DISSERTATION

Beyond Atavistic Structures in Scientific Computing

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
von

PHILIPP SCHWAHA

Wien, im November 2010

ματαιότης ματαιοτήτων είπεν ο
εκκλησιαστής ματαιότης
ματαιοτήτων τα πάντα ματαιότης

ΕΚΚΛΗΣΙΑΣΤΗΣ,1:2

Drum besser wär's, daß nichts
entstünde.

Mephistopheles

Acknowledgement

Knowledge is in the end based on
acknowledgement

Ludwig Wittgenstein

The slow and tedious task of compiling the information found within these pages has had many direct and indirect supporters over the years, to whom I sincerely wish to express my gratitude. Alas, all their diligent and faithful support could probably not avert my limitations and imperfections of introducing oversights, omissions and errors, for which I am deeply sorry and which should in no way be regarded as a sign of disrespect to all their hard work.

I wish to thank professor Grasser who saw fit to offer me an opportunity to work as a member of his group and Robert Entner for introducing me. Furthermore, I wish to thank professor Selberherr for his continued support and for providing a challenging working environment at the institute, which forced me to adapt and evolve. My gratitude goes to my former colleagues at the institute for whenever they provided support and advice, especially Michael Spevak, with whom I worked only a short but influential and inspiring time, before his calling took him elsewhere.

I am deeply indebted to Михаил "Mixi" Недялков, who has patiently been introducing me to the arcane art of Monte Carlo methods and has been a guide through the mysterious and twisting realm of quantum mechanics and its obscure secrets.

I am very grateful to Виктор Свердлов for his time and for his moral support.

I also wish to express my gratitude to Franz Stimpfl and Josef Weinbub for their fresh and interested questions, which reminded me, that there is always more to learn and explore.

Проф. Иван Димов has my sincerest gratitude for agreeing to be an examiner of my thesis despite his busy schedule.

I owe an exceptionally big debt of gratitude to René Heinzl, who has been my coworker, friend and brother in arms since the earliest days on this quest for knowledge and understanding. Not only did the heated debates and discussions provide new, valuable insights, but our joint travels, which took us all around the globe, opened my eyes to new views on life beyond the merely scientific. I would have abandoned the project, without his fierce and fiery loyalty and belief in me being able to of its feasibility, when mine had already faltered.

Of course nothing would have been possible without the continued and unwavering support of my family and friends, who provided me with a silver lining in the dark times and kept me from completely loosing the last traces of my sanity. That their names are not given explicitly makes them no less dearer, my gratitude no less, as those knowing me indubitably know.

I hope no one is offended by these words, but if this should be the case, you should know that I do take exception to you taking offence and therefore invite you to seek me out so that I may offend you properly.

Non nobis . . .

Abstract

The evolution of both scientific and engineering developments results in an ever growing demand to address problems of increasing complexity in a swift and precise manner. Numerical simulations using digital computing devices are a valuable asset in meeting these demands. The resulting evolution of digital computers has lead to the development of several programming techniques and paradigms, which are shortly outlined. While it is critically important to be aware of the capabilities as well as the limitations of the deployed digital tools, which are also briefly touched upon, it is not sufficient for the setting of scientific computing, since at least a rudimentary understanding of the problem domain to be treated is also required.

As both purely scientific as well as applied engineering make extensive use of mathematical formalisms, it is only natural to use the mathematical structures as a guide. This is especially important in order to transcend the still comparatively primitive abstractions, consequently also the implemented structures. Therefore a two pronged approach has been chosen, presenting on the one hand the theoretical outline, including the mathematical backbone as well as a physical application of the outlined mathematical structures, while on the other hand showing realizations of the described structures.

Thus, the two parts mirror each other as both progress from essential base components, such as sets and algebraic structures, to form abstract constructs of increasing complexity and sophistication, including entities from differential geometry, integration, and probability.

The components are then put to use in order to model what is perceived as physical reality with a special attention to the field of dynamics, including even a small foray into the field of quantum mechanics. Again, the corresponding sections of theory and treatment in the environment of application attempt to mirror each other.

Kurzfassung

Die Evolution der wissenschaftlichen und technischen Entwicklung erlauben stetig wachsende Probleme zunehmender Komplexität einer raschen und präzisen automatisierten Behandlung zuzuführen. Der Bereich der numerische Simulation, ausgeführt auf digitalen EDV-Geräten, ist ein in Bedeutung gewinnender Teil der Wissenschaft zur Erfüllung dieser Anforderungen. Die daraus resultierende Veränderung der digitalen Rechner führte zur Entwicklung von verschiedenen angepassten Programmiertechniken, die kurz erklärt werden. Während es von entscheidender Bedeutung ist, sich der Fähigkeiten sowie der Grenzen der eingesetzten digitalen Werkzeuge und der darunterliegenden Paradigmen, die ebenfalls kurz angesprochen werden, bewußt zu sein, ist es im Zusammenhang mit wissenschaftlichem Rechnen nicht ausreichend, da zumindest auch ein rudimentäres Verständnis des zu behandelnden Problemgebiets erforderlich ist.

Da der rein akademische wie auch der angewandete Bereich der technischen Wissenschaften umfangreichen Einsatz von mathematischen Formalismen erfordert, ist es nur natürlich, die mathematischen Strukturen als Leitfaden zu nutzen. Dies ist besonders wichtig, um die noch vergleichsweise einfachen Abstraktionen und die damit einhergehende Umsetzung zu überwinden. Aus diesem Grund ist die Arbeit in zwei sich ergänzende Teile gegliedert, wobei der erste Teil den theoretischen Entwurf einschließlich des mathematischen Rückgrats sowie einer physikalische Anwendung der beschriebenen mathematischen Strukturen präsentiert, und der zweite Teil Realisierungen der beschriebenen Strukturen zeigt.

So spiegeln die beiden Teile einander, da beide von wesentlichen Basiskomponenten wie Mengen und algebraischen Strukturen fortschreiten, um abstrakte Konstrukte zunehmender Komplexität und Allgemeinheit, einschließlich Elemente der Differentialgeometrie, Integration und Wahrscheinlichkeitsrechnung, zu bilden.

Anschließend werden diese Komponenten eingesetzt, um das, das als physische Realität wahrgenommen wird, mit einem besonderen Augenmerk auf den Bereich der Dynamik, darunter sogar einen kleinen Ausflug in den Bereich der Quantenmechanik, zu modellieren. Auch die entsprechenden Abschnitte der Theorie und Behandlung in der Anwendung versuchen einander zu vervollständigen.

Contents

Motivation

What is it that drives the development and refinement of learning and understanding? While not being able to give a definite answer to this simple question, I can outline the motivation for endeavouring to advance the level of abstraction in the field of scientific computing.

From ancient times people strove to unravel the mysterious workings of the world surrounding us. From these considerations sprang what has been called philosophy, which gradually gave birth to different distinct fields of science, when they had sufficiently matured. The maturation of additional sciences brought the establishment of formalisms with which to express ideas in a clear and reproducible fashion. To this end natural sciences naturally employ abstractions which have been developed and refined in the field of mathematics. The application of mathematical principles has also lead to the spawning of machines to aid in the evaluation of the resulting mathematical formulations. The machines, now commonly known as computers, have grown rapidly in complexity, so as to become a field of research themselves, in order to provide more and more powerful tools with which to express ideas and patterns of thought in this automated setting.

At the same time the descriptions used in theoretical branches of sciences have evolved from bare basic expressions to formulations of increasing abstraction and expressive power. The increase in abstractions facilitates the treatment of a wide range of problems, but does not remove the need to finally carry out basic calculations. It is the simplest algebraic manipulation which computers are typically applied to. However, limiting computers to deal with only the lowest level of problems denies to acknowledge, and even less use, the whole evolution of abstractions developed in the course of several millennia.

However, it seems that many working with computers are reluctant and even blatantly refuse to consider employing higher levels of abstraction. The reasons seem to range from fear of squandering precious computing resources to a fear of the unknown and a refusal to learn, which while they may have some merit in a conservative industrial setting, are misplaced in anything even remotely connected to academia, which should always attempt to push the envelope of what is possible on all fronts. Nevertheless, the use of higher levels of abstractions seems to be alien to software development and is still not commonplace, despite the fact that software is itself nothing but abstraction and it is thus desirable to map mathematical abstractions of a higher level to the computational context of the machines.

To this end abstract mathematical concepts are defined for which a subsequent realization in a programming language are shown. By utilizing a conceptual approach to implementation the provided structures can transcend the primitive notions of basic data types and allow to maintain a high level of abstraction throughout all of the development cycle.

This dictates that the theoretical setting is clearly defined, which is best accomplished by provid-

ing mathematical expressions and tools. Since without the availability of mathematical terms and definitions, abstractions quickly degenerate into a vague mass of wishful thinking, considerable effort is invested to prevent this.

The approach to spend considerable effort in analysing a given problem instead of rabidly starting a frantic process of writing as much code as possible is often frowned upon and seen as unnecessarily philosophical. While it may seem preferable to some to produce enormous amounts of code of the lowest level of abstraction instead of utilizing high levels of abstraction, I find this notion abhorrently appalling. Not only does such an approach neglect the labours of previous generations of scientist, it is also a burden for any subsequent development as the sheer amounts of code become unmanageable, resulting in costly redevelopment cycles, which could have been avoided in the first place. This is not to say that higher abstractions will guarantee against redevelopment, but the lower the abstraction is, the longer it takes to decipher, as anybody digging through legacy code may already have experienced.

Therefore, I consider it as a big compliment should anyone find something he may call philosophical, as which it may not have been intended. For better or worse, philosophy is the mother of modern sciences and it would not be appropriate to break with the procedures and traditions completely, which have served philosophers to deal with the abstract and ungraspable. As is already derivable from the etymology of philosophy, it attempts to unveil the reasons behind phenomena, to develop deep understanding. It is only when we have a deep and profound understanding that we can arrive at efficient solutions applicable to a wide field of particular problems. While this has apparently already been known in ancient times[1], it seems to be neglected in connection with the field of digital computers, where generic programming techniques are just currently emerging. Their proper development and deployment, however, is contingent on a highly abstract description, which requires more forethought compared to more traditional approaches, which can very well be described as being philosophical. The rewards for investing additional time in analysis and contemplation include greater flexibility and an even wider field of application.

Some may wonder with regard to the usefulness of anything treated here, if what has worked so far still works. However, I challenge those who ask what this is for to apply the question recursively to whatever they are doing and see if they find an answer that is neither dogmatic nor beyond all doubt. It may be called to have no use, but who, in the very end, truly has?

[1] As is attributed to Archimedes of Syracus.

Doh! I wanted to leave this page blank intentionally ...

Chapter 1

Introduction

Lasciate ogni speranza,
voi ch'entrate!

Dante Alighieri

One possibility of dealing with the increasing intricacies is to attempt to uncover the minimal and fundamental structures at the core of the problem under investigation. This procedure has already been applied to great success in the field of theoretical sciences. It accomplishes this by means of suitable abstractions. In a sense it continues an age old quest already present in the days of alchemists for the true name of something[1]. The process of abstraction makes heavy use of mathematics and has seen applications, and thereby intensified development, for mathematical constructs which previously had been purely theoretical, such as topology and differential geometry. The desired benefit is a simplified presentation of the problem at hand at the cost of increasing the presumed knowledge, thereby dividing and encapsulating complexities.

Research and development have been greatly supported and accelerated by using the resources and opportunities made available by digital computers and the numerical methods they support. The desire for the availability of alternatives to analytical solutions of scientific problems is at least as old as the calculus used to describe them, as it proved difficult to impossible to provide solutions, when the number of degrees of freedom increases[2]. However, applying the facilities offered by digital computers, especially in the field of solid state physics, may serve not only to promote the field of application, but also the advancement of the computing infrastructure itself as it is itself designed using the foundations of our understanding of physics. Thus advances obtained are fed back directly to obtain the tools employed for the advance, thereby making them more powerful and providing the means to tackle bigger or more complex problems.

The increase in complexity of both, the problem scenarios which need solution as well as the tools deployed for the solution is, however, not without problems. A failure to address this complexity properly may very well lead to a stall of the whole process as it begins to diverge. While science and engineering always strive for as simple as possible descriptions of physical models[3], it is not reasonable to expect the continuing trend of increasing complexity of descriptions to be

[1] The idea was that once the true name was discovered, it was possible to exercise complete control.
[2] The dreaded and yet admired many body problem.
[3] Linear time invariant (LTI) is a very common desire.

reversed drastically, as more and more previously neglected details are taken into consideration. It therefore falls on the tools to be adapted to compensate for or at least deal with the rise in complexity.

While these same ideas have also been established in conjunction with software development, they more often than not remain isolated to a select few, who concern themselves (solely) with the theory of software development. The ideas and connected methods have not spread to be applied in the field of various scientific applications outside of this small circle of software specialists. This may in part be attributable to the fact that topics connected with software development are often viewed as unscientific[4] and more as a craft which should be dealt with as swiftly as possible in scientific communities employing numerical schemes as tools. This results in the problem of the divergent development of what methodology has already been made available and what is actively deployed. This discrepancy is amplified to the level of a problem as now the underlying hardware, which, while evolving swiftly in resources and performance, has remained basically unchanged in programability and architecture, has now started to evolve and further diversify as multi-core CPUs and special purpose GPUs have become widespread.

A change in the underlying architecture naturally does not go without repercussions on the style of programming, which can be implemented efficiently. Of course vendors and other developers make an effort to provide compilers and libraries to reduce the inconvenience resulting for users and developers alike. To be effective, however, it is required to make use of the provided libraries and adapt the style of programming to such an extent that the compiler is provided with sufficient information to perform adaptations and optimizations for the target. These requirements are hard to meet in a highly conservative environment which often relies on libraries and interfaces from the '70s of the preceding millennium, such as the Basic Linear Algebra Subprograms (BLAS), and continues to develop using a procedural programming.

The adoption of newer strategies and methods of tackling a problem more often comes by experiencing need for change. The continued push to increasingly intricate modelling of physical phenomena and the resulting demand for concise, abstract yet highly efficient descriptions alone is sufficient to provide ample incentive. The equations of evolution of physical systems are investigated for the classical case as well as for quantum mechanical settings with respect to their underlying structures. These structures should then provide a guide for a modelling with software components.

Organization

A fundamental part of scientific computing is related to the transformation of physical field problems into discrete mathematical problems, which can then be solved on digital computers. This transition is mirrored in a two part organization of this work.

Before proceeding in elaborations about scientific computing, a short outline, what the term should be understood as, is provided in order to reinforce foundations upon which the remainder is built. Scientific computing aspires to complement the classical methods of scientific investigations, namely theory and experiment. It can accomplish this by applying the facilities made available by digital computers to the fields of scientific theories. Since this procedure can be ap-

[4] Perhaps overcompensation is an explanation, why computer science uses science in its own name.

plied to virtually every science whose theory can be modelled using mathematical methods the requirements are very disparate due to the vast field of problem domains. Thus, it is imperative to have flexibility and reusability in mind, when dealing with the field of scientific computing, in order to prevent a fracture of the relatively young field into several distinct disciplines. This issue is further complicated as scientific computing is applied not to the easiest of tasks, but focuses on problems which result in vast amounts of data and high computational needs, thus efficiency in terms of computational resources can obviously not be neglected. Flexibility is again called for in order to make adaptations to changes in available resources as easy as possible.

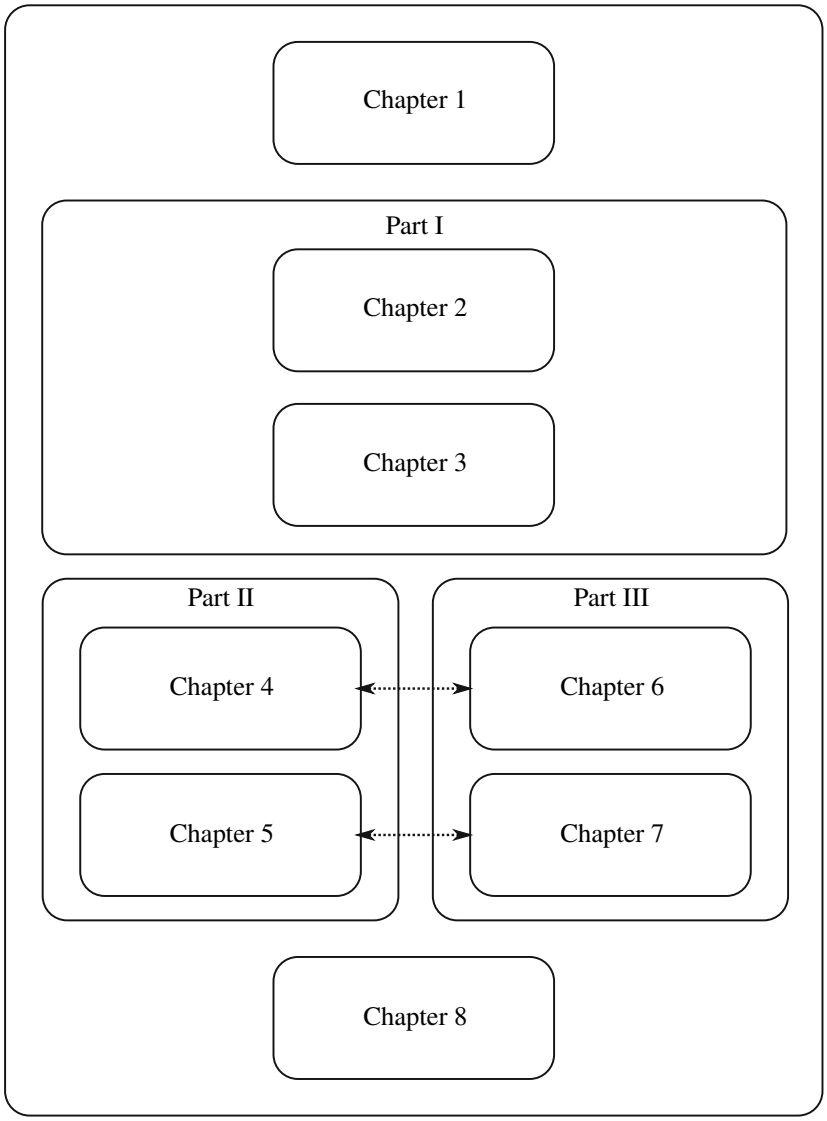


Figure 1.1: Organization of chapters and parts.

The thesis is subdiveded into 8 Chapters. The general structuring of chapters is illustrated in Figure 1.1. Chapter 2 establishes the value of abstraction necessary for efficient scientific computing and prepares for the contributions and results of this work. Chapter 3 gives an outline of the options available for implementations. It furthermore motivates the choice of programming language used throughout the remainder of the thesis. The chapter concludes giving an illustrative example application using generic programming.

The main bulk of this document is split into two dual parts. While the two parts are distinct, they are linked to one another by being designed to mirror each other regarding topics and concepts of the contained Chapters.

Part II contains Chapter 4 and Chapter 5 which provide theoretical background for further exploits. In particular Chapter 4 introduces terms and definitions, and the overall mathematical apparatus, which forms the intellectual backbone. It is the result of a refinement of information acquired in a hunter-gather style from various sources to produce a self contained and consistent compilation, which had not been available in the context of scientific computing. Where Chapter 4 deals with the bare mathematical structures, even when already hinting at applicability, Chapter 5 concerns itself with the establishment of models from a physical perspective. It already draws on the mathematical formalisms previously made available and is thus relatively concise. Section 5.7 may be of special interest, as it represents a departure from the standard introduction of quantum mechanics in order to better illustrate similarities between the classical and the quantum case.

In contrast to these theoretical aspects, Part III deals with methodologies of utilizing them in order to create the abstract realizations of implementations. Chapter 2 not only reiterates basic motivating ideas, but also describes caveats when dealing with abstractions. The chapter finally presents the idea of using generic programming guided by mathematical concepts to organize implementations for scientific computing.

Since the chapters of Part III are able to draw on a firm foundation of terms and definitions from Part II, they can be kept conveniently concise, thus avoiding unnecessary distractions. For ease of association Chapter 6 and Chapter 7 follow the outline presented in the chapters of Part II, as indicated in Figure 1.1. Thus, Chapter 6 shows examples of realizing the mathematical concepts of Chapter 4 mainly using the generic programming paradigm in the C++ language. Where Chapter 4 is the backbone of the thesis as a whole, Chapter 6 is the structuring backbone of the implementation examples of Chapter 7, which reflect implications from the theoretical Chapter 5. As such, where the examples of Chapter 6 represent instances of mathematical concepts to form components, Chapter 7 assembles components and applies them to the problem domain of physics.

At first the focus is on examples illustrating how basic components are combined to form applicable modules for physically motivated sub-problems before applying basic components to examples providing illustrative results. The final sample application combines not only the previously introduced software components, but also is physically challenging, where the other examples have been kept simple not to detract from implementation issues. Chapter 8 concludes with attempting to turn to the gloom yet to come.

Having established the basic field in Chapter 3, the executing part on computing machines and the rising of abstraction by algorithms and mathematics, Part II not only introduces the necessary foundation for strict and formal mathematical structures to formalize the modelling of algorithms, but also consolidates different areas of mathematics. Own contributions are reflected by refinement and convergence of available structures and methods, to provide compatible, realizable, and effective abstraction levels in Part III, which translates the consolidated concepts from Part II to the area of discrete and finite resources present in digital computers.

Chapter 6 initiates the translation of mathematical structures into applicable components and contributions to a software environment. Several data structures are presented to model topological

spaces and fiber bundles as well as the application of geometrical properties. Special consideration is placed on alternate methods for integration in Section 6.6. Applications of Chapter 5 are outlined in Chapter 7, which also presents additional contributions of this work. In particular, Section 7.1 discusses dynamical systems due to Boltzmann's equation in a setting of differential equations and an important area of solid state physics, semiconductor analysis, is further analysed regarding macroscopic descriptions of electrons.

In contrast to the local setting of Section 7.1, Section 7.2 derives a direct approach to use the phase space and its geometry applying the integral form of Boltzmann's equation as introduced in Section 5.6. Additional contributions are then presented by casting sections of trajectories through the phase space and enabling trajectory transitions. Application examples are demonstrated by using modern programming techniques to not only enable the integration of already existing and well-tested code, but also allowing the seamless transition from already developed applications to new applications, thus enabling the use of modern hardware features such as parallelization and GPU computing.

Where previous parts of this work regarding implementations deal with classical settings, Section 7.3 alters the discussion to non-local coherent phenomena, contributing an alternate, scattering induced, correction to quantum mechanics (Section 5.7). Finally a resonant tunnelling diode example is presented in Section 7.4 to show approximations to quantum effects in semiconductor devices.

Part I

Common Ground

Chapter 2

The Value of Abstraction

So che molti diranno questa essere opera inutile

Leonardo Da Vinci

Abstractions are with us every step of our every day lives, where they usually go unnoticed, at least until a boundary is (b)reached. Numbers are among the most widely employed abstractions taken for granted today. However, numbers alone do not carry sufficient information without contexts, as a very simple example of how little bare numbers are helpful can illustrate, which is known to all who have travelled or communicated beyond their habitual time zone. Given only the information something is to happen at 11 o'clock is not helpful. Only when also supplying the notion which timezone it corresponds to, even if simply given as "localtime" along with a location, is the numerical value actually useful.

