*, 2007). The GSL is used to compute the integrals contained in equations 3.7 and 3.8 numerically (using the function `gsl_integration_qag`). The implementations of the Bessel functions provided by the GSL are used within this plug-in as well. Since the hyperbolic functions are strictly monotonic increasing and quickly approach infinity, the bounds of the integral must be confined. *Pinel (pers. comm., 2007)* observed that the integrals of equation 3.7 and 3.8 show convergent behavior for values smaller than the onset of the trend towards infinity of the hyperbolic functions. The integral interval is thus confined to $[0, 0.003]$. Using this

[2] `http://www.gnu.org/software/gsl/`

interval solutions can be obtained for realistic parameter settings (e.g., $H <= 50\,km$, an unrealistically thick crust would again lead to terms of infinity). This approach underestimates the values for the Green's function.

The `get_value_at` function invokes the plug-in for the elastic half-space which returns a result just as if it was the main Green's function plug-in. To this value the product of constants and the integral function is added to obtain a value for the implemented Green's functions at point (x, y).

E.2.3 Load functions and load history functions

`disk load`

The '`disk load`' plug-in defines a uniformly distributed load in the shape of a disk. Parameters that define center, radius, and height of the disk have to be defined in the experiment definition (see listing D.1). The implementation of the function `get_value_at` checks whether the Euclidian distance of point `(x,y)` to the center of the disk is smaller than the disks radius. If so, the load for that point is returned; zero otherwise. The load height might, however, be constrained by a load history if defined.

`irregular load`

The '`irregular load`' plug-in differs from the `disk load` mainly in that it reads the load heights from a separate load file during processing of the `init` function. The respective load file must be defined in the experiment definition (see listing 5.1) and is expected to be in a tabular format with three columns separated by spaces; rows contain only numbers. The values of a row are interpreted as:

```
Longitude Latitude Height
```

Longitude and latitude are expected to be integers in Lambert coordinates. The heights (double values) read from the file are stored in an array that represents the examined area. Points not found in the load file are initialized to a height of 0.0. When the function `get_value_at` is invoked, the load height of the requested point

(mapped to the array) is returned and might be constrained by a load history if defined.

`sinusoidal`

The '`sinusoidal`' plug-in defines a simple load history that alternates in the form of a cosine. The plug-ins function `constrain_load_height` implements the following formal model for a load history (*Grapenthin et al.*, 2006):

$$h(\vec{r'}, t) = \frac{h_m}{2} \left[1 + cos(\frac{2\pi}{p} (t - t_{h_m}))\right] \tag{E.1}$$

The load, $h(\vec{r'}, t)$, varies as a harmonic function of time (t) at each point ($\vec{r'}$). h_m is the maximum load height which is a parameter of `constrain_load_height`. The length of the cosines period is influenced by p. If, for instance, a period of one year is desired and t represents days, then $p = 365$. Phase shifting of the cosine enables a control of the time when the load reaches its maximum (t_{h_m}). Thus, p and t_{h_m} are parameters to be defined in the experiment definition.

E.2.4 Postprocessors

The two implemented postprocessors call `crusde_register_output_field` to have the simulation core allocate memory for their results.

`xy2r`

The '`xy2r`' postprocessor plug-in converts the convolution results that are obtained in Cartesian coordinates to cylindrical coordinates. The conversion is implemented in its `run` function (e.g., *Bronstein et al.* (2001)):

$$U_h(\vec{r}) = \sqrt{U_h(x)^2 + U_h(y)^2} \tag{E.2}$$

The horizontal surface displacement in cylindrical coordinates is then stored at the position in the result array that was obtained by calling `crusde_register_output_field`.

`hori2vert-ratio`

This plug-in calculates the ratio between horizontal and vertical surface displacements:

$$k = \frac{\sqrt{U_h(x)^2 + U_h(y)^2}}{U_v(x, y)} \tag{E.3}$$

with k being the ratio. The values for k are also stored at the position obtained upon registration of an output field.

E.2.5 Result handler

Figure 4.2 depicts that plug-ins of the result handler category implement the `Data out pluginIF`. It is their responsibility to assure that model results are written to persistent memory (or in some other way made available to the outside world).

`table writer`

This result handler plug-in writes the model results in tabular form to a text file. Its implementation of the `run` function composes one line for each point of the examined area in the format:

`time x-coord y-coord [U_x] [U_y] [U_v] [add_1...add_k]`

The terms in brackets are, in a way, optional. The existence of the particular displacements depends on whether the Green's function calculates them. Thus, horizontal deformation (`U_x,U_y`) can be missing while vertical deformation (`U_v`) ends up at the position in the output table that succeeds the `y-coord`. However, the output file contains header information that shows which deformation fields are at which position. The semantics of the additional result fields (`add_i`) depend on the order of the (post-processor) plug-in definition in the experiment definition. Momentarily, the header information is quite sparse about this.

`netcdf writer`

All information that complements the description given in section 4.3.1, e.g., variable naming, is found in CrusDe's documentation (on the enclosed CD).

E.3 Implementing a new plug-in

A guide to implement a new plug-in for CrusDe is the template file

`crusde/plugin_src/temp_plugin.c.tmp`.

There, all the functions that implement the `PluginIF`-interface are defined and commented to explain the semantics of each of the functions.

Most important when implementing a new plug-in is the inclusion of the file `crusde_api.h` which provides access to CrusDe's API (see listing E.1).

Once the plug-in is written it must be compiled and a shared library must be created. A simple approach is to copy the source file into the respective subdirectory of the directory `crusde/plugin_src` that denotes the plug-ins category and then type:

```
$> make all
```

All source files (`*.c`) in this directory will be compiled and shared libraries with the names `<source-filename>.so` are created.

If the provided makefile is not used, the sources must be compiled with the `-fpic` or `-fPIC` flag for 'position independent code' generation which is necessary when creating shared libraries. If the source code for the new plug-in is saved to the file `plugin.c`:

```
$> gcc -fPIC -c -Wall plugin.c
```

The `-c` parameter tells the compiler to create an object file (`plugin.o`).

To create the shared library from the object file, the `-shared` flag must be set for the linker:

```
$> gcc -shared -o plugin.so plugin.o -lc
```

The `-lc` parameter links the object file `plugin.o` against the C standard library.

Once the plug-in is created it can be added to CrusDe using the plug-in manager (see appendix D.2).

Index

Erklärung

Ich erkläre, diese Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt zu haben.

Ich bin damit einverstanden, dass ein Exemplar dieser Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt wird.

Berlin, den July 31, 2007 .....