2.1 Abstractions and the World

To further introduce abstractions it is simplest to examine a few historical developments of the abstractions involved in modelling space which is the omnipresent backdrop of any model of reality. The nature of space has been under debate for considerable time, dating back to classical antiquity. While the deeper mysteries of the true nature shall not be investigated here, the procedure of constructing structural models using mathematics shall be touched upon. René Descartes, also known as Renatus Cartesius, formalized, building on the work of his predecessors, the description of space using the concepts of what we now call Cartesian coordinates, which are constructed by employing the Cartesian product (Definition 3) of numbers. This seemingly simple abstraction allows to associate space, or a portion thereof, to be associated with numbers. This, highly geometric, construction then allows to examine the relations and the evolution of objects in space using the obtained numbers. Every component of the Cartesian product needs to be considered individually, for as long as not the notion of vector spaces (Definition 16) and vectors is established as a powerful entity which allows for powerful yet concise expression. This step should not be underestimated, as simply aggregating numbers to form a new entity, since not every collection of numbers results in a vector, even when the reverse is true that an n-dimensional

vector may be represented using n numbers, which are to be understood to be in reference to a given base of the vector space.

The described abstractions are easily compatible and were even driven by geometrical concepts as introduced by Euclid, hence naming such a space Euclidean. However, the consequent acceptance that our world is not flat, already reveals that this Euclidean concept of geometry is insufficient for larger scales, even if it performs marvellously in a local setting. An abstraction capable of carrying this structure was found in manifolds, since they by definition satisfy the local constraint, yet offer the flexibility required on a larger scale. It also explicitly shows how entities or concepts which were previously considered to be constant, need to be generalized in such a fashion, as to conserve local behaviour, while being globally variable.

With the further increase of sophistication of descriptions, which inherently demanded the use of manifolds, the study and classification of these constructs became a subject of interest. Following the seemingly ageless expression "divide et impera", it proved prudent to decompose the previously opaque entity into geometric and topologic components. Such a decomposition should not be mistaken for the crude and brutal dissassembly of a vector into its components, since it aims to reveal previously hidden structure. The introduction of topology allows for a powerful and expressive classification of spaces, again by mapping to numbers, e.g., the Betti numbers.

The analysis of the structures of topological spaces (Definition 29) and manifolds (Definition 35) also spawned the concepts of fibrations (Definition 39) and fiber bundles (Definition 40) in particular, which provide a methodical means of constructing structures of increasing complexity from rudimentary building blocks. The fiber bundle for instance defines a simple attachment of virtually arbitrary quantities to a basic topological space in a structure preserving manner. Thus not only can manifolds be expressed using fiber bundles, but it is furthermore possible to also express quantities defined on these manifolds as fiber bundles again. Relations between different quantities as well as their evolution are then expressible as maps between fiber bundles. It is for its very basic yet powerful nature that it can be considered as the main abstraction for the definition of data structures in a digital computer in addition to its value in other fields.

2.2 Implementations

The question may arise: why bother with complexities of theory, when a "just implement it" approach seems to suffice? Here it is essential to note that while an implementation is certainly possible with only a minimum of understanding regarding anything of the underlying problem, the implementation will follow exactly these minimal lines of understanding of the implementer. Following the principle of Occam's razor [1], the implementation is bound to be crude and primitive, which in itself is not an undesirable quality if this simplicity is accompanied by easy readability and maintainability. The problem begins to arise, when alterations, however small they may initially appear, to such an implementation are required, which transcend the initial understanding of the programmer. The resulting rigorous redesign incurs the problem that any shortcomings (bugs) already addressed in the previous version may be reintroduced in the new incarnation. While the knowledge of the theory governing the models being implemented can

[1] "entia non sunt multiplicanda praeter necessitatem"– Attributed to William of Occam, an English Franciscan friar (approx. 1288 – 1348 AD), but probably predates him. While this demands simplicity, it does not inherently provide a measure for simplicity, thus leaving considerable freedom of choice.

not be regarded as a guarantee that no redesigns become necessary, since this would also indicate a stagnation of the evolution of programming techniques, the lack of such information is bound to result in erroneous, because uninformed, design decisions.

By accepting theories at a high level of abstraction for implementation, more information is accessible on the partitioning of algorithms into modules and components. The lower the level, the less choice is available or required, depending on the point of view. From a point of superior knowledge, which is in theoretical aspects offered by a higher degree of abstraction, complications arising on a purely low level description often in a difficult to classify manner may be avoided or at least quickly identified.

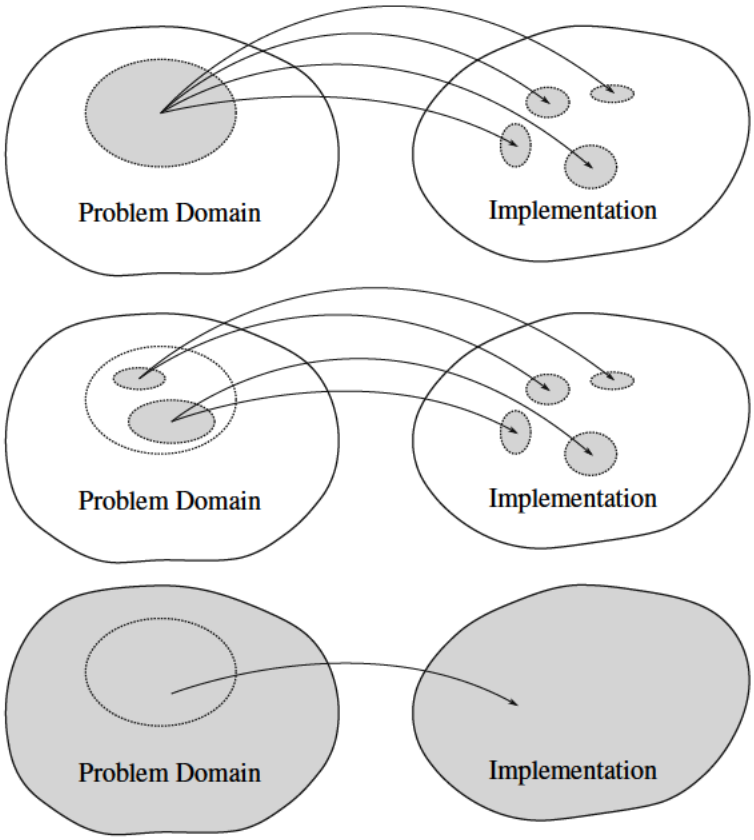


Figure 2.1: Different levels of mapping between the problem domain and the implementation.

Figure 2.1 illustrates different procedures to transfer a problem from its theoretical domain to an implementation. The marked areas indicate the facilities available and used in each of the different cases.

The top most part of the figure indicates a problem with a well defined theoretical domain, where the implementation can not maintain cohesion as a single entity, but shatters into different, distinct entities. This is a common approach for straightforward implementations using a low level of abstraction. An abstract problem is squeezed directly into machine abstractions including low level information such as memory layouts and architecture specializations. The resulting code is tailored to fit exactly one problem on one platform / architecture; reusability is not considered in such an approach. Examples for this mapping include vectors and tensors which are directly mapped to arrays and matrices.

In the center part of Figure 2.1, the theoretical description itself is split to reduce the level of abstraction, and again mapped to a low level implementation. This can be seen as the low level implementation infecting the theoretical setting to reduce the level of abstraction. Hence the overall level of abstraction is even lower than before, since abstractions are sacrificed on both sides and consequently less of the available capabilities can be utilized. Continuing the previous example, this situation occurs when theory is no longer formulated in terms of vectors and tensors, but torn apart into components in an ill-conceived attempt to make implementation easier.

The bottom figure indicates the goal advocated here and which should be strived for. It aims to make the whole instrumentarium of abstractions in the theoretical domain available by embedding the problem at hand into a larger context. At the same time the implementation is not a priori restricted, as a mapping of the extended theoretical setting is strived for. By embedding the task at hand into the bigger setting on both sides, more choices regarding implementation are made available or at least no venues restricted. The procedure does not preclude to specialize any chosen implementation, as long as it fits into the available abstractions. Thus, abstractions increase the number of choices and hence flexibility. On the other hand, it requires the effort of understanding the problem in its context using a minimum of abstract thought. This also results in a responsibility to choose from the offered selection, which may be perceived as a burden. Again picking up the previous example of vectors and tensors, the choices are no longer as clear cut as before. The theoretical setting now concerns itself with the questions of what a vector or tensor is, what concepts the entities model, such as the algebraic structure (see Section 4.2) of tensors and can also easily encompass specializations such as symmetry considerations. At the same time low level concerns of implementation, such as memory layout or endianness, are no longer the main focus, such that, e.g., vectors may be represented by lists, arrays, or whatever data structure can meet the requirements imposed by theory. The resulting increased amount of choices allows to select different specializations for different contexts of the same implementation, such as a change of the underlying hardware. In this fashion generic programming is not primarily concerned with a single solution, but in providing many easily interchangeable components, which allow for flexibility and reusability of any developed code. The price here of course is a higher investment into the abstraction and design process, which while often frowned upon, is exactly in line with the mantra of top down design.

2.3 Obstacles

An ancient and from the current perspective extremely simple example of such a case is presented by the contrast of the Ptolemaic [1] world view compared to the Copernican [2] order of the planetary system. The Ptolemaic setting had found descriptions for many of the movements of celestial bodies. Although the paths of the planets were more complex than in the Copernican setting, the simplicity of proposing a flat world, which was apparently evidently verified in every day observations, was difficult to overcome. Only as an increasing amount of evidence of faults in the model were uncovered by increasingly accurate observations which the theory could no longer accommodate despite the greatest efforts, was the model which had also been elevated to a quasi dogmatic position replaced. A similar shift occurred with the abandonment of aether theory [3] and Newton's view of the world (Section 5.1) in favour of the theory of relativity [4], which also appeared more complex initially, but proved superior in describing available data.

While nothing as world view shattering as just described in the example is aspired to here, it

should serve as an illustration that complexity is not necessarily an absolutely or locally assessable quantity, but may need repeated reconsiderations from different points of view.

Platon's famous parable of the cave [5] provides an excellent means of illustrating the procedure of employing abstractions, although it in itself is an abstraction. We assume the existence of an objective 'real' world whose shadowy images we perceive. From these images we construct patterns of thought, our abstractions. If the construction is well organized using clear definitions, it can, within its boundaries, be checked for consistency, which amounts to checking it for correctness. The abstract model or theory can be used as a guide to predictably influence reality, if it indeed captures the aspects of reality. This has two aspects, firstly, the design and construction of machines as in the field of engineering. Secondly, experiments can be constructed to gauge applicability of abstractions. This is, at times, seen as a validation or proof of a theory, which however, is easily exposed as a fallacy, when realizing that any such test is again restricted to dealing with shadowy images instead of the reality behind them. Thus, by no number of examples or applications of a theory, which are found, can it be considered as proven or correct. Relying on empirical methods of proof too stringently is even detrimental to further development as it may disqualify alternative models prematurely. On the other hand it is possible to invalidate a theory, by finding a single event not matching the abstraction. While this may understandably be an unsatisfying situation, the search for results calling a theory into question is driving progress more than any number of positive results beyond the first, which shows it is applicable at all.

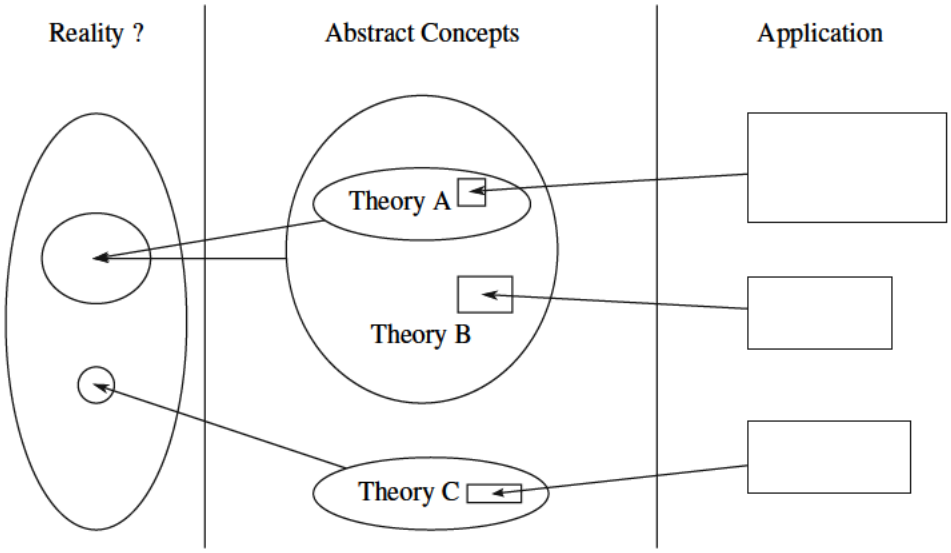


Figure 2.2: Reality, concepts and applications.

Figure 2.2 is an illustration of how theories, based on abstract concepts attempt to map out the elusive thing we call reality, are used for applications. Several different theories may attempt to explain the same aspect of reality, but one is a subset of the other. Examples for such a case are classical mechanics (e.g., theory A in Figure 2.2) and quantum mechanics (e.g., theory B in Figure 2.2), where classical mechanics appears as a subset of quantum mechanics. Many important applications, as simple as levers or as complex as clockworks, have been realized using purely the theory of classical mechanics, without the need for quantum mechanics. By utilizing quantum mechanics new applications, such as in the field of solid state physics or lasers

become feasible, which would not fit into the setting of classical mechanics. It is important to note, however that different theories do not need to be compatible with one another, such as is the case with quantum mechanics and general relativity (e.g., theory C in Figure 2.2). Thus, abstractions should be seen as important and powerful tools, which can be used to realize various applications, without being able to claim to provide a comprehensive model of reality.

From the ability to describe reality, the common misconception arises that by finding examples of any given theory, proof can be extracted. However, while examples are very useful to illustrate a theory, to make it more graspable and are therefore invaluable in text books, they cannot provide anything in terms of proof, if not every possible, even if improbable, configuration is covered by the supplied examples. Since this is not feasible, at the very least in any reasonable amount of time, in the setting of natural sciences, examples can be regarded as merely illustrative, when regarding proof.

A very understandable mistake, when dealing with abstract constructions such as any kind of theory, regards the issue of examples. Examples are often seen as being essential for the validity of scientific theories, when in reality the opposite is more natural. No matter how many examples are presented in support of a theory, it still cannot change the fact that a single example is all that is required to refute a theoretical construct.

Even the world of digital computing, which is constructed completely from logical components, has grown in complexity to such extent that in vivo proofs are hard to come by.

A further observation can be made regarding the evolution of accepted abstractions. Usually, only in the presence of experimental evidence pointing to faults of the old model, a new model may be considered. While this is a natural manner of procedure in natural sciences which attempt to explain the perceived physical reality, it also illuminates a problem in the field of scientific computing, since the term evidence becomes a much more fluid concept. Of course certain measures and benchmarks can be constructed to evaluate the relative affinity of different approaches and abstractions, to certain settings. But to consider this a solution rather than a problem is to deny the rapid, ongoing development of the target platforms. Thus abstractions, which may have been a great boon, may turn into a bane almost without notice [2]. Thus, since true invariants seem difficult to procure, and since the development and implementation of computer programs and algorithms creates the world which should be measured and is hence already an expression of the developer, in the same manner painters add interpretation even to portraits or still lives, the preferences of a developer will be the final judging element.

This situation can be illustrated when looking at old implementations. Even if it has been created with the utmost of skill and technique available at the time of its creation, it does not incorporate developments made, since the code has been written. Thus when new opportunities and choices are later made available, they are not compatible with old code. From the point of view of the initial developer, the newly added features may seem superfluous, since everything can already be expressed using the old methods. This is exactly the same perspective as a supporter of the Ptolemaic view of the world takes in a local setting, everything works, why change and complicate matters. While empirical observations shattered peoples' trust in the Ptolemaic model the same is not as easily available in the realm of programming paradigms, since, as long as the compared variants are Turing complete, a problem is solvable in either, at least in principle. Thus simplicity, elegance or some form of performance measures are presented to make a case for the

[2] An example may be presented by comparing the operations favoured by CPUs and GPUs; or comparing different processor architectures and the use of registers and memory as whole.

use of any programming paradigm. But as simplicity and elegance are subjective matters and performance always depends on the architecture of the underlying hardware, which is subject to changes (definitely not Galilean), it is an illusion to claim hard facts as being decisive.

2.4 Guidance

The example of abandoning the aether also points out that only by the use of available mathematical foundations was it possible to construct "advanced" theories such as the theory of relativity in an acceptable fashion. The formalism alone does not enable to formulate new or better theories, but it is up to the skill, often intuition and persistence of the researchers to find a fitting formulation of their visions. This also points out a limitation of digital computers, since, while they can be programmed to execute an overwhelming complexity of algebraic calculations, they have no intuition or common sense to guide these operations. It is arguable that for this reason it is beneficial to provide mechanisms or frameworks which support the enforcement of algebraic structures. The availability of high level algebraic structures can then aid researchers by being able to point out errors as soon as possible instead of deferring to a low level or not indicating an error at all, but silently calculating an erroneous result.

The main issue here is that abstract theories are further realized on an abstract media. For, while the machines used for computations definitely have a physical reality, programs executed on digital machines are as purely abstract as the values they compute. This lack of substance makes it difficult to properly gauge the accuracy of obtained results. While results are often evaluated from an application point of view, such as physics, the implementation may in fact be flawless in its translation of an idea, but the idea itself may break with the construct of abstraction. Then it follows in an "ex falso sequitur quodlibet" fashion that the result, if it meets expectations or not is in fact useless, since it is borne more of (false) hopes than systematic analysis. Thus, the importance of a clear understanding of terms and definitions, and their inherent limits of applicability cannot be stressed enough, since it is enormously underestimated in a world which is heading for the mediocre goal of "good enough computing", where compromises in realization are introduced prematurely.

A means to prevent the ensuing chaos by using abstractions is schematically shown in Figure 2.3 inspired by the well known process of protein synthesis using DNA/RNA. An abstract theory is expressed clad in mathematical terms and definitions, as is customary in natural sciences, as is depicted on the left hand side of the Figure. Similarly concepts and their various implementations also have mathematical expressions. When a concept corresponds to a requirement in the theory, a fitting implementation can be selected. By relying on generic implementations, the individual components can be combined easily in the necessary configuration. Similar to the biological role model it is not just a simple matching, but depends on the surrounding environment and constraints. The constraints, such as performance guarantees, restrictions on computational complexities, or memory restriction select which particular implementations are made available, while the environment in the least form of a compiler and its accompanying tool chain then fuse the selected components into a working application. Thus, the same theory can give rise to different implementations, as long as they are conceptionally equivalent, thus falling into the same equivalence class (Definition 9) with respect to the imposed demands.

The next chapter is dedicated to providing a possible classification and categorization of available

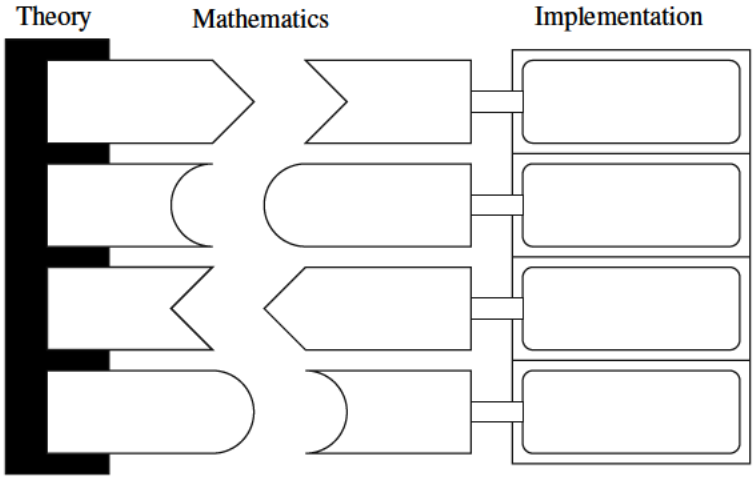


Figure 2.3: Theory and implementation can be elegantly linked using mathematics, in a manner similar to DNA/RNA.

implementations, especially when dealing with the base components, as well as presenting novel implementations as the mathematical sophistication is increased and thus represents a major contribution.

Chapter 3

Machines for Computations

> La machine, qui semblait d'abord l'en écarter, [...] soumet [l'homme] avec plus de rigueur encore aux grands problèmes naturels.
>
> Antoine de Saint-Exupéry

The main stream computing machines we employ today operate in an imperative fashion. The commands are realized using physical principles. The most common physical representation today makes use of electronic circuitry, however, alternative representations have also been employed in the past such as mechanical systems [6][7] and are explored for future developments ranging from optical to quantum systems. An outline of abstraction in a digital computer is illustrated in Figure 3.1.

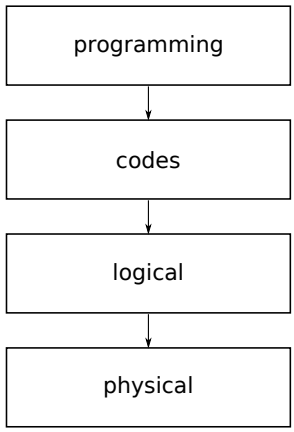


Figure 3.1: Abstraction.

A logical layer is built on the foundations of the physical representation of choice. While again several choices of logic, such as ternary, are available binary logic is used exclusively in the main stream. While various physical representations may yield themselves to varying degrees

to different logical representations, it seems clear that any deviation is considered exotic and is measured with respect to the standard binary system.

Once the simplest of logic operations are made available, it is possible to construct operations of increasing complexity and expressivity leading to the appearance of machine code and the assembly language, which itself is the firm foundation on which tools such as compilers [8] rely. The case of interpreted languages only adds an intermediate layer of a run time environment. Any and all additionally available abstractions in higher level programming languages must lead to a form expressible on this level.

This also reveals that, while a projection from a high level to a lower level must exist for a computer program, it is, in general, not a bijection. This is easily apparent by considering that different expressions even in different programming languages may result in similar or even exactly the same machine code, as can be illustrated using the following, arguably trivial example:

Simple program in C.

```
int main()
{
    float sum = 0;
    int i = 0;

    for (i =0; i < 10; ++i)
    {
      sum = sum + i;
    }
}
```

Simple program in Fortran.

```
PROGRAM HELLO

REAL SUM = 0
INTEGER I = 0

DO I = 0, 9, 1
    SUM = SUM + I
ENDDO

END PROGRAM
```

The central part of the two trivial implementations in Fortran [9][10] and C [11][12], `sum = sum + i`, results in this machine code using a gcc 4.3.4 compiler [13] on an Amd64 architecture without optimizations.

Assembly from C.

```
sum = sum + i;
   cvtsi2ss -0x4(%rbp),%xmm0
   movss    -0x8(%rbp),%xmm1
   addss    %xmm1,%xmm0
   movss    %xmm0,-0x8(%rbp)
```

Assembly from Fortran.

```
SUM = SUM + I
   cvtsi2ss -0x10(%rbp),%xmm1
   movss    -0x4(%rbp),%xmm0
   addss    %xmm1,%xmm0
   movss    %xmm0,-0x4(%rbp)
```

As can be observed the stream of instructions is identical, only the memory addresses do not agree. This should demonstrate that the projection to a lower level of abstraction from several points of origin may produce the same result. However, it also follows from this that simply examining the low level representation it is not possible to reliably reconstruct the source. Although assembly code may contain patterns and peculiarities, which may be connected to a certain compiler or language, the reconstruction of higher abstraction levels is a quite tedious task. This is especially so, as while it may be reliably discerned what is happening, the semantics, as for example, expressed by the naming of variables, is lost. Therefore the reconstruction of an algorithm

from a bottom up method, especially once an optimizing compiler is employed, may be deemed an endeavour doomed to fail. Any information lost by a projection downwards is irrecoverable.

When the generated resulting assembly instructions are identical, which of course greatly depends on the quality of the translation mechanism used, it indicates that the choice of programming language and paradigm does not necessarily affect the generated result. It does, however, very strongly affect how easily and safely different tasks may be expressed and performed. As an example, the same task as above is performed in the following snippet of C++ code.

Sample program in C++.

```
int main()
{
    float sum;

    for_each(counting_iterator<int>(0),
             counting_iterator<int>(10),
             ref(sum) += _1);
}
```

Therefore, the choice of programming language should take into account, how easily a chosen paradigm can be expressed, while also being mindful of the availability and quality of the associated tools such as compilers or interpreters. However, finally, all technical facts aside, the choice usually depends most heavily on a programmer's preference.

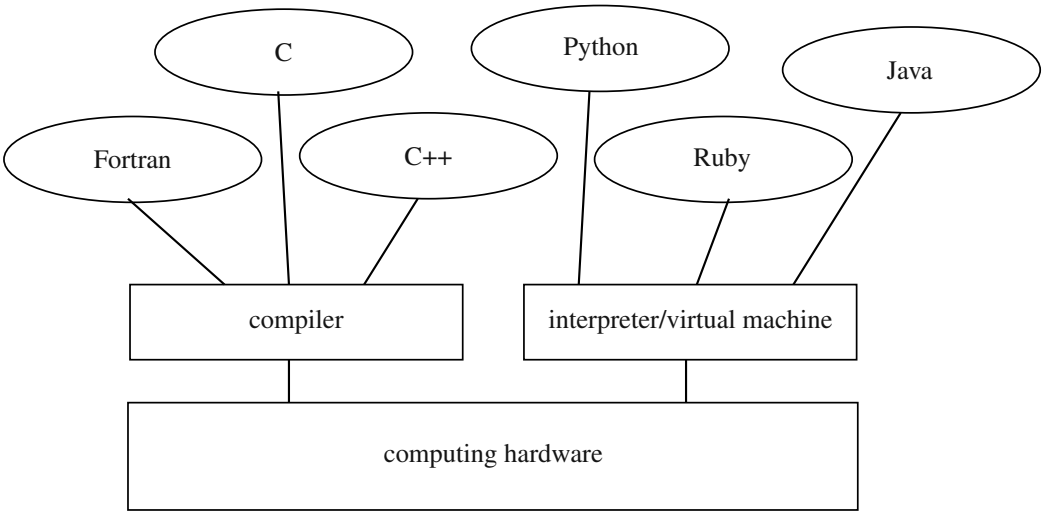


Figure 3.2: From programming languages to execution.

The notion is shown in Figure 3.2 using a by far not complete set of programming languages.

Now that it has been established that all programming abstractions require some form of low level representation, several of the major programming paradigms which embody these abstractions are presented.

3.1 Paradigms

Higher level programming languages may permit higher levels of abstraction, thus easing expression of algorithms. The programming paradigms which have emerged over the years have shaped and influenced the development and evolution of programming languages to a great degree. The paradigms examined more closely here are

- Imperative programming
- Object-oriented programming (OO)
- Functional programming (FP)
- Generic programming (GP)

3.1.1 Imperative Programming

Imperative programming may be viewed as the very bones on which all other abstractions depend. This programming paradigm uses a sequence of instructions which act on a state to realize algorithms. Thus it is always specified in detail what and how to execute next. The modification of the program state, while convenient is, also an Achilles' heel, as with increasing size of the program unintended modifications of the state become an increasing problem. In order to address this issue the primitive imperative programming method has been refined to procedural and structured programming paradigms, which attempt to provide more control of the modifications of the program state.

Even in the refined forms the incurred overhead can be limited to a bare minimum as the level of abstraction is relatively low. This was well suited for the situation of scarce computing resources and the lack of mature and powerful tools. Under these circumstances the overall performance, in terms of execution speed or memory consumption is solely dependent on the skill and ingenuity of the programmer, which has resulted in the almost mythical "hand optimized" code.

However, to achieve the desired specifications in such a fashion, the clarity and readability, and thereby the maintainability of the code were sacrificed. Furthermore, the low level of abstraction also hinders portability, as different architectures favour different assumptions to produce efficient execution. To address this effect, implementations were duplicated in order to optimize for different architectures and platforms, which of course makes a mockery of goals such as code reduction or even reuse. The desire for a reuse of code led to further evolution of the idea of structured programming.

3.1.2 Object-Oriented Programming

The object-oriented paradigm [14] may be viewed as an evolution from the structured imperative paradigm. It, on the one hand, addresses the issue of unchecked modification of state by enforcing data encapsulation, thus enforcing changes through defined interfaces. On the other hand it tries to address the issue of code reuseability by providing the mechanism of inheritance.

Both of these notions are attached to an entity called an object. Therefore an object serves as a self contained unit which interacts with the environment via messages. In this fashion it accomplishes a decoupling of the internal implementation within the object and the interaction

with the surrounding environment. By this construction object-oriented programming enforces interfaces, which is essential for modular programming. The algorithms are expressed much more by the notion of what is to be done as an interaction and modification of objects, where the details of how are encapsulated to a great extent within the objects themselves.

As already mentioned, the concept of inheritance is added to address issues of code reuse. Inheritance is deployed with the aim of reducing implementation efforts by allowing refinement of already existing objects. Using inheritance and the connected sub typing also makes polymorphic programming available at run time. While the concepts of object-orientation have proved to be invaluable to the development of modular software, its limits also became apparent as the goal of general reusability suffers from the stringent limitations of the required sub typing [15]. Problems in modelling as a hierarchy of inheritance arise as a consequence that objects are not necessarily fit to accommodate the required abstractions, such as in the case of algorithms, which operate on the data structures, represented by the objects, but fail to be classified as objects themselves. Furthermore, the extension of existing codes is often only possible by intrusive means, such as changing the already existing implementations thus not leading to the high degree of reduction of effort as was hoped for.

Compared to the run time environment or compiler required to realize the simple imperative programming paradigm, the object-oriented paradigm's demands more sophistication as it needs to be able to handle run time dispatches using virtual functions, for instance. Additionally, seemingly simple statements may hide the true complexity encapsulated within the objects. Thus not only is the demand on the tools higher but the programmer also needs to be aware of the implications of the seemingly simple statements in order to achieve desirable levels of performance.

3.1.3 Functional Programming

In contrast to the imperative and object-oriented paradigms, which explicitly formulate algorithms and programs as a sequence of instructions acting on a program state, the functional paradigm [16] uses mathematical functions for this task and forgoes the use of a state altogether [17]. Therefore, there are no mutable variables and no side effects in purely functional programming. As such it is declarative in nature and relies on the language's environment to produce an imperative representation which can be run on a physical machine. Among the greatest strengths of the functional paradigm is the availability of a strong theoretical framework of lambda calculus [18] for the different implementations.

Since a purely functional description is free of side effects, it is a favourable choice for parallelization, as the description does not contain a state, which would require synchronization. Data related dependence, however, must still be considered in order to ensure correct operation.

As the declarative style connected to the functional paradigm distances itself from the traditional imperative paradigm and its connection to states, input and output operations pose a hurdle which is often addressed in a manner, which is not purely functional. As such functional interdependences may be specified trivially, while the details how these are to be met remain opaque and as a choice to the specific implementation.

Polymorphism, which has to be especially provided in the imperative world, comes naturally to the functional paradigm as no specific assumptions about data types are required, only conceptual requirements must be met.

3.1.4 Generic Programming

Generic programming [19][20][21][22] may be viewed as having been developed in order to further facilitate the goals of code reduction and reusability. While these are among the goals, which lead to the development of object-oriented programming, it may vary quite profoundly in the realization. A major distinction from object-oriented programming, which is focused on data structures, is that it especially allows for a very abstract and orthogonal description of algorithms. It thus requires a separation of the data structures and the algorithms acting on them, such as iterators.

Furthermore, algorithms are aimed to be specified for as broad a class of use cases as possible. To this end it is necessary to abstract the requirements of algorithms to isolate the minimum collection of conceptual requirements. It is possible to provide various specializations for various use cases. New use cases can be accommodated by adding additional specializations. A mechanism to select a particular specialization from among the various possibilities in a non-invasive manner is also required.

This can be accomplished by employing the art of writing programs which generate or modify either themselves or other programs, which is known as meta programming. Tools belonging to this discipline are ubiquitously applied in programming in the guise of compilers, which have the sole purpose of transforming the description using a programming language to an executable representation. While this is an essential task, it also applies to the discipline of generic programming, where it can supply the selection of a fitting specialization. Meta programming may be made available at run time, when it is also referred to as reflection [23] or at compile time, then fittingly also called compile time programming.

When generic programming makes use of meta programming using compile time facilities such as static polymorphism, not only is implementation effort reduced but the resulting run time performance optimized.

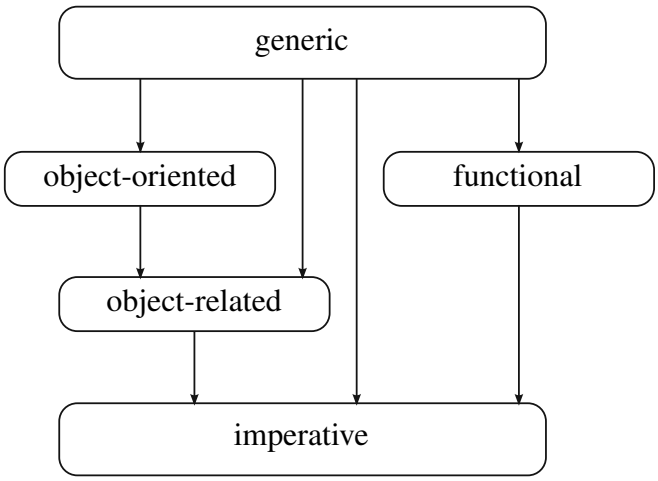


Figure 3.3: Level of abstraction of different paradigms.

When considering the level of abstraction of the programming paradigms, a view as in Figure 3.3 presents itself. Imperative is, by the nature of the machines finally employed, the lowest level of abstraction. Functional as well as object-related or object-oriented bring an increase of the level

of abstraction. The generic approach is free to combine all of the other approaches in order to meet predetermined goals such as reusability or efficiency. The actual choice of which of the paradigms to use for which task in generic programming is driven by setting additional goals, instead of an inherent demand for procedure.

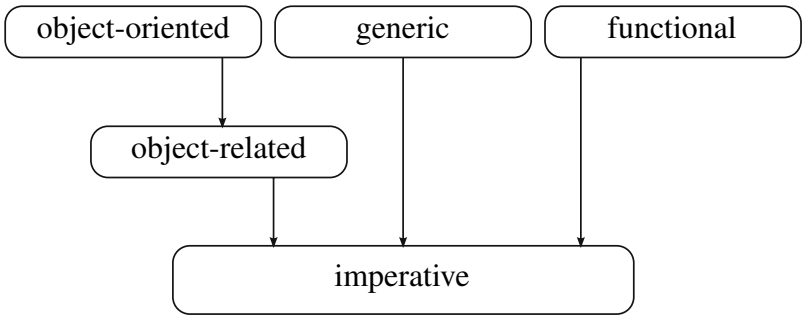


Figure 3.4: Features offered by programming paradigms from a programmer's perspective.

The view changes from a programmer's perspective, if he concerns himself which of the programming paradigms offers a set of features, as sketched in Figure 3.4.

3.1.5 Programming Language for Scientific Computing

A plethora of programming languages has been developed during the history of computers. A language may restrict its facilities to support only a single programming paradigm, such as the purely functional programming languages like Haskell [24] or purely object-oriented languages such as the initial version of Java [25]. It is, however, not uncommon that several paradigms are directly supported in a single programming language, as is the case in Lisp [26] or Python [27].

It is also not uncommon for a programming language, to be retrofitted with features, which had not been developed at the time of its initial conception, such as is the case of Fortran [10], which was extended from its purely imperative beginnings to also include object-oriented features. A programming language supporting the use of several programming paradigms is called a multi paradigmatic language. As the run time performance of the resulting executables is of essential importance in the field of scientific computing, the choice of a programming language to implement algorithms cannot go uninfluenced by this characteristic. The combination of the desires for multi paradigmatic facilities, an open specification and availability of mature tools, considerably restricts the options to choose from.

The field of scientific computing is especially demanding, since it requires not only flexibility and adaptability in terms of specification, while at the same time meeting demands for high speed computations. Since different programming paradigms have different affinities for different problems, a simultaneous combination of several paradigms is a natural choice to meet all the conflicting demands at the same time. Thus a target library or program should incorporate support of as many paradigms as possible.

On the other hand, the different areas of application and their differing requirements together with a desire for high level specification has resulted in the development of domain specific languages (DSLs) with their own grammar and vocabulary. Such a DSL provides a means to

abstract from hardware and even the currently available programming languages. This comes at the price of increased difficulty of interoperability with existing libraries and programs.

One way to deal with this issue of interoperability presents itself by realizing the DSL not in a stand alone fashion, but by directly embedding it in a host language. The new concept, now called a domain specific embedded language (DSEL), provides the benefits of a DSL, the high semantic level, while it also incorporates interoperable access to all the libraries already available to the host language [28].

Table 3.1 shows a comparison of different features in several programming languages. As can be seen from the brief comparison of languages, only a few select languages are available which allow an efficient modelling of operators which are indispensable for the realization of DSELs. Languages such as Haskell were not developed with a focus on high performance, but they still offer the definition of new operators which extend the built-in language syntax.

Language	operator overloading	parametric polymorphism	functors	time of evaluation
C	no	partial [1]	no	compile/run time
D	partial [2]	yes	no	compile/run time
Java	no	partial [2]	no	run time
C#	partial [2]	partial [2]	no	run time
Haskell	yes	partial [2]	yes	run time
C++	yes	yes	yes	compile/run time

Table 3.1: Language comparison.

Performance aspects of application development should be handled orthogonally to the main development of applications. With a multi-language approach performance aspects cannot be easily considered orthogonally because of the use of compiled modules which require an interface layer in order to build applications [29]. The tight integration provided by the use of a DSEL addresses this problem as whole applications may be compiled in a single step. This, of course, puts extensive stress on the tool chain and requires it to have a minimum level of maturity.

3.2 Mathematics and Algorithms

While the machines constructed to perform computational tasks undoubtedly are capable of addressing mathematical problems, it requires special adaptations in order to carry mathematical procedures from the purely theoretical settings to the machine world. The declarative parts of mathematical formalisms pose a problem for the implementation of algorithms. This is true even for very simple mathematical statements such as

$$\forall x \in \mathcal{X} \tag{3.1}$$

which may be encountered for example in preconditions. In the setting of mathematics the statement in itself is sufficient, however, for an implementation some kind of enforcement is required,

[1] using macros
[2] depends on version

if it is not to be just silently assumed. A practical difficulty arises from the fact that the mathematical expression is timeless and does not consume any resources at all beside itself. On the digital computer, however, memory consumption and computing requirements are always a limiting factor.

The limited nature of the computing world also becomes apparent, when comparing the limited numerical data types available to the machine, which may require special attention from the programmer, to the plethora of sets a mathematician has at his command.

A very simple example concerns integers which are commonly denoted as $\mathbb{Z}$ in mathematics. While programming languages offer a data type of exactly the same name, it falls far short from the mathematical entity, with which it shares its name. The most apparent fact is that the data type encompasses only a finite amount of distinct elements, which directly corresponds to the amount of memory it requires, while $\mathbb{Z}$ is infinite and has no resource consumption associated with it. Furthermore, while both cases are groups, the topology of integer data types resembles $\mathbb{S}^1$, as shown in the left part of Figure 3.5 where the minimum and maximum values are next to each other and wrap around, which substantially deviates from the $\mathbb{Z}$ topology.

In the case of floating point data types, such as defined by IEEE 754 [30], which are used to emulate the mathematical concepts of the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$, are at best capable of exactly representing a subset of $\mathbb{Q}$, but fail to capture irrational numbers at all. While the topology of this data type is linear for the numbers it represents, there is also an isolated element `nan`, as is depicted in the right half of Figure 3.5. Furthermore, while being a super set of the integer data type, it fails to be a group.

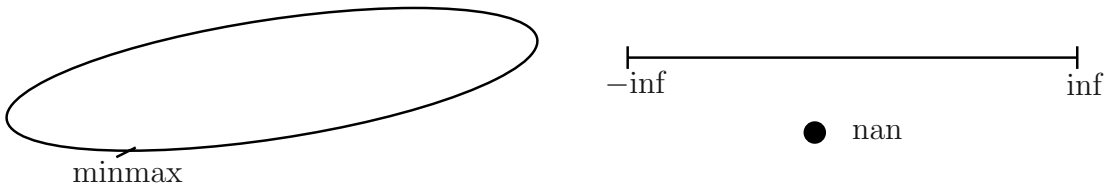


Figure 3.5: Different topologies inherent to data types. The data type on the left wraps around from its maximum to minimum values, corresponding to an $\mathbb{S}$ type topology. The data type on the right even includes an isolated point (`nan`).

While being far from comprehensive, the given examples should indicate that care has to be taken, when considering implementing algorithms on a digital computer. An awareness of a machine's limitations, which may vary considerably due to architecture and platform, is essential for realizing reliable scientific computing applications.

3.3 Example of Generic Programming

To demonstrate the facilities and opportunities of the described methods a short example calculating the value of a determinant of a matrix is presented. While relatively simple in terms of mathematical sophistication, it provides the opportunity to show the procedures and methods of generic programming in the C++ programming language [31][32], which differ quite significantly from those usually prevalent in C++ run time expressions.

The selection of the C++ programming language for use in examples was made according to the criteria provided in Section 3.1.5, starting with the fact that C++ is a multi paradigmatic programming language, since it supports the following programming paradigms [33]:

- Structured imperative – as a direct derivative of the C language
- Object-oriented – it has direct support for inheritance and virtual functions
- Functional programming – using libraries such as Boost Phoenix [34]
- Generic programming – using the meta programming facilities

The currently unique combination of features present in C++, makes it highly suitable for deployment in the field of scientific computing. Furthermore, not only is a mature tool chain for C++ available, but generic programming has been a part of the C++ programming language in the form of the STL [35] form the initial standardization. The underlying design patterns have since been examined [36][37] and elaborated [38][39][40]. Additionally, the mechanisms directly supporting generic programming in C++ have been the subject of investigation [41][42][43] as well as the performance penalties encountered due to abstractions in generic programming [44][45][46][47][48][49][50][51][52]. With the advent of multi-core CPUs, the issue of parallelization has increased in importance and has been shown to be compatible with generic programming in C++ [53][54]. Finally, generic programming in C++ has already been successfully applied to scientific computing [55][56][57].

Concepts for scientific computing have already been impressively demonstrated as applicable for topological frameworks especially for the use in numerical schemes to solve partial differential equations due to physical problems [57]. Building on such a firm base and since examples are already broadly available, this thesis attempts to investigate the interaction and interrelation of the realm of topology and geometry. A particular contribution of this work deals with the physical phenomena between both classical and quantum descriptions using integral descriptions, which are connected to the geometric aspects found in theory. While this work shares fundamental components of topology and their implementation, it is the abstract geometrical requirements which are a driving force for the investigation and application of programming concepts. As source code is much more easily constructed than ideas and thought patterns, it is the attitude with which to approach and solve problems which is of primary concern and which has been further developed.

Since the implementation makes use of both, run time and compile time structures, special consideration of elements crossing the border between these two worlds is required. To make efficient use of the facilities provided by meta programming, generic programming methods necessarily need to cross this border, as every of these points of evaluation embosses their respective advantages and disadvantages on the executable. Only their compound use can converge to optimal use.

3.3.1 Mathematical Description

Determinants have several applications from orientation tests to eigenvalue problems. A very simple algorithm to calculate the value of a determinant, is to apply Laplace expansion to the determinant. While it is not the most efficient algorithm, it is well suited to demonstrate the use of a generic C++ program incorporating template meta programming techniques, due to its recursive nature. For completeness the algorithm is outlined shortly before showing an implementation.

Given a matrix

$$A = \begin{pmatrix} a_{1\,1} & \dots & a_{1\,n} \\ \vdots & \ddots & \vdots \\ a_{n\,1} & \dots & a_{n\,n} \end{pmatrix} \tag{3.2}$$

its determinant $\det(A)$ can be calculated by breaking it down into a series of smaller determinants. To this end a column j is chosen to arrive at the formula

$$\det(A) = \sum_{1}^{n} (-1)^{i+j} a_{i\,j} \det(A_{i\,j}) \tag{3.3}$$

The matrices $A_{i\,j}$ are minors to the original matrix A, meaning that the rows and columns indicated by the indices are omitted to obtain $A_{i\,j}$. This procedure can be performed recursively until the determinant expressions are already known, such as is the case for 3×3 or 2×2 matrices or even the trivial case of a single value. The final expression is then of a form

$$\det(A) = \sum_{\sigma} \left(\operatorname{sign}(\sigma) \prod_{i} a_{i\,\sigma(i)} \right), \tag{3.4}$$

where σ is a permutation and $\operatorname{sign}(\sigma)$ the sign of the permutation. The recursive nature of the construction of this sum of products makes this algorithm well suited for implementation using the functional style required by template meta programming.

It favours the implementation of an algorithm, how the correct access patterns into the data are to be constructed and how to combine this data. Thus while patterns of access will be completely available at compile time, the actual values, which will be manipulated may be supplied at run time. This distinction offers the compiler the complete set of information of memory accesses and it can therefore do its best to optimize them. However it should not go unnoted that the optimizer may be overwhelmed by the amount of data connected to the calculation as the size of matrices increases. While the consequences depend on the particular optimizer in use by the compiler, a sub optimal solution is the most likely result, with an abnormal termination being an extreme, but not impossible case.

The structural, purely compile time parts shall be given first. It constructs sequences of types, which encode the required operations. These operations are then carried out using a run time adaptor. This results in a clear and complete separation of the algorithmic structure created at compile time and the data structure used to store actual values. The compile time structure in this case is thus equivalent to a computer program written by a programmer by hand, only that

it is constructed algorithmically from the structural rules provided by the mathematical description. The use of the algorithm requires the binding of the compile time prescription to run time values. Since only when combining a run time matrix accessor with the compile time structure an algorithm for the calculation of the value of the determinant is obtained. Otherwise the compile time structure merely implements a means of determining permutations as is evident from Equation 3.4. This fact is also an indication that the description here is for demonstration purposes of a venue for generic programing, instead of considering it for high performance determinant calculations. In this regard it would be much more fitting to use an implementation following Gaussian elimination, LU or QR factorization, which, while still costly, have far less computational complexity than the number of permutations.

3.3.2 Compile Time Structure

It should be recalled that C++ meta programs are completely stateless, making recursion the only option to express repetitions. In the following the `mpl` name space indicator shall be short for `boost::mpl` which indicates the Boost MPL library [58].

Examining the terms involved in Equation 3.3 and Equation 3.4 provides the guidelines for the definition of the compile time structure. The recursion to sub determinants is a means of construction all the required permutations required for the calculation of the determinant. Thus the necessary tasks may be broken down into the following steps

- Determine a mechanism to encode a single element A_{ij}.
- Choose a representation for the multiplication of elements.
- Incorporate the sign associated with a permutation.
- Set an encoding for the sequence of additions.

A short outline of each of these steps is provided before the meta programs for the generation and manipulation are detailed.

Access to any of the elements of the matrix data set is determined by the pair of indices. To encode this in a compile time structure, the indices are encoded into integral types such as `mpl::long_` and encapsulated into a `mpl::pair`. Thus, a matrix element A_{ij} shall be represented by:

Listing 3.1: A pair of indices encoded into an `mpl::pair`.

```
typedef mpl::long_<i> pos_i;
typedef mpl::long_<j> pos_j;

typedef mpl::pair<pos_i, pos_j>::type matrix_element_type;
```

This encoding concerns itself purely with the position ij, not with the value stored at the indicated position.

The multiplication of thusly encoded elements is accomplished utilizing a simple type sequence, the `mpl::vector`.

Listing 3.2: A sequence of pairs representing elements for multiplications.

```
typedef mpl::pair<pos_i1,pos_j1>::type matrix_element_type_1;
typedef mpl::pair<pos_i2,pos_j2>::type matrix_element_type_2;
typedef mpl::vector<matrix_element_type_1,
                    matrix_element_type_2>::type mult_sequence;
```

In order to bond a sign, which is encoded as a integral constant, to the multiplication sequence an `mpl::pair` is used:

Listing 3.3: A sequence of pairs representing elements for multiplications.

```
typedef mpl::pair<sign_type,
                  mult_sequence>::type mult_signed_sequence;
```

The resulting sequences are then collected within an `mpl::vector` which encodes all the necessary operations.

Listing 3.4: A sequence of pairs representing elements for multiplications.

```
typedef mpl::vector<mult_signed_sequence_1,
                    mult_signed_sequence_2
                    >::type determinant_structure_type;
```

Having outlined the means of representation, the means with which to generate these representations is sill missing and is provided in the following.

3.3.3 Compile Time Meta Program

The top level of the compile time program takes the form of a simple struct containing the invocation of the recursive meta program components with the natural order of the indices. It accepts a sole input parameter indicating the size of the data set.

Listing 3.5: Top level of the determinant meta program.

```
template<long size>
struct determinant_struct_sequence
{
 typedef typename mpl::range_c<long,0,size>::type initial_map;
 typedef typename recursive_determinant<initial_map, initial_map,
                                        size>::type type;
};
```

The recursive part is given in the next piece of code. Particularly it is the specialization for two-dimensional data sets, which ends the recursion. It uses the well known prescription to calculate the determinant in this case. To this end the correct mappings are extracted into the types `x0`, `x1`, `y0` and `y1`. The obtained indices are then paired into an `mpl::pair` which specifies the access to a single point within the data set.

The pairs are enclosed into compile time containers, `mpl::vectors`, which encode the multiplication of the pairs it contains. This has the benefit of making the multiplications extensible in a simple fashion. The two possible configurations of the two-dimensional case are represented by the types `first` and `second` for the term corresponding to the main diagonal and the off diagonal terms respectively.

Finally the two types corresponding to multiplication are packaged into another pair, which now links the sign with which the expression is to be evaluated to the multiplications, and assembled into another compile time container which is used to store the additive terms. It is this compile time container which is returned by the meta function.

Listing 3.6: A two-dimensional determinant serves as the end of the recursion of the determinant meta program.

```
template<typename MappingTypeX, typename MappingTypeY>
struct recursive_determinant<MappingTypeX, MappingTypeY, 2>
{
 typedef typename mpl::at_c<MappingTypeX,0>::type x0;
 typedef typename mpl::at_c<MappingTypeX,1>::type x1;

 typedef typename mpl::at_c<MappingTypeY,0>::type y0;
 typedef typename mpl::at_c<MappingTypeY,1>::type y1;

 typedef typename mpl::vector<typename mpl::pair<x0,y0>::type,
                              typename mpl::pair<x1,y1>::type
                              >::type first;

 typedef typename mpl::vector<typename mpl::pair<x1,y0>::type,
                              typename mpl::pair<x0,y1>::type
                              >::type second;

 typedef typename mpl::vector<
     typename mpl::pair<typename mpl::long_<1>::type,
                        first>::type,
     typename mpl::pair<typename mpl::long_<-1>::type,
                        second>::type>::type type;
};
```

Having established how to calculate a two-dimensional determinant as the end of a recursion, the general definition of the recursive component is discussed next. The parameters are the same as in the two-dimensional case, with the exception of leaving the size unspecialized.

The implementation then proceeds by omitting an element from one of the index sequences. The return type is constructed by folding a parametrized functor over the sequence of indices, whose consecutive results are gathered within a compile time container. The types generated by the repeated functor invocations encode the terms which need to be summed for the calculation of the value of the determinant.

Listing 3.7: Main recursion meta algorithm.

```
template<typename MappingTypeX, typename MappingTypeY,
         long matrix_size>
struct recursive_determinant
{
   typedef typename omit_view_c<MappingTypeY,
                                0>::type remappedY;

   typedef typename mpl::range_c<long,
                                 0, matrix_size>::type steps;

   typedef typename mpl::fold<
       steps,
       mpl::vector<>,
       recurse<MappingTypeX, remappedY,
               typename mpl::at_c<MappingTypeY,0>::type>
       >::type type;
};
```

The just described section of code was referred to as being recursive, however no obviously recursive call is directly discernible. The truly recursive nature is revealed by examining the employed functor in more detail. The functor follows Boost MPL semantics for functors by containing a nested struct named `apply`. The additional information of the outside mappings along with the currently discarded index are supplied directly to the `recurse` template, while the nested `apply` is fed with arguments accumulating previous evaluations as well as the current evaluation item during the `fold`.

It is the functor's task to construct determinants for the minors within the `apply` struct. To this end it omits an element from the so far unaltered index sequence, thus creating an expression corresponding to a minor of the given matrix, which in turn is passed to the recursive determinant meta function `recursive_determinant`. The element corresponding to the omitted row and column for the construction of the minor is again encoded in a `pair` along with the appropriate sign, which is also fully determined at compile time using a utility meta function `sign`. This element is multiplied with the result from the evaluation of the minor's determinant, as indicated in Equation 3.3. This multiplication is accomplished by appropriately inserting the signed local element into the compile time container generated for the sub determinant. The final result is obtained by concatenating the current result with the results obtained from previous calls using the `mpl::joint_view` construct.

Listing 3.8: Functor used to obtain the return type of the recursive determinant meta function.

```
template<typename MappingTypeX, typename MappingTypeY,
         typename LostType>
struct recurse
{
  template<typename StateType, typename c>
  struct apply
  {
    static const size_t length =
                        mpl::size<MappingTypeX>::type::value;

    typedef typename omit_view<MappingTypeX,c>::type
                                                reduced_mapping;
    typedef typename recursive_determinant<reduced_mapping,
                                           MappingTypeY,
                                           length-1>::type
                                           sub_determinant_type;

    typedef typename mpl::at<MappingTypeX,c>::type current_index;

    typedef typename mpl::pair<current_index,
                               LostType>::type
                                     local_element_type;
    typedef typename mpl::pair<
            typename mpl::long_<sign<c::value>::value>::type,
            local_element_type>::type signed_local_element_type;

    typedef typename append_to_subsequences<sub_determinant_type,
                                signed_local_element_type>::type
                                sub_multiplied_type;

    typedef typename mpl::joint_view<StateType,
                                     sub_multiplied_type>::type
                                     type;
  };
};
```

The next snippet of code shows a simple calculation at compile time, as used to determine the sign. Note that only simple operators and static const integer types are available.

Listing 3.9: Utility meta function for sign calculation.

```
template<long exponent>
struct sign
{
  static const long value = (exponent % 2) ? -1 : 1;
};
```

The next piece of code shows how intermediate results are concatenated to appropriately encode the desired mathematical operations. The procedure is again split into two components. The top level meta function being:

Listing 3.10: Utility meta function used to correctly combine intermediate result.

```
template<typename SequenceType, typename ItemType>
struct append_to_subsequences
{
  typedef typename mpl::fold<
      SequenceType,
      mpl::vector<>,
      push_back<mpl::_1,
                signed_inserter<mpl::_2, ItemType> >
                >::type type;
};
```

It should not go unnoted that the `push_back` here is a meta function working on the compile time container `mpl::vector` in the same fashion as the run time equivalent would on a `std::vector`. However, where the run time function simply modifies an existing container, the compile time version generates a completely new sequence, due to the immutable nature of C++ meta programming. The `mpl::_1` and `mpl::_2` are named to resemble their run time lambda expressions.

A further utility meta function `signed_inserter` is employed to transform the elements of the input sequence before pushing them into the result sequence, in order to match the chosen logical encoding.

Listing 3.11: Generic algorithm for calculating determinants compile time meta program.

```
template<typename SignedSequType, typename ItemType>
struct signed_inserter
{
  typedef typename SignedSequType::first  initial_sign_type;
  typedef typename SignedSequType::second coefficients_type;

  typedef typename ItemType::first  additional_sign_type;
  typedef typename ItemType::second item_type;

  typedef typename mpl::times<initial_sign_type,
                              additional_sign_type>::type
                              sign_type;

  typedef typename mpl::push_back<coefficients_type,
                                  item_type>::type
                                  extended_sequence_type;
  typedef typename mpl::pair<sign_type,
                             extended_sequence_type>::type type;
};
```

First the input elements, which are expected to be `pairs`, are unpacked. Then the resulting sign is calculated by using the `mpl::times` meta function, which works on integral constant types such as `mpl::int_`, on the extracted individual signs. The element to be added is pushed into the coefficient sequence. It should again be noted that this does not mutate the original sequence in any way, but rather results in a completely new sequence, which contains the original types along with the pushed type. The extended sequence is then repacked together with the calculated sign type to resulting type of this meta function.

All the meta functions so far are sufficient to generate a structure of types, which encodes all the operations associated with a determinant, by providing all the required permutations at compile time. This structural part is in fact completely independent from the task of computing determinants and can be freely reused beyond the field of matrices and matrix data types. Since this part of the algorithm is not tied to the domain of matrices, this implementation, which follows the mathematical description very closely, is consequently completely independent of the matrix data type for which a determinant is to be calculated. It therefore surpasses any plain attempt to arrive at data type independence using a simpler template approach, which attempts to simply make a generic implementation from a procedural implementation by converting it to template functions, since that will implicitly introduce the matrix data type. It is this implicit nature which degrades the level of generality.

The components shown this far solely belong to the compile time regime. In order to enable the evaluation of determinants whose values are determined at run time, additional facilities are required which bridge the run time / compile time border.

3.3.4 Run Time Adaption

The first component of the run time adaption deals with the evaluation of multiplications. The following unary function object is a key component of this endeavour:

Listing 3.12: Function object for the evaluation of the multiplicative sequences.

```
template<typename MatrixAccessorType, typename NumericType>
struct multiplication_sequence_eval
{
   const MatrixAccessorType& matrix;
   NumericType& return_value;
   multiplication_sequence_eval(const MatrixAccessorType& matrix,
                                NumericType& return_value) :
      matrix(matrix), return_value(return_value) {};

   template< typename U >
   inline void operator()(U x)
   {
      typedef typename U::first  first;
      typedef typename U::second second;

      return_value *= matrix(first::value, second::value);
   }
};
```

It is parametrized on the access mechanism to the data contained within the matrix as well as the numeric type indicating the type of the result of the computations. Matrix access is held by constant reference within the object, thus being able to access the data in place in a safe manner. The return value is also used by reference and hence also has to be supplied at construction time.

Evaluation of the result of the multiplicative sequence takes place in the templated `operator()`. The parameter it formally takes is merely a dummy whose value is never actually used. It is used solely for type deduction. The indices for access to the matrix are extracted from the supplied type and fed into the matrix access mechanism. In this fashion consecutive evaluations of the operator along a type sequence enable to evaluation of the multiplication of the matrix elements indicated by the indices. A fact which might escape attention due to the compactness of the specification using the template mechanism is that each individual type within the type sequence will spawn its own implementation of `operator()` specialized to appropriately.

Another unary function object `determinant_structure_eval` is used to not only initiate the iteration required by the `multiplication_sequence_eval` function object, but also to aggregate the individual results of multiplication.

Listing 3.13: Function object for the aggregation of multiplicative terms.

```
template<typename MatrixAccessorType, typename NumericType>
struct determinant_structure_eval
{
  const MatrixAccessorType& matrix;
  NumericType& result;

  determinant_structure_eval(const MatrixAccessorType& matrix,
                             NumericType& result) :
    matrix(matrix), result(result) {};

  template<typename SignedSequenceType>
  inline void operator()(SignedSequenceType X)
  {
    typedef typename SignedSequenceType::first  sign_type;
    typedef typename SignedSequenceType::second sequence_type;

    NumericType local_result(1);

    mpl::for_each<sequence_type>
      (multiplication_sequence_eval<MatrixAccessorType,
                                    NumericType>(matrix,
                                                 local_result));

    result += sign_type::value * local_result;

  }
};
```

The structure of the function object follows exactly the same pattern as in the previous case. The difference lies solely in the computations within `operator()`. The sequence of multiplications is extracted and evaluated using the MPL's sole run time algorithm, `mpl::for_each`, which traverses the type sequence and invokes a supplied unary function object. By using the `multiplication_sequence_eval` function object, the terms indicated in the type sequence are multiplied. Finally, the sign of the current term, which has been extracted from the input data type, is applied in the final addition.

Now that the components have been defined it is possible to define a top level interface for the calculation of a determinant. Casting it into the form of a function object:

Listing 3.14: Top level interface.

```
template<typename DeterminantStructure>
struct determinant_interface
{
 template<typename MatrixAccessorType, typename NumericT>
 inline void operator()(const MatrixAccessorType& matrix_access,
                        NumericT& result)
 {
  result = NumericT(0);
  mpl::for_each<DeterminantStructure>(
   determinant_structure_eval<MatrixAccessorType,
                              NumericT>(matrix_access,result) );
 }
};
```

Here the determinant structure has to be specified explicitly during type creation, while the `MatrixAccessorType` and `NumericType` can be derived automatically from the arguments of the `operator()`. This allows for the reuse of the function object for matrices of the same size, and hence of the same structure, but different matrix types or different numerical requirements.

Reexamining the provided run time implementations allows to assess the requirements placed on the matrix and numeric data types. The matrix data type's requirement is determined by its use in Listing 3.12. It is required that an `operator()` is available with which to access the data within the matrix, when supplied with two integer type arguments. No further restrictions, such as memory layout, apply in the given implementation for matrices. In fact anything complying with the required interface will be considered a valid matrix with respect to evaluation.

The requirements enforced on the numeric data type, beside being capable of the basic numeric operations of addition and multiplication, are on the one hand connected to a convertibility/assignability of the result of accessing the matrix to the numeric type, as seen in Listing 3.12. Additionally the numeric type needs to be constructable as done in Listing 3.13 and Listing 3.14.

To complete the given example, the application of the compile time determinant structure and the given run time evaluations is presented using a simple matrix constructed from `std::vector`. Since this representation does not provide the required access mechanism, a thin wrapper is required. It is presented in Listing 3.15 and shows how access is remapped to the inherent mechanisms.

Listing 3.15: Application of the compile time determinant.

```
template<typename MatrixT>
struct vector_vector_like_matrix
{
 const MatrixT& matrix;
 vector_vector_like_matrix(const MatrixT& mx): matrix(mx){};


 double operator()(long x, long y) const
 {
    return matrix[x][y];
 }
};
```

Equipped with the wrapper, the final deployment is illustrated in Listing 3.16. First, the compile time program is evoked, resulting in a data type encoding the instructions to be processed. It requires no information about the run time values of the matrix but takes only the size of the matrix as input. Next, the matrix type is defined and instantiated and this instance bound to the wrapper shown in Listing 3.15. Omitting the insertion of values into the matrix the snippet of code proceeds to provide a variable to contain the result and finally applies the compile time algorithm to the matrix via the `matrix_access` wrapper, thus completing the demonstration of how to calculate the determinant using compile time meta-programming methods.

Listing 3.16: Application of the compile time determinant.

```
typedef determinant_struct_sequence<3>::type det_struct;
typedef std::vector<std::vector<double> > matrix_type;
matrix_type matrix(3, std::vector<double>(3,0));
vector_vector_like_matrix<matrix_type> matrix_access(matrix);

double result(0);
determinant_interface<det_struct>()(matrix_access, result);
```

It has to be stressed at this point that the purpose is to illustrate the procedures and idioms encountered in meta-programming, not to arrive at the optimal solution in terms of run-time, as there are more advanced methods of obtaining the value of a determinant than the presented algorithm [59][60][61]. The recursive nature, which might be detrimental in a run time implementation eases implementation and is eliminated for run time evaluation, since the recursive construction of the prescription has already been completed at compile time.

Furthermore the presented implementation is versatile with respect to acceptable data types. The structure itself is indeed completely agnostic to the employed numeric data type. In contrast to more efficient algorithms this brute force implementation also does not require division operations, thus making it viable to a wider range of data types from a theoretical (see Definition 13) as well as practical point of view.

Finally, the generated compile time structure is not limited to be used for the evaluation of determinants. The structure encountered in the calculation of the determinant may be applied to other fields, such as geometric products in the field of geometric algebra [62].

So far a short outline of the basic settings of digital computers has been sketched into which all further considerations must be mappable. Several different strategies, so called programming paradigms, for describing such a mapping have been presented, which have also been related to different programming languages. After noting several key differences between pure mathematical structures and their realizations within machines, the mathematically simple example of calculating determinants illustrating the procedure of using a compile time program to fully determine the memory accesses and computational steps while leaving the freedom for the values, on which is to be acted on to be supplied at run time. The implemented algorithm was chosen for its relative simplicity which allows to demonstrate what may hide behind relatively simple use of generic implementations. It should therefore be reiterated that for a performance sensitive calculation of determinants different algorithms should be considered. On the other hand it demonstrates how the calculation of permutations can be extracted from the task and made available as an algorithm for reuse.

Part II

Theory

Chapter 4

Mathematical Tools

quanto erit maius aedificium,
tanto altius fodit fundamentum

Aurelius Augustinus

While the problems in scientific computing always have a footing in a specific problem domain, such as one of the disciplines of physics, the algorithms used are often the same in very different contexts. This, however, raises the question, what the uniting elements of scientific computing are, which prevents it from fracturing into separate parts, which are then considered more as a part of the application domain instead of scientific computing. As already hinted at, algorithms may be applied to problems with greatly differing semantics. As such algorithms and solution techniques may be regarded as a first step of identifying common items, which can be identified to belong to the field of scientific computing and which are deployed as tools in the variously different fields of application. Thus, the algorithms require a sufficient level of abstraction to describe them in an application agnostic fashion.

While many implementations of algorithms directly utilize a very low level of abstraction, it is desirable to provide higher levels of abstraction, in order to make broader use more easy to recognize. Following a generic programming approach does not preclude to also utilize specializations which are often already well established, while providing a more general and top level view.

The task of formulating algorithms on a high level of abstraction is eased in case the problem descriptions are of similarly high level. Thus the introduction of mathematical formalisms has a particular focus on terms encountered in the field of dynamics.

Mathematics and its strict and formal definitions are employed to formalize the modelling of algorithms. Hence, it is unavoidable to be familiar with a basic set of mathematical terms and definitions. Whithout an understanding of the setting (the terrain) a problem is much more difficult to solve [63].

Therefore, it is necessary to provide solid foundations concerning the terms used and the concepts connected to them, before proceeding to the analysis or synthesis of systems of non-trivial complexity. To address at least the barest of minima of requirements, a short aggregation of terms is provided in the following to provide an attempt at consistent nomenclature for the remaining chapters, thus making the terrain easier to navigate. The collection of terms draws upon

several distinct sources [64][65][66][67][68][69][70][71][72], which should be consulted in the case that this limited and short compilation of terms and definitions, which is in this context unprecedented in both thoroughness as well as extent, thus acting not only as a backbone but also represents a contribution in its own right, is found to be insufficient. It should be noted, however that the definitions in the given literature are not necessarily compatible to amongst each other when attempting to simply combine several sources, so that great care needs to be taken, not to miss small, but crucial differences in the definitions. Therefore, considerable effort has been invested to make the definitions as self-contained and consistent as possible.

Another venue, not requiring the procurement of books of any kind, is also available by simply turning to Wikipedia [73], which also contains a surprising treasure of mathematical knowledge. A greater caveat applies to that source, however, since not only are the entries often applying terms inconsistently, where every book at least aims to be consistent within itself. Not only has no such uniting aspect in this regard yet surfaced in the context of Wikipedia, but some of the entries may even be outright incorrect.

The different wording and points of view provided by different sources may provide easier insights to different backgrounds and motivations. However, the uniting goal should always be to develop an understanding which is consistent and hopefully has more depth and covers at least a bit more than the problem at hand demands in order to allow for the exploration and investigation, as should be the goal for any and all scientific endeavours.

4.1 Bare Basics

Silently assuming basic familiarity with the entity known as a set [74], several distinctions connected to sets should not go undefined, especially as later concepts build on these very basic definitions and the lack of a clear distinction can be nothing but detrimental. Among others, Definition 36 is a higher level representative of the concept provided in the following definition.

Definition 1 (Open cover) *An open cover $\mathcal{C}$ of a set $\mathcal{X}$ is a family of open sets $\mathcal{U}_i$, whose union results in $\mathcal{X}$:*

$$\mathcal{X} = \bigcup_{i \in A} \mathcal{U}_i \tag{4.1}$$

with A being an arbitrary index set.

A similar, yet distinct concept is given in the next definition.

Definition 2 (Partition of a set) *A partition of a set $\mathcal{A}$ is a set $\mathcal{P}$ of non-empty subsets $p_i \subset \mathcal{A}$, which fulfil the following conditions*

$$\bigcup \mathcal{P} = \mathcal{A} \tag{4.2a}$$

$$p_i \cap p_j = \emptyset, \forall p_i, p_j \in \mathcal{P}, p_i \neq p_j \tag{4.2b}$$

Thus a partition implies that every element of the set $\mathcal{A}$ is assigned to exactly one element of the partition $\mathcal{P}$.

To extend the simple notion of sets and their partitions, the following definition is of high utility.

Definition 3 (Cartesian Product) *The Cartesian product $\mathcal{X} \times \mathcal{Y}$ of two sets $\mathcal{X}$ and $\mathcal{Y}$ is defined as the set of all ordered tuples of the form*

$$\mathcal{X} \times \mathcal{Y} = \{(x, y) | x \in \mathcal{X} \wedge y \in \mathcal{Y}\} \tag{4.3}$$

It is through this simple mechanism represented by the Cartesian product that extensions to higher dimensions are available not only in a theoretical, but especially in an applied setting.

Sets alone present elements and very basic predicates such as testing, if a given element is contained within a set. It is, however, important to provide a more involved mechanism which is readily found in relations [1].

Definition 4 (Relation) *Considering two sets $\mathcal{A}$ and $\mathcal{B}$, a relation R is defined as a subset of the Cartesian product $\mathcal{A} \times \mathcal{B}$.*

$$R \subset \mathcal{A} \times \mathcal{B} \tag{4.4}$$

A relation is not only an important step for further theoretical refinement, but has also been a driving abstraction for digital data storage systems such as databases.

Relations can be further qualified with attributes in case they conform to certain requirements. A selection of qualifiers is provided in order to keep subsequent definitions from becoming unnecessarily convoluted.

Definition 5 (Reflexive relation) *A binary relation R on a set $\mathcal{X}$ is called reflexive, if every element is related to itself*

$$\forall x \in \mathcal{X} : (x, x) \in R \tag{4.5}$$

Definition 6 (Symmetric relation) *A binary relation R on a set $\mathcal{X}$ is called symmetric, if the following is always true:*

$$\forall x, y \in \mathcal{X} : (x, y) \in R \Rightarrow (y, x) \in R \tag{4.6}$$

Definition 7 (Transitive relation) *A binary relation R on a set $\mathcal{X}$ is called transitive, if it follows that:*

$$x, y, z \in \mathcal{X} : (x, y) \in R \wedge (y, z) \in R \Rightarrow (y, z) \in R \tag{4.7}$$

Definition 8 (Equivalence relation) *A binary relation which is reflexive, transitive and symmetric is called an equivalence relation.*

The application of an equivalence relation to the elements of a set leads to the concept of equivalence classes, as defined in the following.

Definition 9 (Equivalence class) *Given an equivalence relation $\sim$, equivalence classes $[x]$ are defined as the sets*

$$[x] = \{y \in \mathcal{X} | y \sim x\} \tag{4.8}$$

[1] This should not turn into a debate which carries more weight, the predicate indicating membership or the set itself.

Equivalence classes are of special interest as they provide a means to partition a set $\mathcal{X}$. The demands placed on equivalence relations (Definition 8) ensure that a set of equivalence classes (Definition 9) is indeed a partition of the set $\mathcal{X}$ (Definition 2):

- Reflexivity (Definition 5) ensures that $[x] \neq \emptyset$
- Symmetry (Definition 6) ensures $[x] = [y] \iff x \sim y$
- Transitivity (Definition 7) asserts that $a \in [x] \wedge a \in [y] \iff [x] = [y]$.

In other words, two elements are either in the same equivalence class, or in disjoint classes. An element $x \in [x]$ is called a representative of the equivalence class $[x]$.

The memory within digital computers is conceptually a set, because every cell of memory can be distinctly addressed even without having to resort to querying the contents of the cells, thus it is really the structure of the cells that forms a set. The presented relations are only a small fraction of the relations available and indeed required to represent modern memory management, but are at least sufficient to cover theoretical considerations.

4.2 Algebraic Structures

Beside the bare structures described so far, the structures resulting from operations defined on top of the sets as a combined union gives rise to algebraic structures. They are of importance as they provide rules, how elements interacts and create new elements.

In the setting of digital computers types are models or extensions of algebraic structures. Therefore, several algebraic structures are introduced in the following with an increasing number of requirements imposed on elements and/or operations.

Definition 10 (Monoid) *A set of elements M on which a binary operation $(\cdot)$ is defined is called a monoid $(M, \cdot)$, if the binary operation satisfies the following conditions:*

- *Closed: The result of the binary operation $(\cdot)$ is again an element of the initial set M:*

$$\forall a, b \in M \rightarrow a \cdot b \in M \tag{4.9}$$

- *Associativity: The order in which consecutive operations are applied does not change the result:*

$$(a \cdot b) \cdot c = a \cdot (b \cdot c), \qquad \forall a, b, c \in M \tag{4.10}$$

- *An identity (neutral) element $e \in M$ exists. such that:*

$$a \cdot e = e \cdot a = a, \forall a \in M \tag{4.11}$$

Further requirements on Definition 10 result in

Definition 11 (Group) *A monoid $(G, \cdot)$, where the binary operation additionally fulfils the condition that for every element in G an inverse element exists, which produces the identity element under the binary operation is called a group*

$$a \cdot b = e, \qquad \forall a, b \in G \tag{4.12}$$

and as a further qualification

Definition 12 (Abelian group) *When the order of the operands of the binary operation $(\cdot)$ of a group $(G, \cdot)$ does not change its result, the group is called Abelian or commutative.*

$$a \cdot b = b \cdot a, \quad \forall a, b \in G \tag{4.13}$$

Groups are basic building blocks in the exploration of further structures, which can be defined by demanding additional operations on the basic sets.

Definition 13 (Ring) *An Abelian group (Definition 12), which is equipped with an additional binary operation under which it is a monoid (Definition 10), is called a ring $(R, +, \cdot)$, if the two binary relations are distributive:*

$$a(b + c) = (ab) + (ac) \tag{4.14}$$
$$(a + b)c = (ac) + (bc) \qquad \forall a, b, c \in R \tag{4.15}$$

The structure of the described ring is, however, insufficient to describe the basic notion of real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. To this end the following definition is required:

Definition 14 (Field) *If the multiplication operation of a ring K is invertible $\forall x \in K \setminus \{0\}$, it is called a field.*

The next definition describes the algebraic structure of entities (Definition 61), which have proven to be immensely useful.

Definition 15 (Module) *A Module $\mathcal{M}$ over a ring R (Definition 13) is an Abelian group (Definition 12) with respect to the operation of the addition of two elements $u, v \in \mathcal{V}$, while additionally being a ring with respect to the operation of multiplying elements $v \in \mathcal{V}$ by elements $a \in R$, which are called scalars.*

A related definition with somewhat stronger requirements yields a structure, which is essential for the construction of simple geometric settings.

Definition 16 (Vector space) *A vector space $\mathcal{V}$ over a field $\mathbb{F}$ is an Abelian group with respect to the operation of the addition of two elements $u, v \in \mathcal{V}$, while additionally being a ring with respect to the operation of multiplying elements $v \in \mathcal{V}$ by elements $a \in \mathbb{F}$. Elements of a vector space $v \in \mathcal{V}$ are called vectors.*

While the elements of vector spaces, the vectors, constitute a powerful concept, they are insufficient to describe all the entities required in modelling scientific processes. Additional entities are therefore required. It is not limited to entities introduced later and therefore provided here to clearly distinguish the algebraic structure from the elements.

Definition 17 (Algebra) *An algebra A over a field F is a vector space equipped with an additional binary relation (Definition 4)*

$$\cdot : A \times A \to A \tag{4.16}$$

A further qualification of the just defined structure may be possible. The availability of the following term allows a more precise classification as found in Definition 56 and in conjunction with Definition 60.

Definition 18 (Graded algebra) *In case the algebra admits the decomposition into additive groups of the form*

$$A = \bigoplus_n A_n, \tag{4.17}$$

where additionally the multiplication operation results in

$$A_i \times A_j \to A_{i+j}, \tag{4.18}$$

it is called a graded algebra.

Among the most versatile and useful, almost ubiquitous, algebraic entities are:

Definition 19 (Polynomials) *A formal prescription of the form*

$$p(X) = \sum_i c_i X^i, \qquad c_i \in R, i \in \mathbb{N} \cup \{0\} \tag{4.19}$$

with coefficients c_i is called a polynomial over the ring (Definition 13) R in the variable X.

The variable X in the purely algebraic definition is a formal symbol and need not be an element of a field (Definition 14), such as $\mathbb{R}$ or $\mathbb{C}$, as in the case of polynomial codes. The algebraic considerations, however, assert that polynomials defined in this fashion can be added (subtracted) and multiplied, thus forming a ring (Definition 13). The case that the variables X are either from $\mathbb{R}$ or $\mathbb{C}$ is of particular usefulness in many fields of mathematics, with the field of interpolation as well integration among them. Then the expression X^i is simply the ith power of a variable $x \in \mathbb{R}(\mathbb{C})$ and values can be derived by simple multiplication within their respective fields.

4.3 Mappings

While the algebraic structures focus on the interplay of elements within given sets, a means to assign elements from differing sets as a special yet highly important case of Definition 4 is desired.

Definition 20 (Mapping) *Considering a relation f, between the sets $\mathcal{D}$ and $\mathcal{B}$, it is considered a map, mapping, or function, if every element of $\mathcal{D}$ is assigned to an element of $\mathcal{B}$. The converse, however, is not necessarily true that is an element of $\mathcal{B}$, called an image, may result from several elements of $\mathcal{D}$, called preimages. The set $\mathcal{D}$ is called the domain, the set $\mathcal{B}$ the codomain of the function f.*

The terms mapping and function can be used synonymously.

Similar to the qualification of relations, mappings can also be further qualified. Qualifications regarding the domain and codomain are provided, from which can be deduced, if a mapping is invertible.

Definition 21 (Injective) *A mapping*

$$f : \mathcal{A} \rightarrow \mathcal{B} \tag{4.20}$$

is called injective, if there is at most one value $b = f(a) \in \mathcal{B}$ for each element $a \in \mathcal{A}$. This means that every preimage $a \in \mathcal{A}$ has an image $b \in \mathcal{B}$, but there may be elements in $\mathcal{B}$, which are not obtained as images.

Connected to injectivity is the concept of a kernel:

Definition 22 (Kernel) *For a given mapping*

$$f : \mathcal{A} \rightarrow \mathcal{B} \tag{4.21}$$

the set of elements resulting in the identity element $e \in \mathcal{B}$ is called the kernel of the mapping f

$$\ker f = \{x \in \mathcal{A} | f(x) = e \in \mathcal{B}\} \tag{4.22}$$

As such the kernel provides information on the injectivity of a mapping. If a mapping is injective, the kernel is trivial

$$\ker f = \emptyset. \tag{4.23}$$

Where injectivity states that there is at most one solution to every element from the domain of a function, it is also possible to assert that there is at least one image for each preimage, as is done in the next definition.

Definition 23 (Surjective) *A mapping*

$$f : \mathcal{A} \rightarrow \mathcal{B} \tag{4.24}$$

is called surjective or onto, if $b = f(a)$ has at least one solution $b \in \mathcal{B}$ for every $a \in \mathcal{A}$. Several identical images in $\mathcal{A}$ may have different preimages in $\mathcal{B}$.

The combination of Definition 21 and Definition 23 yields:

Definition 24 (Bijective) *A mapping*

$$f : \mathcal{A} \rightarrow \mathcal{B} \tag{4.25}$$

that is both injective and surjective is called bijective. It maps every preimage from $\mathcal{B}$ to a distinct image in $\mathcal{A}$. There are no elements in $\mathcal{A}$ which cannot be obtained as images.

A bijective mapping, also called a bijection, is an invertible function. Since injectivity (Definition 21) demands the triviality of the kernel (Definition 22), it follows that an invertible function necessarily also has a trivial kernel.

Mappings are an essential instrument in mathematical descriptions and modelling, because it is possible to compose individual mappings in order to construct new mappings. Considering the mappings

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & & \downarrow g \\ & & \mathcal{C} \end{array} \tag{4.26}$$

it is possible to combine them in a fashion as to obtain:

$$g \circ f : \mathcal{A} \to \mathcal{C} \tag{4.27}$$

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow^{g \circ f} & \downarrow g \\ & & \mathcal{C} \end{array} \tag{4.28}$$

This may also be presented as

$$\forall x \in \mathcal{A} \to g(f(x)) \in \mathcal{C} \tag{4.29}$$

more constructively illustrating the procedure, how the new mapping is to be constructed. The agreement of the codomain of f and the domain of g are crucial for the feasibility of such a composition.

Mappings and their compositions are essential for the operation of modern computational machines, e.g., when dealing with memory management such as virtual memory, which relies on partitions as well as mappings.

The definition of cases of mappings simplifies the formulation of subsequent definitions. Therefore, several special cases are provided here in a collected form.

Definition 25 (Projection map) *A projection map* proj_j*, which may also given as* π_j*, extracts the* j*th component of elements from a Cartesian product (Definition 3) space* $\mathbf{x} \in X_1 \times \ldots \times X_j \times \ldots \times X_k$ *to* X_j*:*

$$\mathrm{proj}_j(\mathbf{x}) = \pi_j(\mathbf{x}) = x_j \tag{4.30}$$

A mapping, which can be used to extend a concept from a setting to another, as is seen in Section 5.2, is provided in the following.

Definition 26 (Lift) *Given the mappings* $f : \mathcal{A} \to \mathcal{B}$ *and* $g : \mathcal{C} \to \mathcal{B}$*, a lift is defined as a map* h *with the property that* $g \circ h = f$

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{f} & \mathcal{B} \\ & \searrow^{h} & \uparrow g \\ & & \mathcal{C} \end{array} \tag{4.31}$$

A subsequently important class of mappings can be abstractly described here, where realizations are then provided by Definition 70 and Definition 68.

Definition 27 (Derivation) *Given an associative algebra $\mathcal{A}$ (Definition 17) and a module $\mathcal{M}$ (Definition 15), a derivation is defined as a mapping*

$$D : \mathcal{A} \to \mathcal{M} \text{ with the property} \tag{4.32a}$$

$$D(fg) = D(f)g + fD(g) \qquad f, g \in \mathcal{A} \tag{4.32b}$$

4.4 Topology

Building on the available definitions, additional structures can be introduced, which can be described as being the bones of geometry as well as being useful for programming abstractions.

Definition 28 (Topology) *A topology $\mathcal{T}$ of a set $\mathcal{X}$ is defined by the following properties:*

- *Both $\emptyset \in \mathcal{T}$ and $\mathcal{X} \in \mathcal{T}$.*
- *A finite intersection of members of $\mathcal{T}$ is in $\mathcal{T}$.*
- *An arbitrary union of members of $\mathcal{T}$ is in $\mathcal{T}$.*

The concept of a topology alone is incomplete, when not applied to a set, thus providing:

Definition 29 (Topological space) *The pair of the set $\mathcal{X}$ and the topology $\mathcal{T}$ are denoted as the topological space $(\mathcal{X}, \mathcal{T})$.*

Topological spaces also allow to further qualify the concepts of mappings, as provided by Definition 20.

Definition 30 (Continuous mapping) *A mapping between two topological spaces*

$$\phi : (\mathcal{X}, \mathcal{T}_X) \to (\mathcal{Y}, \mathcal{T}_Y) \tag{4.33}$$

is continuous, if the preimages of open sets are open sets.

Equipped with the concept of continuity it is beneficial to specially label a certain class of continuous, invertible mappings.

Definition 31 (Homeomorphism) *A continuous mapping ϕ is called a homeomorphism or topological isomorphism, if it satisfies the following properties:*

- *ϕ is a bijection*
- *the inverse mapping ϕ^{-1} is also continuous*

Beside continuity (Definition 30), relations between two mappings from a topological space to another are of interest. Special requirements in this regard are encountered for example in Definition 39, thus motivating the following definition.

Definition 32 (Homotopy) *Given two continuous mappings $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{A} \to \mathcal{B}$, a homotopy h is a continuous mapping such that:*

$$h : \mathcal{A} \times [0;1] \to \mathcal{B} \text{ such that} \tag{4.34a}$$
$$\mathcal{A} \times \{0\} : h(x,0) = f(x) \tag{4.34b}$$
$$\mathcal{A} \times \{1\} : h(x,1) = g(x) \tag{4.34c}$$

A homotopy thusly describes the smooth deformation of one mapping into another.

Further refinement of the concept of a topological space (Definition 29) by including separability of neighbourhoods of different elements of the topological space allows to move closer to the desired settings of geometry described in Definition 37.

Definition 33 (Hausdorff spaces) *The topological space $(\mathcal{X}, \mathcal{T})$ is called Hausdorff, if*

$$\forall x, y \ (x \neq y) \in \mathcal{X} \tag{4.35}$$

there exist open sets U_1, U_2 such that $x \in U_1, y \in U_2$ and $U_1 \bigcap U_2 = \emptyset$.

From the concept of a Hausdorff space an important definition can be derived, which is invaluable for use in the discrete settings of digital computers:

Definition 34 (CW Complex) *The pair of a Hausdorff space X (Definition 33) together with a decomposition E, whose elements $e \in E$ are called cells, is a CW complex, if the following criteria are met:*

- *The interior of every cell $e \in E$ can be mapped continuously (Definition 30) to an open ball.*
- *Every closed cell is contained in a finite union of open cells.*
- *A subset $A \subset X$ is open if and only if $\forall e \in E, A \cap e$ is open.*

A further intermediate definition is introduced, before arriving at the topological space, which carries sufficient additional structure to accommodate the geometric entities (e.g., Definition 54). It serves to again partition the requirements into manageable components, so as not to end up with overly complex and contorted constructs.

Definition 35 (Topological manifold) *A topological manifold $\mathfrak{M}$ of dimension n is a Hausdorff space which is locally homeomorphic to $\mathbb{R}^n$. Consequently, this results in the existence of bijective mappings κ, called charts, of the form*

$$\kappa : \mathcal{U} \to \mathcal{M} \subseteq \mathbb{R}^n, \tag{4.36}$$

which map open neighbourhoods $\mathcal{U}$ for every element $p \in \mathfrak{M}$ to open subsets $\mathcal{M} \in \mathbb{R}^n$. The n-tuple of numbers resulting from $\kappa(p)$ is called the (local) coordinates of p.

As the subsets $\mathcal{U}$ are open and the charts operate solely on them, any conclusion from a single chart alone must remain a local one. However, open subsets may be combined to overcome this limitation thus giving rise to:

Definition 36 (Atlas) *A collection of charts forming an open cover (Definition 1)*

$$\mathcal{A} = \bigcup_i (\mathcal{U}_i, \kappa_i) \tag{4.37}$$

is called an atlas $\mathcal{A}$.

Since the atlas is now defined on the whole manifold, it is possible to judge global properties. Considering two charts of an atlas $\kappa_i(\mathcal{U}_i) \in \mathcal{A}$ and $\kappa_j(\mathcal{U}_j) \in \mathcal{A}$ with the non-empty intersections of open sets

$$\mathcal{U}_i \cap \mathcal{U}_j \neq \emptyset \tag{4.38}$$

it is possible to define chart transitions (a change of coordinates) between these charts within this intersection:

$$\kappa_i \circ \kappa_j^{-1} : \kappa_j(\mathcal{U}_i \cap \mathcal{U}_j) \to \kappa_i(\mathcal{U}_i \cap \mathcal{U}_j). \tag{4.39}$$

An illustration is given in Figure 4.1.

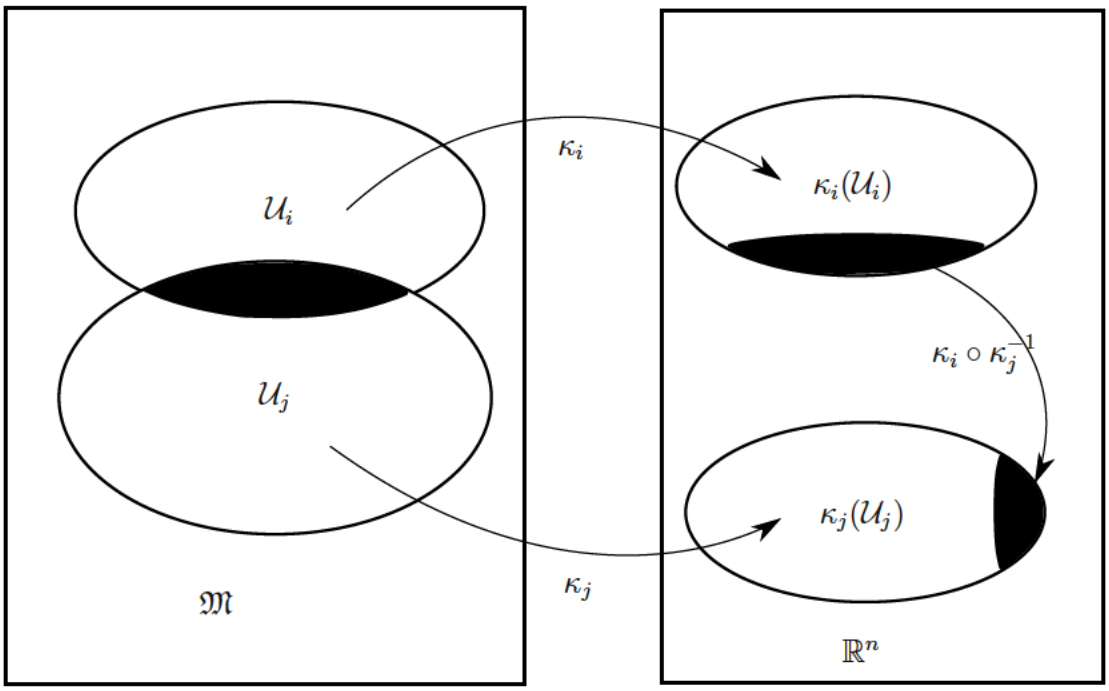


Figure 4.1: Chart transitions on a manifold.

The following definition finally introduces a topological space, which is equipped to admit the differential structures required in the further geometrical considerations (Definition 48).

Definition 37 (k-Differentiable manifold) *A topological manifold $\mathfrak{M}$ with an atlas $\mathcal{A}$ is called a differentiable manifold, if the transitions between charts $\forall \kappa \in \mathcal{A}$ are of differentiability class C^k.*

The requirement of differentiable transitions between mappings relies on the charts themselves being homeomorphisms (Definition 31) as continuity is a prerequisite for differentiability. The additional structure of a differentiable manifold also allows to place a more stringent requirement on a homeomorphism, which is expressed in the following definition.

Definition 38 (Diffeomorphism) *A homeomorphism (Definition 31), which is itself differentiable as well as its inverse, is called a diffeomorphism.*

Differentiable manifolds and diffeomorphisms are of particular importance for physical applications, as already attributed by the ancient adage "natura non facit saltus".

4.5 Fibers

Having defined topological spaces and manifolds, which are the setting of further considerations, it is also required to define how such spaces may be combined and attached to each other. The next definitions provide the fundamentals of these endeavours and also provide the major abstraction for the storage of values in digital computers.

Definition 39 (Fibration) *A continuous mapping (Definition 30)*

$$p : \mathcal{E} \to \mathcal{B} \tag{4.40}$$

is a fibration, if the following assertions hold. For a space $\mathcal{X}$ every homotopy (Definition 32)

$$f : \mathcal{X} \times [0;1] \to \mathcal{B} \tag{4.41}$$

and a mapping

$$\bar{f} : \mathcal{X} \times \{0\} \to \mathcal{E} \tag{4.42}$$

the diagram

$$\begin{array}{ccc} \mathcal{X} \times \{0\} & \xrightarrow{\bar{f}} & \mathcal{E} \\ {\scriptstyle \mathrm{id}_X \times \{0\}} \downarrow & \overset{F}{\nearrow} & \downarrow {\scriptstyle p} \\ \mathcal{X} \times [0;1] & \xrightarrow{f} & \mathcal{B} \end{array} \tag{4.43}$$

commutes, especially noting $p \circ F = f$.

It is also commonly written as fibration sequence

$$\mathcal{F} \to \mathcal{E} \to \mathcal{B} \tag{4.44}$$

The first arrow indicates the inclusion of the so called fiber $\mathcal{F}$ by means of a mapping into the total space $\mathcal{E}$, while the second map is the fibration to the base space $\mathcal{B}$.

An important specialization, with an additional local requirement, can be defined which has important realizations as demonstrated in Definition 52.

Definition 40 (Fiber bundle) *The topological spaces $\mathcal{B}$, called the base space, and $\mathcal{E}$, called the total space, along with a surjective (Definition 23) projection (Definition 25) $\pi : \mathcal{E} \to \mathcal{B}$ is known as a fiber bundle $(\mathcal{E}, \mathcal{B}, \pi, \mathcal{F})$, if it locally satisfies the following condition:*

For every element $x \in \mathcal{B}$ there exists an open neighbourhood U such that the preimages of the projection π, $\pi^{-1}(U)$ are homeomorphic to a product space $U \times F$, such that the following diagram commutes:

$$\begin{array}{ccc} \pi^{-1}(U) & \xrightarrow{\varphi} & U \times F \\ \downarrow{\scriptstyle \pi} & \swarrow{\scriptstyle \mathrm{proj}_1} & \\ U \in \mathcal{B} & & \end{array} \tag{4.45}$$

F is called a fiber over $x \in \mathcal{B}$.

A fiber bundle is also commonly given as

$$\mathcal{F} \to \mathcal{E} \xrightarrow{\pi} \mathcal{B}, \tag{4.46}$$

which also marks fiber bundles as a special case of fibrations. The speciality compared to Definition 39 is the demand for an at least local trivialization of the form $U \times F$. It should not be omitted to point out a certain kinship with Definition 35, which also features a local requirement at its core. It is thus not surprising that manifolds can be represented as a fiber bundle, which, while maybe not theoretically exciting, provides a guide to the storage and representation of even non-trivial manifolds in digital form by using the fiber bundle as the main abstraction [75]. The fiber bundle as an abstraction mechanism should not be underestimated, especially with respect to guiding the implementations on digital computers, as the utilized memory structures represent fiber bundles. The set of memory cells, along with a topology defined on it, acts as a base space, on which the fiber space of values is attatched to. Fiber bundles, however, also provide elegant and powerful solutions in the theoretical setting. Several topics covered in Section 4.6.4 have a simple representation using fiber bundles, when also considering the following definition.

Definition 41 (Section of a fiber bundle) *A mapping*

$$f : U \subseteq \mathcal{B} \to \mathcal{E} \tag{4.47}$$

is called a section of a fiber bundle, if

$$\forall x \in U : \pi \circ f = \pi(f(x)) = x. \tag{4.48}$$

Beside the already presented mechanisms it is also desirable to firmly establish a formal manner in which to transport properties of mappings to various topological spaces, where they have previously not been defined. To provide Definition 44 with in depth backing, first very general notions are introduced.

Definition 42 (Fiber product) *Given two mappings with identical codomain*

$$f : \mathcal{A} \to \mathcal{C} \tag{4.49a}$$
$$g : \mathcal{B} \to \mathcal{C} \tag{4.49b}$$

the fiber product over $\mathcal{B}$ consists of two mappings

$$p_1 : \mathcal{P} \to A \tag{4.50a}$$
$$p_2 : \mathcal{P} \to B \tag{4.50b}$$

such that $p_1 \circ f = p_2 \circ g$, which may also be expressed by saying that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & C \\ \uparrow{\scriptstyle p_1} & & \uparrow{\scriptstyle g} \\ P & \xrightarrow[p_2]{} & B \end{array} \tag{4.51}$$

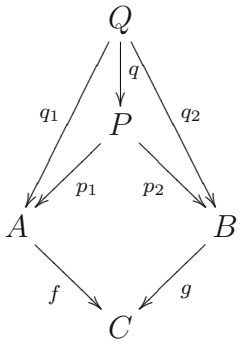
Additionally, it is required that given additional mappings

$$q_1 : Q \to A \tag{4.52}$$
$$q_2 : Q \to B \tag{4.53}$$

which also satisfy the relation $q_1 \circ f = q_2 \circ g$, defines a mapping

$$q : Q \to P \tag{4.54}$$

as is illustrated in the following diagram



(4.55)

This demand ensures that a tuple (P, p_1, p_2) is defined uniquely up to an isomorphism. It is also common to find the notation

$$P = A \times_C B. \tag{4.56}$$

The general notion just defined can be applied to fiber bundles to attach fibers originally situated in one topological space to another one using a simple mapping. The formalization of this is presented next.

Definition 43 (Pullback bundle) *Given a fiber bundle $\pi : \mathcal{E} \to \mathcal{B}$ and a mapping $f : \mathcal{A} \to \mathcal{B}$ it is possible to define a fiberbundle, the so called pullback bundle $f^*\mathcal{E}$, which uses $\mathcal{A}$ as a base space by attaching at every element $x \in \mathcal{A}$ the fiber corresponding to the element $f(x) \in \mathcal{B}$ (the position of attachment is given by the index):*

$$(f^*\mathcal{E})_x = \mathcal{E}_{f(x)} \; \forall x \in \mathcal{A}. \tag{4.57}$$

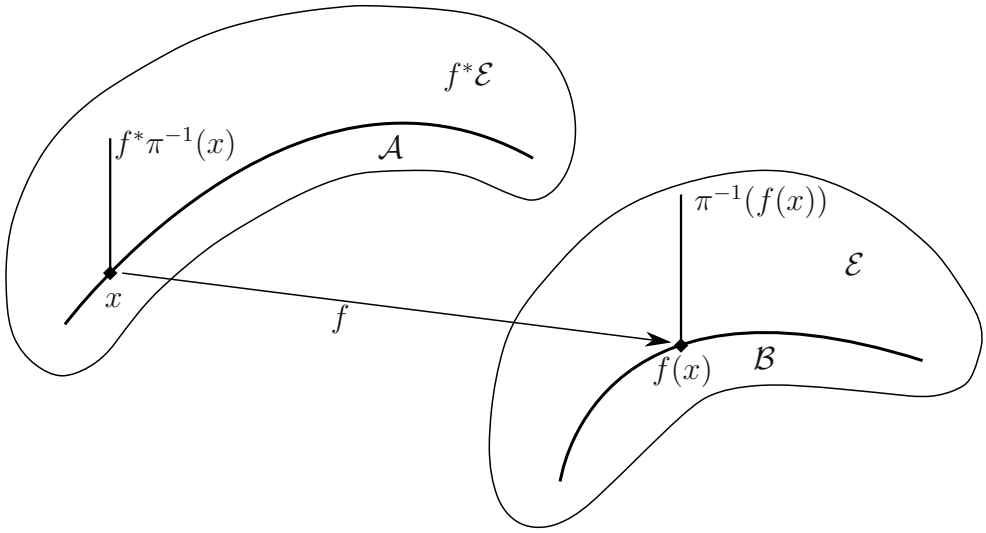


Figure 4.2: Illustration of a pullback bundle.

$$\begin{array}{ccc} f^*\mathcal{E} & \xrightarrow{\pi^* f} & \mathcal{E} \\ {\scriptstyle f^*\pi}\downarrow & & \downarrow{\scriptstyle \pi} \\ \mathcal{A} & \xrightarrow[f]{} & \mathcal{B} \end{array} \tag{4.58}$$

In short, the pullback bundle, as sketched in Figure 4.2, is simply the fiber product (Definition 42) $\mathcal{A} \times_{\mathcal{B}} \mathcal{E}$. It should not go unnoticed that this construct is compatible with sections of fiber bundles (Definition 41); therefore entities which appear as the section of a fiber bundle, such as presented in Definition 61 and Definition 62, will be pulled back and appear again as sections of the pullback bundle (Definition 43).

Considering two topological spaces $\mathfrak{M}$ and $\mathfrak{N}$ and the mappings

$$f : \mathfrak{M} \to \mathfrak{N} \tag{4.59}$$

$$\varphi : \mathfrak{N} \to \mathbb{R} \tag{4.60}$$

it is possible by using the composition of the maps to define a mapping

$$\varphi \circ f = \varphi(f(\cdot)) = f^*\varphi : \mathfrak{M} \to \mathbb{R}, \tag{4.61}$$

which effectively transports the function φ from $\mathfrak{N}$ to $\mathfrak{M}$ against the direction of f. It is said that φ is pulled back from $\mathfrak{N}$ via f.

Definition 44 (Pullback (of functions)) *The mapping f^* resulting from a mapping between two topological spaces*

$$f : \mathfrak{M} \to \mathfrak{N}, \tag{4.62}$$

which transports functions from the codomain of f to its domain is called a pullback (of functions).

The pullback of functions is a particular case of the of the pullback bundle (Definition 43), which illustrates the concept in a relatively simple fashion.

4.6 Differential Geometry

Having carefully prepared the foundations which present the scenery for the geometrical considerations at hand, it is time to turn to geometric entities. It is again prudent to present a very simple case first, which allows to build higher structures upon.

Definition 45 (Curve) *A smooth map of an interval $I \subset \mathbb{R}$ to a differentiable manifold $\mathfrak{M}$ (Definition 37) is called a curve:*

$$\gamma : I \subset \mathbb{R} \to \mathfrak{M}. \tag{4.63}$$

Two curves are not identical, in the strictest of senses, if their images coincide, while their parametrization differ.

Using a curve, every number $s \in I$ is mapped to a point of $\mathfrak{M}$, which may be used as input for the charts of the manifold (Definition 35), thereby effectively creating the new mapping:

$$\kappa \circ \gamma : I \to \mathbb{R}^n \tag{4.64a}$$
$$x^i(s) : I \to \mathbb{R}, \forall i \in [0, \ldots, n-1] \tag{4.64b}$$

This pullback (Definition 44) by the charts also results in coordinate representations for the involved points. While the properties of a considered curve do not depend on the choice of a particular coordinate system, the mere existence of the mapping allows to explain the concept of a differentiable curve.

Definition 46 (Differentiable curve) *A curve $\gamma(s)$ is considered to be differentiable, if the functions describing its coordinate representations $x^i(s)$ as defined by Equation 4.64 are differentiable functions.*

Having defined differentiable curves, it is only fitting to recall basic mathematics and how tangents to graphs of simple functions may be linked to differentials, and extend the concept by applying similar reasoning in this setting.

Definition 47 (Tangent curves) *Two curves $\gamma_1(t), \gamma_2(t)$ are tangent to each other in a point P, if*

$$P = \gamma_1(t_0) = \gamma_2(t_0) \tag{4.65}$$
$$\frac{\mathrm{d}}{\mathrm{d}t}\gamma_1(t)\big|_{t_0} = \frac{\mathrm{d}}{\mathrm{d}t}\gamma_2(t)\big|_{t_0} \tag{4.66}$$

The differentiation is with respect to a single parameter and touches the coordinate expressions according to Equations 4.64.

Figure 4.3 illustrates that Definition 47 can be met by several, in fact infinitely many, curves. However, recalling Definition 9 it is possible to extract essential parts of information which leads to the following definition introducing a key geometric entity.

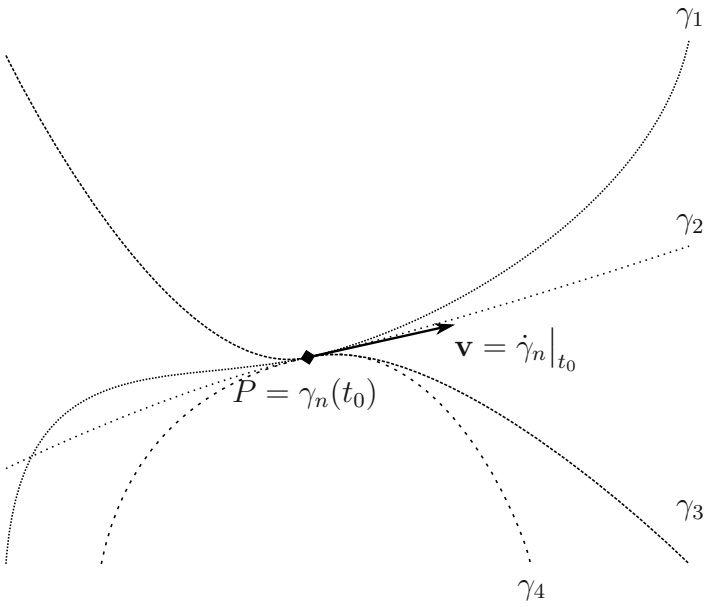


Figure 4.3: Several curves tangent in a point P.

Definition 48 (Tangent vector space) *Considering the set of all curves passing through a point $P \in \mathfrak{M}$, the identity of the first derivatives defines a set of equivalence classes (Definition 9):*

$$\dot{\gamma} = [\gamma] \tag{4.67}$$

The equivalence classes at a given point P define a space TM_P, which carries the structure of a vector space (Definition 16) and tangent to $\mathfrak{M}$ at the point P. The dimensions of the manifold $\mathfrak{M}$ and TM_P are identical

$$\dim(\mathfrak{M}) = \dim(TM_P). \tag{4.68}$$

The vectors now available after Definition 48 alone do not suffice to easily develop complex models of physical processes. A method of evaluation, a measurement, of the vectors is also required, which motivates the next definition.

Definition 49 (Dual vector space) *Given a vector space $\mathcal{V}$, its dual space $\mathcal{V}^*$ is comprised by the linear functionals*

$$\alpha \in \mathcal{V}^* : \mathcal{V} \to \mathbb{R}. \tag{4.69}$$

The dual space $\mathcal{V}^*$ also carries the structure of a vector space (Definition 16), its elements are called co-vectors or one-forms. In particular it is possible to define a non-degenerate bilinear-form, called the scalar product.

$$\langle \alpha, a \rangle : \mathcal{V}^* \times \mathcal{V} \to \mathbb{R} \tag{4.70}$$

Where non-degenerate means that $\langle \alpha, a \rangle = 0$ may only occur, if either

$$\alpha = 0 \qquad \forall a \in \mathcal{V} \quad \text{or} \tag{4.71}$$

$$a = 0 \qquad \forall \alpha \in \mathcal{V}^*. \tag{4.72}$$

In finite dimensions, the dimensions of two dual vector spaces are equal.

$$\dim(\mathcal{V}) = \dim(\mathcal{V}^*) \tag{4.73}$$

Definition 50 (Cotangent vector space) *The dual space of the tangent vector space is called the cotangent vector space T^*M_P .*

In the case of infinite-dimensional vector spaces such as the spaces of functions, a slight modification is required. When dealing with infinite dimensions, the topological dual space is defined by demanding the linear functionals to be continuous. In finite dimension this demand is not required explicitly, since all linear functionals are inherently continuous.

Before proceeding further it is necessary to clarify another piece of terminology connecting Definition 40 and Definition 16.

Definition 51 (Vector bundle) *A fiber bundle (Definition 40), where the fibers F carry the structure of a vector space (Definition 16), is called a vector bundle.*

Definition 52 ((Co)Tangent bundle) *The (co)tangent spaces TM_P(T^*M_P) (Definition 48) are parametrized on the points $P \in \mathfrak{M}$. The disjoint union of all (co)tangent spaces*

$$TM = \bigcup_{P\in\mathfrak{M}} TM_P \tag{4.74}$$

$$\left(T^*M = \bigcup_{P\in\mathfrak{M}} T^*M_P\right) \tag{4.75}$$

together with a projection

$$\pi_M : TM \to \mathfrak{M} \tag{4.76}$$

$$(\pi^*_M : T^*M \to \mathfrak{M}) \tag{4.77}$$

is a vector bundle (Definition 51), called the tangent bundle of $\mathfrak{M}$. The manifold is the base space, while the tangent spaces at each of the points P are the attached fibers.

The structure of a (co)tangent bundle TM (T^*M) of a smooth manifold $\mathfrak{M}$ (Definition 37) is again a smooth manifold.

A particular mapping between the tangent and co-tangent bundle will be interesting in conjunction with Section 5.3 and is thus introduced here.

Definition 53 (Legendre map) *The Legendre map is a mapping between the tangent and the cotangent bundle (Definition 52) of a manifold $\mathfrak{M}$ (Definition 37)*

$$\hat{L} : TM \to T^*M \tag{4.78}$$

such that the projection map π_1 (Definition 25)

$$\pi_1 \circ \hat{L} = \pi_1. \tag{4.79}$$

Particularly, with $v, w \in \pi^{-1}(x)$ vectors from a fiber over x it is related to a function

$$L : TM \to T^*M \tag{4.80}$$

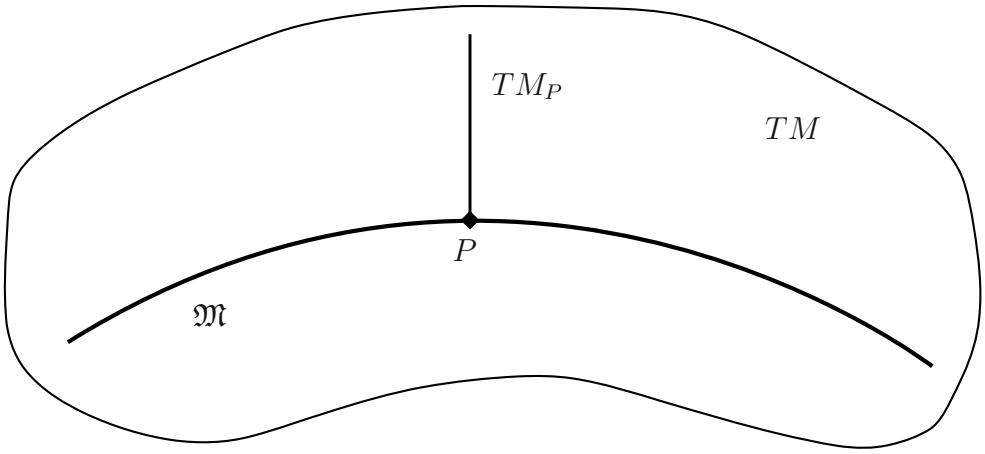


Figure 4.4: Illustration of a tangent fiber and the tangent bundle of a manifold.

by

$$\langle \hat{L}(v), w \rangle = \frac{\mathrm{d}}{\mathrm{d}t} \Big|_0 L(v + tw). \tag{4.81}$$

Similar to the introduction of the elements of the dual vector space, the 1-forms, it is possible to define more complex mappings not only from the vector space but also its dual to the field above which the vector space has been constructed.

Definition 54 (Tensor) *A multilinear mapping (linear every argument)*

$$t^r_s : \underbrace{\mathcal{V} \times \ldots \times \mathcal{V}}_{s} \times \underbrace{\mathcal{V}^* \times \ldots \times \mathcal{V}^*}_{r} \to \mathbb{R} \tag{4.82}$$

is called an (r, s)-tensor.

In this context the original vectors and 1-forms appear as the special cases of $(1, 0)$ and $(0, 1)$ tensors respectively. The collection of all tensors of type (r, s) based on the vector space $\mathcal{V}$ again has the structure of a vector space and shall be denoted by $\mathcal{T}^r_s(\mathcal{V})$. The dimension of this vector space is

$$\dim \mathcal{T}^r_s(\mathcal{V}) = r + s. \tag{4.83}$$

While the bases of the tangent and co-tangent vector spaces can be linked to the charts of the manifold $\mathfrak{M}$, this is so far not available for the vector space of tensors. It is therefore desirable to have a mapping available which, among other things, enables the determination of bases for the tensors spaces of arbitrary dimension from the initial tangent and co-tangent spaces.

Definition 55 (Tensor product) *The non-commutative and associative mapping*

$$\otimes : \mathcal{T}^r_s(\mathcal{V}) \times \mathcal{T}^t_u(\mathcal{V}) \to \mathcal{T}^{r+t}_{s+u}(\mathcal{V}) \tag{4.84}$$

where the components follow

$$\xi \in \mathcal{T}^r_s(\mathcal{V}), \zeta \in \mathcal{T}^t_u(\mathcal{V}) : (\xi \otimes \zeta)(v_1, \ldots, v_s, \alpha_1, \ldots, \alpha_r, w_1, \ldots, w_u, \beta_1, \ldots, \beta_t) = \tag{4.85a}$$

$$\xi(v_1, \ldots, v_s, \alpha_1, \ldots, \alpha_r)\zeta(w_1, \ldots, w_u, \beta_1, \ldots, \beta_t) \tag{4.85b}$$

is called a tensor product.

4.6.1 Tensor Coordinates

Tensors with their algebraic structures represent a very abstract and powerful concept with considerable applicability.The notion of a tensor and its properties do not rely on the particular choice of its representation. The properties do not specify, however, how a particular tensor is defined and how to obtain its corresponding value for a given input. Coordinates with respect to a given choice of base b provide a means to address this problem.

The linear structure of tensors and using the tensor product (Definition 55) allows to express any tensor ξ in the form of

$$\xi \in \mathcal{T}_s^r(\mathcal{V}) \tag{4.86}$$

$$\xi = \sum \Xi_{1\dots s}^{1\dots r} \, b_{1\dots r}^{1\dots s}, \tag{4.87}$$

where $\Xi_{1\dots s}^{1\dots r}$ are the coordinates of the tensor ξ in the base elements $b_{1\dots r}^{1\dots s}$. Providing the full set of tensor coordinates along with the base they refer to, completely specifies the tensor. This important fact is quite commonly abused attributing special properties to the coordinates themselves and in fact considering tensors and vectors alike as little more then a suitable collection of values as found, for example, in matrix representations. While indispensable, a reduction to this primitive level of abstraction belies the true structure of the entities at hand. A tendency of this reductionism has even been compared to a mathematical disease [76]. Where this regression of abstraction may be tedious from a theoretical setting, its severity only increases in the case that properties should be qualified from the coordinates, when they are subjected to calculations of limited precision. It is therefore highly desirable to retain as much abstract information as possible in both theoretical as well as practical fields.

4.6.2 Bra-Ket Notation

As it is prominently used in quantum physics (see Section 5.7) a few words shall be invested to address the topic of Dirac's bra-ket notation [77], which allows rapid coordinate manipulation of entities. Dirac identified a vector $\mathbf{v}$ of a vector space $\mathcal{V}$ (Definition 16) with a so called ket, represented as:

$$|v\rangle \in \mathcal{V} \tag{4.88}$$

At the same time a 1-form $\alpha \in \mathcal{V}^*$ (Definition 49) is identified by a so called bra, written in the following form:

$$\langle\alpha| \in \mathcal{V}^* \tag{4.89}$$

The simplicity stems from the fact that the scalar product may simply be written as:

$$\langle\alpha|v\rangle, \ \alpha \in \mathcal{V}^*, v \in \mathcal{V} \tag{4.90}$$

While this expression is highly similar to (4.70), it can further be enhanced as follows:

$$\langle\alpha|A|v\rangle, \ \alpha \in \mathcal{V}^*, v \in \mathcal{V} \tag{4.91}$$

Where A is a map of the form

$$A : \mathcal{V} \to \mathcal{V} \tag{4.92}$$

and in this context is often called an operator.

Furthermore it is possible to express the tensor product (Definition 55) by simply reversing the order of notation:

$$|v\rangle\langle\alpha| = v \otimes \alpha,\ \alpha \in \mathcal{V}^*, v \in \mathcal{V} \tag{4.93}$$

The simplicity to describe these, in quantum physics regularly required, tasks in a concise, elegant and coordinate independent manner is the true expressive power made available by Dirac's notation.

4.6.3 Beyond 1-Forms

The tensor product (Definition 55) allows the definition of an associative, non-commutative, graded algebra (Definition 18).

Definition 56 (Tensor algebra) *The direct sum of all spaces $\mathcal{T}^r_s(\mathcal{V})$*

$$\mathcal{T}(\mathcal{V}) = \bigoplus_{r,s=0}^{\infty} \mathcal{T}^r_s(\mathcal{V}) \tag{4.94}$$

together with the tensor product forms the tensor algebra.

Since there is no restriction on any of the indices r or s, the dimension of this tensor algebra is infinite.

While the tensor algebra is highly versatile and adaptable, its structure is not quite suited to conveniently represent physical quantities. A different structure is therefore introduced in the following. To this end, it is profitable to examine the symmetry properties of tensors. Since symmetry is determined by the exchange of the arguments of a tensor, only tensors which are purely drawn from either $\mathcal{T}^r_0(\mathcal{V})$ or $\mathcal{T}^0_s(\mathcal{V})$ are eligible for symmetry considerations.

Definition 57 (Symmetric Tensor) *A tensor is called symmetric in case that an exchange of two of its arguments does not change its value. Considering both kinds of eligible tensor types gives*

$$\xi \in \mathcal{T}^0_p : \xi(\ldots, v, \ldots, w, \ldots) = \xi(\ldots, w, \ldots, v, \ldots), \tag{4.95a}$$
$$x \in \mathcal{T}^p_0 : x(\ldots, \alpha, \ldots, \beta, \ldots) = x(\ldots, \beta, \ldots, \alpha, \ldots). \tag{4.95b}$$

The definition of anti-symmetric tensors follows analogously.

Definition 58 (Anti-Symmetric Tensor) *A tensor is called anti-symmetric or skew symmetric in case that an exchange of two of its arguments reverses the sign of its value. Again considering both kinds of eligible tensor types gives*

$$\xi \in \mathcal{T}^0_p : \xi(\ldots, v, \ldots, w, \ldots) = -\xi(\ldots, w, \ldots, v, \ldots), \tag{4.96a}$$
$$x \in \mathcal{T}^p_0 : x(\ldots, \alpha, \ldots, \beta, \ldots) = -x(\ldots, \beta, \ldots, \alpha, \ldots). \tag{4.96b}$$

The cases which lack the degrees of freedom to accommodate the previous notion of symmetric and anti-symmetric are defined as being symmetric (Definition 57) as well as, at the same instant, anti-symmetric (Definition 58). In particular this concerns tensors of types $\mathcal{T}_1^0(\mathcal{V})$, $\mathcal{T}_0^1(\mathcal{V})$ and $\mathcal{T}_0^0(\mathcal{V})$.

The special case of anti-symmetric tensors of type $\mathcal{T}_p^0(\mathcal{V})$ is further distinguished by identifying them.

Definition 59 (p-form) *A totally anti-symmetric tensor $\xi \in \mathcal{T}_p^0$ is called a p-form. The collection of all p-forms is denoted by $\bigwedge^p \mathcal{V}^*$.*

$$\bigwedge^p \mathcal{V}^* \subseteq \mathcal{T}_p^0(\mathcal{V}) \tag{4.97}$$

The anti-symmetry limits the dimensionality of the non-trivial subspaces and links them to the dimension n of the underlying vector space $\mathcal{V}$ (Definition 16), since the anti-symmetry forces tensors to vanish in case

$$\xi = 0 \in \bigwedge^m \mathcal{V}^*, \quad \forall m > \dim \mathcal{V}^* \tag{4.98}$$

thus resulting in

$$\dim \bigwedge^p \mathcal{V}^* = \frac{n!}{(n-p)!p!} = \binom{n}{p}. \tag{4.99}$$

Furthermore, the union of these subspaces

$$\bigwedge \mathcal{V}^* = \bigcup_p \bigwedge^p \mathcal{V}^* \tag{4.100}$$

is no longer closed under the tensor product (Definition 55). It is however possible to define a binary relation which is closed in the union of the fully anti-symmetric tensors.

Definition 60 (Exterior product) *The exterior product*

$$\wedge : \bigwedge^p \mathcal{V}^* \times \bigwedge^q \mathcal{V}^* \to \bigwedge^{p+q} \mathcal{V}^* \tag{4.101}$$

is a non-commutative, due to

$$\alpha \wedge \beta = (-1)^{pq} \beta \wedge \alpha, \quad \forall \alpha \bigwedge^p \mathcal{V}^*, \beta \in \bigwedge^q \mathcal{V}^* \tag{4.102}$$

binary relation.

The space of fully anti-symmetric tensors (forms) $\bigwedge \mathcal{V}^*$ together with the exterior product form an associative, non-commutative graded algebra (Definition 18).

The exterior product and the tensor product are related to one another according to

$$\begin{aligned}(\alpha \wedge \beta)(v_1, \ldots, v_{p+q}) =& \frac{1}{n!m!} \sum_\pi \operatorname{sign}(\pi)(\alpha \otimes \beta)(v_{\pi(1)}, \ldots, v_{\pi(n+m)}) \\ & \forall \alpha \bigwedge^p \mathcal{V}^*, \beta \in \bigwedge^q \mathcal{V}^*,\end{aligned} \tag{4.103}$$

where π are permutations of the input vectors and $\operatorname{sign}(\pi)$ is the sign of a permutation.

4.6.4 Fields

After the basic geometric entities have been introduced in a per point nature it is desirable to also provide notions of distributed geometric items.

Definition 61 (Scalar field) *A smooth mapping ω assigning to each point P of a differentiable manifold $\mathfrak{M}$ (Definition 37) a value of $\mathbb{R}$ is called a scalar field,*

$$\omega : \mathfrak{M} \to \mathbb{R}. \tag{4.104}$$

Recalling that charts are bijections (Definition 24), it is possible to pull the scalar field ω back (Definition 44) to open sets of $\mathbb{R}^n$ using

$$\omega \circ \kappa^{-1} : \mathbb{R}^n \to \mathbb{R}. \tag{4.105}$$

This provides an opportunity to define continuity of the scalar field ω by requiring continuity on $\mathbb{R}^n$. Similarly, differentiability of the scalar field ω is defined by demanding the existence of the partial derivatives of the coordinate functions x^i.

The scalar fields on $\mathfrak{M}$ together with the binary operations of addition and multiplication form a ring $\mathbb{F}(\mathfrak{M})$ (Definition 13) or $\mathbb{F}$ for short over the manifold.

Using the scalar fields $\mathbb{F}(\mathfrak{M})$ it is possible to define a derivation (Definition 27)

$$\mathbf{v} : \mathbb{F}(\mathfrak{M}) \to \mathbb{R}, \tag{4.106}$$

which is isomorphic to the definition of vectors presented before (Definition 48) and is therefore suited as an alternate definition. This enables to view a vector as a directional derivative of a scalar field at a point P .

$$\mathbf{v}(\omega)(P), \quad P \in \mathfrak{M}, \omega \in \mathbb{F}(\mathfrak{M}), \mathbf{v} \in TM_P \tag{4.107}$$

As scalar values are now associated to a manifold $\mathfrak{M}$ as well as a means of to derive vectors from the scalar fields, it is also desirable to extend this concept to vectors.

Definition 62 (Vector field) *A vector field is defined as a mapping assigning to each point $P \in \mathfrak{M}$ a vector from the tangent space TM_P (Definition 48) at P*

$$V : \mathfrak{M} \to TM_P. \tag{4.108}$$

Thus any vector field is a section (Definition 41) of the tangent bundle (Definition 52).

Where a vector has been introduced as a mapping of the form $\mathbb{F}(\mathfrak{M}) \to \mathbb{R}$, a vector field is a mapping of the form $\mathbb{F}(\mathfrak{M}) \to \mathbb{F}(\mathfrak{M})$. Thus a vector field appears as a derivation (Definition 27) of scalar field giving rise to the new scalar field ω_n:

$$\omega_n = V(\omega), \quad \omega_n, \omega \in \mathbb{F}(\mathfrak{M}) \tag{4.109}$$

A vector field V, which indeed maps back to the scalar fields $\mathbb{F}(\mathfrak{M})$, is said to be differentiable.

Having established the concepts of both differentiable curves (Definition 46) and vector fields (Definition 62) it is possible to link them.

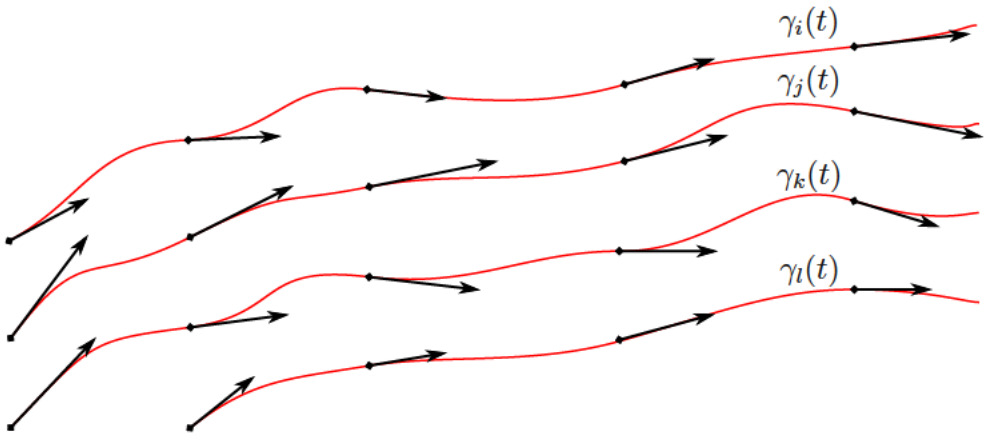


Figure 4.5: Integral curves $\gamma(t)$ of a vector field.

Definition 63 (Integral curves (of vector fields)) *A vector field V (Definition 62) on a manifold $\mathfrak{M}$ (Definition 37) assigns a vector $V_P \in T_P M$ to each point $P \in \mathfrak{M}$. Considering this vector V_P as a representative of the differential of a curve γ (Definition 46) at P defines a set of integral curves or streamlines of the vector field V as illustrated in Figure 4.5.*

$$\dot{\gamma}_P = V_P, \quad \forall P \in \mathfrak{M} \tag{4.110}$$

The existence of integral curves of a vector field allows to further qualify a vector field, since it links the extended entity of a curve to the local entity of a vector.

Definition 64 (Complete vector field) *A vector field V (Definition 62) is said to be complete, if the integral curves (Definition 63), which are initially only defined locally $s \in I \subset \mathbb{R}$, can be extended to all parameters $s \in \mathbb{R}$ for all points $P \in \mathfrak{M}$.*

Now that vector fields and curves have been intrinsically connected to each other, it is prudent to reexplore the issue of differing parametrization. Given two curves γ_1 and γ_2 with identical images in the manifold $\mathfrak{M}$ but differing parametrization

$$\gamma_2(t) = \gamma_1(\sigma(t)) \tag{4.111}$$

results in different values of the tangent vector

$$\dot{\gamma}_2 = \dot{\sigma}\dot{\gamma}_1. \tag{4.112}$$

This, however, indicates that the special cases of $\dot{\sigma} = 1$ such as reparametrizations of the form $t' = t + \tau$ do not change the obtained tangent vectors.

Similar to vector fields, it is also possible to associate a tensor to every point of a given manifold, thus leading to the next definition.

Definition 65 (Tensor field) *A tensor field is defined as a mapping assigning to each point $P \in \mathfrak{M}$ a tensor from the tangent tensor algebras $\mathcal{T}_s^r(\mathcal{V})$ (Definition 56) at P*

$$V : \mathfrak{M} \to \mathcal{T}_s^r(\mathcal{V}). \tag{4.113}$$

Considering a vector field V on a manifold $\mathfrak{M}$ defines a system of integral curves, which completely fill the manifold without intersecting.

Definition 66 (Local flow) *The integral curves of a vector field (Definition 63) give rise to a mapping*

$$\Phi_t : \mathfrak{M} \to \mathfrak{M}, \tag{4.114}$$

which depends on one parameter t and describes a displacement of all points along the local integral curves, shown in Figure 4.5. This so called flow is considered local, when the mapping is defined for a limited range of the parameter t. It may also be written as

$$\Phi : \mathfrak{M} \times \mathbb{R} \to \mathfrak{M}. \tag{4.115}$$

Where the previous definition was only concerned with local properties, an extension to global scale is possible as well.

Definition 67 (Global flow) *A local flow (Definition 66) resulting from a complete vector field V (Definition 64) and therefore an unconstrained parameter $t \in [-\infty; \infty]$ is called a global flow or simply flow.*

A global flow is a fibration (Definition 39) of the manifold $\mathfrak{M}$ on which it is defined along one-dimensional, non-intersecting sub manifolds, the integral curves (Definition 63) of the associated vector field (Definition 62).

Flows and vector fields correspond to each other bijectively (Definition 24): each vector field may be viewed as generating a field by its integral curves, while the corresponding vector field is recovered from a given flow by differentiation. Furthermore, flows can be combined

$$\Phi_{s+t} = \Phi_s \circ \Phi_t, \tag{4.116}$$

which endows flows on $\mathfrak{M}$ with a group structure (Definition 11) where the identity element is

$$\Phi_0 \tag{4.117}$$

a one parameter group.

Flows may also be combined with functions f on the manifold $\mathfrak{M}$.

$$\begin{array}{ccc} P & \xrightarrow{\Phi_t} & Q \\ {\scriptstyle f}\downarrow & \swarrow{\scriptstyle f_t^* = f \circ \Phi_t^{-1}} & \\ f(P) & & \end{array} \tag{4.118}$$

4.6.5 Derivatives

The structure of flows will prove to be of importance in further considerations in Section 5.3. It is therefore prudent to further explore relations of flows. As has already been established flows and vector fields (Definition 62) are linked by differentiation, whereby similar structures instantiating Definition 27 become apparent.

A flow Φ_t (Definition 67) defines a pull back (Definition 44) of the tensor fields on the manifold $\mathfrak{M}$ (Definition 37)

$$\Phi_t^* : \mathcal{T}_s^r(\mathfrak{M}) \to \mathcal{T}_s^r(\mathfrak{M}) \tag{4.119}$$

called a Lie transport. A tensor field $A \in T_s^r$, which is invariant under a flow

$$\Phi_t^* A = A \tag{4.120}$$

is called Lie dragged. This invariance, however, is not the general case, which motivates the next definition.

Definition 68 (Lie derivative) *The measure of non-invariance of a tensor field with respect to the effect to the pull back due to a flow is obtained by the Lie derivative which is defined as*

$$\mathcal{L}_{\mathbf{v}} : \mathcal{T}_s^r(\mathfrak{M}) \to \mathcal{T}_s^r(\mathfrak{M}) \tag{4.121}$$

$$\mathcal{L}_{\mathbf{v}}(A) = \frac{\mathrm{d}}{\mathrm{d}t}\Big|_0(\Phi_t^*(A)), \tag{4.122}$$

where $\mathbf{v}$ *is the vector field (Definition 62) generating the flow* Φ_t *(Definition 67).*

In the case of functions, which are represented as tensors of class $\mathcal{T}_0^0(\mathcal{V})$, the Lie derivative yields

$$\mathcal{L}_{\mathbf{v}}(f) = \mathbf{v}(f), \tag{4.123}$$

the directional derivative.

Since the special case of vector fields (Definition 62) is often encountered, it is awarded special notation, which shall not go omitted here, as it will also be used, especially in Chapter 5, which deals with dynamics.

Definition 69 (Lie bracket) *The Lie derivative (Definition 68), in the case of vectors, which are included in Definition 68 as* $\mathcal{T}_0^1(\mathcal{V})$ *tensors, is used to define the Lie bracket, also known as the Jacobi bracket or commutator, as:*

$$\mathcal{L}_{\mathbf{v}}(\mathbf{w}) = [\mathbf{v}, \mathbf{w}] \tag{4.124}$$

The Lie bracket is skew-symmetric

$$[\mathbf{v}, \mathbf{w}] = -[\mathbf{w}, \mathbf{v}] \tag{4.125}$$

and satisfies the so called Jacobi identity

$$0 = [[\mathbf{v}, \mathbf{w}], \mathbf{u}] + [[\mathbf{u}, \mathbf{v}], \mathbf{w}] + [[\mathbf{w}, \mathbf{u}], \mathbf{v}]. \tag{4.126}$$

In case the Lie bracket vanishes,

$$[\mathbf{v}, \mathbf{w}] = 0, \tag{4.127}$$

the vector fields $\mathbf{v}$ and $\mathbf{w}$ are said to commute. Figure 4.6 gives a visualization of the commutator. Depending on the order in which the integral curves are followed, either the point D or the point

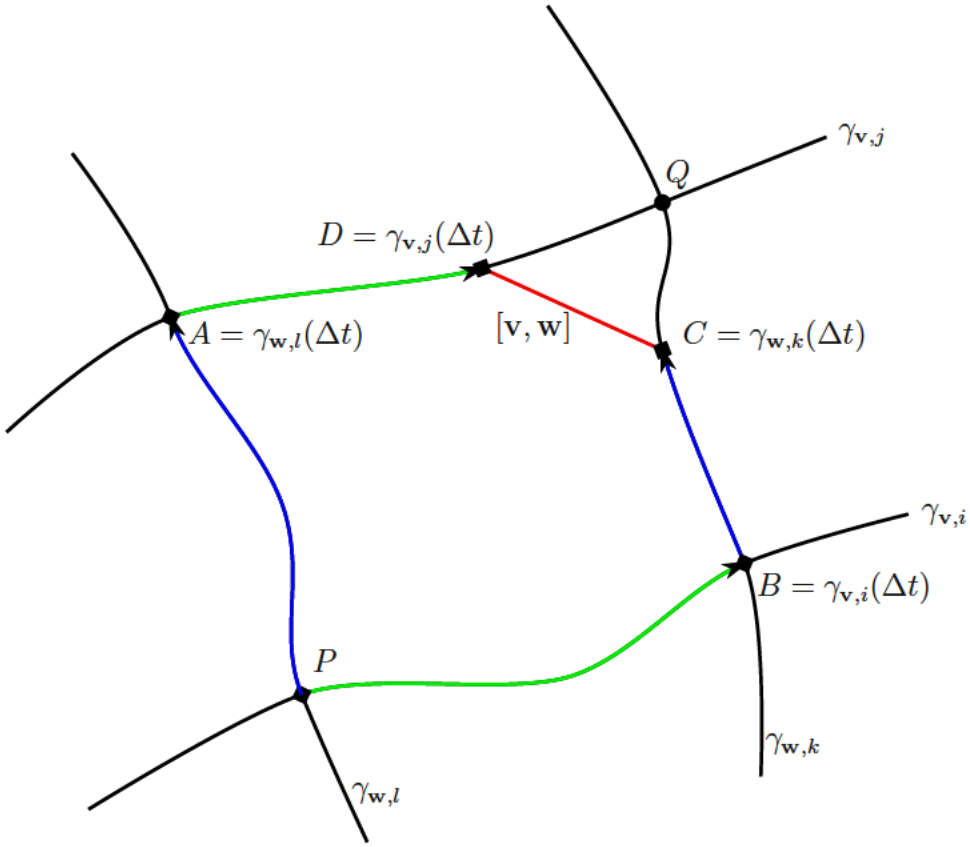


Figure 4.6: Illustration of the commutator $[\mathbf{v}, \mathbf{w}]$ of two vector fields $\mathbf{v}$ and $\mathbf{w}$.

C is reached, only in case the vector fields commute, the figure is closed at the point Q, otherwise, the Lie bracket yields by how much the closure falls short.

Furthermore, there is a relation between the Lie bracket and the Lie derivative.

$$\mathcal{L}_{[\mathbf{v},\mathbf{w}]} = [\mathcal{L}_\mathbf{v}, \mathcal{L}_\mathbf{w}] = \mathcal{L}_\mathbf{v}\mathcal{L}_\mathbf{w} - \mathcal{L}_\mathbf{w}\mathcal{L}_\mathbf{v} \tag{4.128}$$

The connections of flows with vectors and the Lie derivative enables the definition of an exponential function exp using the following characteristics

$$\gamma_\mathbf{v}(1) = \exp(\mathbf{v}), \tag{4.129}$$
$$\dot{\gamma_\mathbf{v}}(0) = \mathbf{v}. \tag{4.130}$$

This leads to the following expression which also meets the group structure established in Equation 4.116,

$$\Phi_t = \exp(t\mathbf{v}), \tag{4.131}$$

called the exponential map. Regarding the pull back (Definition 44) Φ_t^* the following expression can be established:

$$\Phi_t^* = \exp(t\mathcal{L}_v) \tag{4.132}$$

Where the Lie derivative (Definition 68) is specially connected to flows and is a map among tensors (Definition 54) of the same type, mappings among different types of tensors and forms are also needed. They are presented in the following along with their connection to the Lie derivative.

A mapping especially important among p-forms (Definition 59) is provided in the following form of:

Definition 70 (Exterior derivative) *A mapping*

$$\mathrm{d} : \bigwedge^p \to \bigwedge^{p+1} \tag{4.133}$$

of p-forms to $(p+1)$-forms with the following properties

- d *is linear.*
- $\mathrm{d}f$ *is the differential for smooth functions f.*
- $\mathrm{d}(\mathrm{d}f) = 0 - \mathrm{d}$ *is nilpotent.*
- $\mathrm{d}(\alpha \wedge \beta) = \mathrm{d}a \wedge \beta + (-1)^p(\alpha \wedge \mathrm{d}\beta) - \mathrm{d}$ *obeys a graded Leibniz rule.*

is called the exterior derivative. It is a (anti)derivation of degree 1.

While the exterior derivative has special importance to forms (Definition 59), a mapping only involving mixed tensors is also useful.

Definition 71 (Contraction) *Given a tensor $t \in T^r_s$ (Definition 54) for $s, r \geq 1$ the contraction C^k_l is defined as a linear map*

$$C^k_l : T^r_s \to T^{r-1}_{s-1} \tag{4.134}$$

resulting in a tensor whose rank has been reduced by 2.

As a particular special case the contraction can be used to express the scalar product given by Equation 4.70. Given a vector $\mathbf{v}$ and a 1-form α the scalar product of these two entities can be expressed using the tensor product (Definition 55) and the just introduced contraction as

$$C^0_0(\alpha \otimes \mathbf{v}) = \langle \alpha, \mathbf{v} \rangle. \tag{4.135}$$

The tensor product first produces a tensor of type $\mathcal{T}^1_1(\mathcal{V})$, whose contraction is of type $\mathcal{T}^0_0(\mathcal{V})$ which corresponds to scalar values.

Where this contraction is defined to act on a single tensor of mixed type, it is also possible to provide a similar and explicit mapping of forms involving an explicitly provided vector field (Definition 62).

Definition 72 (Interior product) *Given a vector field $\mathbf{x}$ (Definition 62) on a manifold $\mathfrak{M}$ (Definition 37) a linear mapping of the form*

$$i_{\mathbf{x}} : \bigwedge^k \to \bigwedge^{k-1} \tag{4.136}$$

on the differential forms $\alpha \in \bigwedge^k$ is called the interior product.

Written in component vectors it takes the shape:

$$(i_{\mathbf{x}}\alpha)(\mathbf{v}_0,\ldots,\mathbf{v}_{k-1}) = \alpha(\mathbf{x},\mathbf{v}_0,\ldots,\mathbf{v}_{k-1}) \tag{4.137}$$

For 1-forms it takes the simple form:

$$i_{\mathbf{x}} = \alpha(\mathbf{x}) \tag{4.138}$$

The interior product has similar properties to the exterior derivative (Definition 60) d such as antisymmetry

$$i_{\mathbf{x}}i_{\mathbf{y}}\alpha = -i_{\mathbf{y}}i_{\mathbf{x}}\alpha, \tag{4.139}$$

nilpotence

$$i_{\mathbf{x}}i_{\mathbf{x}} = 0, \tag{4.140}$$

and in conjunction with the exterior product (Definition 60) of a p-form α and a q-form β.

$$i_{\mathbf{x}}(\alpha\wedge\beta) = (i_{\mathbf{x}}\alpha)\wedge\beta + (-1)^k\alpha\wedge(i_{\mathbf{x}}\beta) \tag{4.141}$$

For these reasons it is also often called an antiderivative of degree -1.

With the given properties the Lie derivative (Definition 68), the exterior derivative (Definition 70), and the interior product (Definition 72) can be linked in the expression

$$\mathcal{L}_{\mathbf{x}}\omega = \mathrm{d}(i_{\mathbf{x}}\omega) + i_{\mathbf{x}}\mathrm{d}\omega, \tag{4.142}$$

which is known as Cartan's identity or also homotopy (Definition 32) formula.

The exterior derivative allows to qualify differential forms (Definition 59), where the nomenclature is provided in the following definitions.

Definition 73 (Closed form) *A form α (Definition 59) with vanishing exterior derivative (Definition 70)*

$$\mathrm{d}\alpha = 0 \tag{4.143}$$

is called a closed form.

Definition 74 (Exact form) *A form α (Definition 59) is called exact, if it can be expressed by the external derivative (Definition 70) of another form β*

$$\alpha = \mathrm{d}\beta \tag{4.144}$$

From the nilpotence of d it follows that every exact form (Definition 74) is closed.

4.6.6 Special Spaces

Having established definitions for all components required for the treatment of geometrical problems, they are now assembled into the final settings used for the considerations, which are due to their importance specially named.

The further qualification of differentiable manifolds (Definition 37) is by pairing them with specific tensor fields (Definition 65).

Definition 75 (Metric tensor) *A non-degenerate, symmetric bilinear form $g \in \mathcal{T}_2^0(\mathcal{V})$ is called a metric tensor.*

Definition 76 (Metric tensor field) *A tensor field (Definition 65) comprised entirely of metric tensors (Definition 75) is called a metric tensor field.*

The availability of a metric tensor field allows to define the following:

Definition 77 (Inner product) *A metric tensor field g on a differentiable manifold (Definition 37) defines a non-degenerate bilinear mapping*

$$(\cdot,\cdot) : TM_P \times TM_P \to \mathbb{R} \tag{4.145a}$$
$$(\mathbf{v},\mathbf{w}) = g_P(\mathbf{v},\mathbf{w}), \quad \forall \mathbf{v},\mathbf{w} \in TM_P, \forall P \in \mathfrak{M} \tag{4.145b}$$

for the tangent space at every point. This mapping is called the inner product on the manifold $\mathfrak{M}$.

Equipping a manifold with an inner product, defines the notion of orthogonality of two vectors in the tangential spaces at each of the points. This structure is of such significance, it warrants the following definition.

Definition 78 (Riemannian manifold) *The pair $(\mathfrak{M}, g)$ of a differential manifold $\mathfrak{M}$ (Definition 37) on which a positively definite metric tensor field g (Definition 76) is defined and hence an inner product is available, is called a Riemanninan manifold.*

Riemannian manifolds carry sufficient structure to define angles between vectors in each of the tangential spaces, but also allows for the definition of the concept of how far any two points of the manifold are apart, which is explored further in Section 4.8.

A Riemannian manifold, where all tangent spaces (Definition 48) are identical and which can therefore be covered by a single chart (Definition 36), is a very important special case encountered in everyday perception and has been the foundation of important developments in physics, as illustrated in Section 5.1.

In this case the structure can be simplified considerably, as can be experienced in the following two definitions.

Definition 79 (Affine space) *A set of points $\mathfrak{A}$ accompanied by a vector space $\mathcal{T}$ (Definition 16) is called an affine space, if the following assertions hold:*

- *A mapping assigning to every pair of points $P, Q \in \mathfrak{A}$ to an element $\mathbf{v} \in \mathcal{T}$ exists.*

$$\mathfrak{A} \times \mathfrak{A} \to \mathcal{T} \tag{4.146a}$$

$$\mathbf{v} = \overrightarrow{PQ} \in \mathcal{T} \tag{4.146b}$$

- *For every pair of a point $P \in \mathfrak{A}$ and vector $v \in \mathcal{T}$ there is exactly one point $Q \in \mathfrak{A}$ such that*

$$\mathbf{v} = \overrightarrow{PQ}. \tag{4.147}$$

- *For every three points $P, Q, R \in \mathfrak{A}$ it holds*

$$\overrightarrow{PQ} + \overrightarrow{QR} = \overrightarrow{PR}. \tag{4.148}$$

$\mathcal{T}$ is called the tangent space of $\mathfrak{A}$.

The nomenclature in the case of the affine space is reminiscent of the manifold, however, the previously locally varying structures are now constant over all of the considered space. With these requirements an affine space inherently supports the notion of parallel lines. It however lacks structure to define lengths and angles.

Adding the structure of an inner product (Definition 77) remedies this deficiency and results in:

Definition 80 (Euclidean space) *An affine space (Definition 79), where the tangent space $\mathcal{T}$ (Definition 48) is equipped with an inner product which defines a norm, is called an Euclidean space $\mathcal{E}$.*

Similarly to the symmetric metric structure (Definition 76) yielding the concepts of length and angles, a skew-symmetric structure has a distinct application as is illustrated in Section 5.3. The required structure begins by a simple definition.

Definition 81 (Symplectic form) *A non-degenerate, closed (Definition 73), skew-symmetric bilinear form ω is called a symplectic form.*

This definition is now used to qualify a manifold similarly to the Riemannian (Definition 78) case.

Definition 82 (Symplectic manifold) *The pair $(\mathfrak{M}, \omega)$ of a differentiable manifold $\mathfrak{M}$ (Definition 37), where a symplectic form ω is defined in every tangent space TM_P, is called a symplectic manifold.*

Due to the non-degeneracy together with the demand of skew-symmetry all symplectic manifolds necessarily have even dimension.

Definition 83 (Symplectic map) *A diffeomorphism (Definition 38)*

$$f : \mathfrak{M} \to \mathfrak{N} \tag{4.149}$$

is called a symplectic map or symplectomorphism, if $(\mathfrak{M}, \omega)$ and $(\mathfrak{N}, \eta)$ are symplectic manifolds (Definition 82). The symplectic form can be expressed as a pullback (Definition 44).

$$f^*\eta = \omega \tag{4.150}$$

Connected with symplectic spaces is an important theorem regarding the representation of the symplectic form. It is the basis for canonical representations in Section 5.3.

Definition 84 (Darboux's theorem) *Given a $2n$-dimensional symplectic manifold $(\mathfrak{M}, \omega)$ (Definition 82) the symplectic form ω (Definition 81) can always be rendered in the canonical form*

$$\omega = \sum_{i=1}^{n} \mathrm{d}p_i \wedge \mathrm{d}q_i \tag{4.151}$$

in a neighbourhood U for every point $x \in \mathfrak{M}$.

Definition 85 (Kähler manifold) *A complex manifold $\mathfrak{M}$ which has both a Riemannian and symplectic structure is called a Kähler manifold. It is required that the metric structure g and symplectic structure ω are compatible in the following manner:*

$$g(i\mathbf{x}, \mathbf{y}) = \omega(\mathbf{x}, \mathbf{y}) \tag{4.152}$$

Equivalently, the space may be viewed as a being equipped with a Hermitian inner product h composed as

$$h(\mathbf{x}, \mathbf{y}) = g(\mathbf{x}, \mathbf{y}) + \omega(\mathbf{x}, \mathbf{y}). \tag{4.153}$$

4.6.7 Peculiarities in Special Spaces

The existence of a non-degenerate bilinear form, metric (Definition 76) or symplectic (Definition 81), allows to define an isomorphism between the tangent and co-tangent bundles of a manifold. This is accomplished by associating to every vector $\mathbf{v}$ tangent to $\mathfrak{M}$ at a point $x \in \mathfrak{M}$ one form by

$$g_{\mathbf{v}}(\mathbf{w}) = g(\mathbf{v}, \mathbf{w}) \tag{4.154}$$

in the case of a Riemannian metric and

$$\omega_{\mathbf{v}}(\mathbf{w}) = \omega(\mathbf{v}, \mathbf{w}) \tag{4.155}$$

in the symplectic case. Thus a vector $\mathbf{v}$ defines a 1-form and is thereby linked to it, defining an isomorphism.

$$I : T^*M \to TM \tag{4.156}$$

The non-degenerate nature of the used bilinear form ensures that this relation also uniquely associates a vector with a 1-form. The existence of such a mapping is essential as it also provides a structure with which to translate different kinds of tensors into one another.

Furthermore it is possible to identify flows and their associated vector fields which preserve the metric or symplectic structures. Using the Lie derivative this reads

$$\mathcal{L}_{\mathbf{v}} g = 0 \tag{4.157}$$

in case of the a metric structure g. The vector fields meeting this requirement are called Killing vector fields [74]. In the symplectic case the expression is almost identical with the symplectic form taking the place of the metric

$$\mathcal{L}_{\mathbf{v}}\omega = 0. \tag{4.158}$$

Complying vector fields are called Hamiltonian in this case. It is useful to further examine the symplectic case by expanding the expression using the homotopy formula

$$\mathcal{L}_{\mathbf{v}}\omega = \mathrm{d}(i_{\mathbf{v}}\omega) + i_{\mathbf{v}}\mathrm{d}\omega = 0. \tag{4.159}$$

Since ω is closed and hence $\mathrm{d}\omega = 0$, it follows

$$\mathrm{d}(i_{\mathbf{v}}\omega) = 0 \tag{4.160}$$

thus indicating

$$i_{\mathbf{v}_f}\omega = -\mathrm{d}f, \tag{4.161}$$

where f is a function on the symplecetic manifold $\mathfrak{M}$. The function f generates the vector field $\mathbf{v}_f$. The definition of $\mathbf{v}_F$ in Equation 4.161 is implicit and may be converted to an explicit form using the isomorphism defined in Equation 4.156 to obtain

$$\mathbf{v}_f = I\mathrm{d}f. \tag{4.162}$$

This explicit form can be combined with the Lie bracket (Definition 69) to obtain additional structure.

Definition 86 (Poisson bracket) *The Poisson bracket is obtained by applying the Lie bracket for vector fields to an expression due to the established isomorphism (Equation 4.156).*

$$[I\mathrm{d}f, I\mathrm{d}g] = [\mathbf{v}_f, \mathbf{v}_g] = \mathbf{v}_{\{f,g\}} \tag{4.163}$$

The Poisson bracket or commutator of functions on a manifold $\mathfrak{M}$ is a bilinear, skew symmetric map satisfying the Jacobi identity.

A manifold $\mathfrak{M}$ along with a Poisson structure is called a Poisson manifold. Every symplectic manifold is also a Poisson manifold, while the reverse is not necessarily true, as the Poisson structure may be degenerate, thus being more general. In the symplectic case, the Poisson structure and the symplectic structure are inverse to each other.

$$P \circ \omega = -\mathrm{id} \tag{4.164}$$

4.7 Integration

So far differential structures have been explored which have proved essential in the development of physical sciences. Differential descriptions are, however, not the only descriptions available. In the following the concepts for the dual operation of differentiation are outlined, which not only provide different descriptions as in Section 5.6, but also inspire different methodologies. It requires a few very basic definitions on which to build.

Definition 87 (Borel set) *A set which is obtained from open sets of a topological space (Definition 29) using a countable number of unions, intersections and complements is called a Borel set.*

Another term is introduced in order to simplify the subsequent Definition 89.

Definition 88 (σ-algebra) *A non-empty collection Σ of subsets of a set χ such that it is closed under countably many unions, intersections, or complements of elements is called a σ-algebra.*

In the case of dealing with topological spaces a particular σ-algebra, which contains all open sets, is of special interest.

Definition 89 (Borel-σ-algebra) *The set of all Borel sets (Definition 87) is the Borel-σ-algebra.*

Definition 90 (Measurable space) *The pair (χ, Σ) of a set χ along with a σ-algebra Σ is called a measurable space.*

Definition 91 (Signed measure) *A mapping*

$$\mu : \Sigma(\chi) \to \mathbb{R} \tag{4.165}$$

assigning a real number to any subset contained in the σ-algebra Σ (Definition 88) over the set χ, such that for any collection of disjoint sets $S_1, S_2, \ldots, S_n$ in Σ it holds that

$$\mu(\bigcup_i S_i) = \sum_i \mu(S_i) \tag{4.166}$$

is called a signed measure.

When the mapping μ is restricted to values ≥ 0, it is called simply a measure.

Definition 92 (Measure space) *A measurable space (χ, Σ) (Definition 90) along with a measure μ forming the triple of a set χ, a σ-algebra (Definition 88) and a measure μ (χ, Σ, μ) form a measure space.*

Providing a measure on a Borel-σ-algebra (Definition 89) is an essential step of defining integration on topological spaces (Definition 29) such as manifolds (Definition 35).

The Lebesgue measure is a particular choice for a measure which is related to an interval of $\mathbb{R}$ in the form

$$I = [a, b] \in \mathbb{R} \qquad \lambda(I) = b - a. \tag{4.167}$$

The generalization to subsets of $\mathbb{R}^n$ is then simply obtained by

$$I^n = [a_1, b_1] \times \ldots \times [a_n, b_n] \in \mathbb{R}^n \qquad \lambda(I) = \prod_{i=1}^{n} (b_i - a_i) \tag{4.168}$$

Definition 93 (Almost everywhere) *A property is said to apply almost everywhere in a measure space (χ, Σ, μ), if the complement of the set, where the property does not hold, has measure 0.*

Using a measure it is possible to introduce the notion of integration of functions. The first step is to examine functions of the form

$$f : D \to B \qquad B = \{v_0, v_1, \ldots, v_n\}, \quad v_i \geq 0 \tag{4.169}$$

resulting in only a limited, finite number of results v_i, also called step functions. An integral of f using a measure μ is then obtained by

$$\int_D f \mathrm{d}\mu = \sum_{i=1}^{n} v_i \mu(\{x \in D | f(x) = v_i\}) \tag{4.170}$$

Definition 94 (Integral of a function) *Using the step functions, the integral of functions of the form*

$$g : D \to \mathbb{R} \tag{4.171}$$

is then defined as

$$\int_D g \mathrm{d}\mu = \sup \left\{ \int_D f \mathrm{d}\mu \mid 0 \leq f \leq g \right\}. \tag{4.172}$$

It follows from this definition that the integral does not change its value as long as only function values associated with sets of measure 0 are changed.

The integral obtained by using a Lebesgue measure is called the Lebesgue integral, which while not being the most general notion of integration available, shall be sufficient for matters at hand.

In the context of manifolds the Borel-σ-algebra (Definition 89) can be presented very powerfully by the concept of chains [78], which provide convenient methods of expression for operations on the integration domain.

It should be noted that the integration domain and the domain of the function being integrated must agree for the integral to be well defined (Definition 94). An example of such agreement, which may still be embedded in a greater structure of a manifold (Definition 35) is given next.

Definition 95 (Curve integrals) *The integral of a 1-form (Definition 50) along a segment of a curve φ (Definition 46) connecting two points $P = \varphi(a)$ and $P = \varphi(b)$ is simply defined as*

$$\int_\varphi \omega = \int_{[a,b]} \omega(\dot{\varphi}(t)) \mathrm{d}t. \tag{4.173}$$

One of the most important properties of integration is that operations on the integration domain may be mapped to the integrand by pullback (Definition 44).

$$\int_{\phi(\Omega)} \alpha = \int_\Omega \phi^* \alpha \tag{4.174}$$

This general case specializes to a particularly important case for ϕ being the boundary operation ∂ called Stokes' theorem, which asserts that when integrating a p-form on a p-integration domain it follows that

$$\int_\Omega \mathrm{d}\alpha = \int_{\partial\Omega} \alpha. \tag{4.175}$$

4.8 Distances

To define the distance between two objects is a concept universally graspable on an intuitive level. In mathematical terms the notion of distance as reinforced by our every day experiences can be expressed using the simple notion of an affine space (Definition 79) and the lengths of the vectors inhabiting the vector space, which is part of the affine space's definition. This simple notion fails, however, when considering distances between geographical locations, since the shape of the world can not be modelled as an affine space, but necessarily takes the shape of a manifold (Definition 37), even if this manifold is embeddable into an affine space. A simple

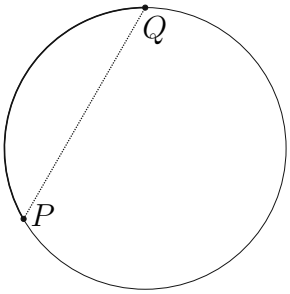


Figure 4.7: The distance within the manifold differs from the distance obtained, when it is embedded into an affine space.

example of this circumstance is presented in Figure 4.7. The manifold of a circle is embedded in the affine space of the plane. Considering the simple case, where the circumference of a circle equals $2\pi r$, the maximum distance between any two points on the circle is πr, while in contrast the embedded case has a maximum distance of a mere $2r$.

Therefore a more abstract and general approach is presented in order to alleviate this issue. Before the length can be assessed, it should be noted that the question whether any two points are connected or not, is a question of topology (Definition 28), not of the geometry built on top of this topology. The connection established by topology manifests itself in the existence of curves (Definition 45) connecting the two points under consideration. As the number of curves

$$\forall \gamma_i : P, Q \in \gamma_i \qquad G = \{\gamma_i\} \tag{4.176}$$

joining the points P and Q is infinite, even when the set of points involved is unique, since different parametrization define different curves, an additional selection criterion, the length of a curve γ_i, which is independent of the parametrization, is required to select a particular curve and assign the distance $d(P, Q)$. The length of a segment of a curve may be defined using integration (Definition 95) along the curve.

$$\gamma(t_0) = P, \quad \gamma(t_1) = Q, \qquad P, Q \in \mathfrak{M} \tag{4.177a}$$

$$\operatorname{len}\left(\gamma(t_0 \to t_1)\right) = \int_{[t_0, t_1]} \Gamma(\gamma(t))\, \mathrm{d}t \tag{4.177b}$$

The distance is now readily obtained as the result of a minimization of the lengths of all the available curves G as described in Equation 4.176 connecting the desired points

$$d(P, Q) = \inf\left(G\right). \tag{4.178}$$

As this formulation uses the infimum, it allows for the simple inclusion of open sets, which would otherwise pose a problem. Applying this definition allows not only for the proper treatment of distances within Riemannian manifolds (Definition 78), which enable proper treatment of distances along the globe, but also encompasses settings used in the theory of relativity [4]. In these cases the term Γ is connected to the metric tensor field (Definition 76).

4.9 Integral Equations

The integral equation under consideration in this context is the Fredholm integral equation of the second kind, which is commonly given in the form

$$\phi(s) = f(s) + \lambda \int_B K(s,t)\phi(t)\mathrm{d}t. \tag{4.179}$$

$K(s,t)$ is the kernel of the integral equation. As the solution $\phi(s)$ appears on both sides of Equation 4.179, it can be inserted into itself, thus yielding

$$\begin{aligned}\phi(s) =& f(s) + \lambda \int_a^b K(s,t) \left(f(s) + \lambda \int_B K(s,t)\phi(t)\mathrm{d}t \right) \mathrm{d}t = & \text{(4.180a)}\\ =& f(s) + \lambda \int_B K(s,t) f(s)\mathrm{d}t + \lambda^2 \int_B K(s,t) \left(\int_B K(s,t)\phi(t)\mathrm{d}t \right) \mathrm{d}t. & \text{(4.180b)}\end{aligned}$$

Thus the function $\phi(s)$ takes the form of a series

$$\phi(s) = \phi_0(s) + \phi_1(s)\lambda + \phi_2(s)\lambda^2 + \ldots = \sum_{i=0}^{\infty} \phi_i(s)\lambda^i \tag{4.181}$$

where

$$\phi_n = \int_B K(s,t)\phi_{n-1}(t)\mathrm{d}t \qquad \phi_0(s) = f(s). \tag{4.182}$$

While this recursion relation is straightforward, an explicit expression for the $\phi_n(s)$ depending only on $\phi_0(s) = f(s)$ is favourable. It can be obtained by rewriting the recursion with a focus on the kernels instead of the functions to read.

$$\phi_n = \int_B K_n(s,t) f(t)\mathrm{d}t \tag{4.183}$$

Now the $K_n(s,t)$ are given as

$$K_n(s,t) = \int_B K_{n-1}(s,t_1)K(t_1,t)\mathrm{d}t \qquad K_1(s,t) = K(s,t) \tag{4.184}$$

which can be generalized to

$$K_{p+q}(s,t) = \int_B K_p(s,t_1)K_q(t_1,t)\mathrm{d}t. \tag{4.185}$$

This representation is known as iterated kernels. The resulting series is known as a Neumann series or the resolvent series of the Equation 4.179 and takes the form

$$R(s,t,\lambda) = K_1(s,t) + K_2(s,t)\lambda + K_3(s,t)\lambda^2 + \ldots = \sum_{i=0}^{\infty} K_{i+1}(s,t)\lambda^i. \tag{4.186}$$

Thus the expression of Equation 4.181 may also be expressed as

$$\phi(s) = f(s) + \lambda \int_B R(s,t,\lambda) f(t) \mathrm{d}t = f(s) + \lambda \sum_{i=0}^{\infty} \int_B K_{i+1}(s,t)\lambda^i f(t) \mathrm{d}t. \tag{4.187}$$

4.10 Probability

While determinism was pervasive through most of the development of classical physics, it could not be maintained and uncertainty was even embraced as a central theoretical feature [79][80], as such probabilities and statistics have taken an important part among scientific modelling.

Definition 96 (Probability measure) *The measure space (χ, Σ, P) (Definition 92), where the measure P (Definition 91) has the properties*

$$0 \leq P(\Sigma(\chi)) \leq 1, \tag{4.188a}$$
$$P(\emptyset) = 0, \tag{4.188b}$$
$$P(\chi) = 1, \tag{4.188c}$$

is called a probability measure. The values assigned by the measure to the subsets are called probabilities.

From the construction of probability as a measure it follows for a configuration as shown in Figure 4.8 that

$$P(C \backslash D) = P(C) - P(D), \tag{4.189a}$$
$$P(A \cup C) = P(A) + P(C), \tag{4.189b}$$
$$P(A \cup B) = P(A) + P(B) - P(A \cap B), \tag{4.189c}$$

which illustrates a few basic properties of probabilities which follow directly from the use of a σ-algebra (Definition 88) in the definition of a probability measure (Definition 96). The fact that the focus is not always on the introduction of a measure on a space, but rather on the resulting structure as a whole, warrants the following definition.

Definition 97 (Probability space) *A measure space (χ, Σ, P) (Definition 92), where P is a probability measure (Definition 96), is called a probability space.*

Definition 98 (Random Variable) *Given a probability space (χ, Σ, μ) (Definition 97) and a measurable space $(\bar{\chi}, \bar{\Sigma})$ (Definition 90) a mapping of the form*

$$X : \chi \to \bar{\chi} \tag{4.190}$$

is called a random variable, if the preimages of all subspaces of the σ-algebra $\bar{\Sigma}$ (Definition 88) are part of the σ-algebra Σ

$$X^{-1}(b) \in \Sigma, \quad \forall b \in \bar{\Sigma}. \tag{4.191}$$

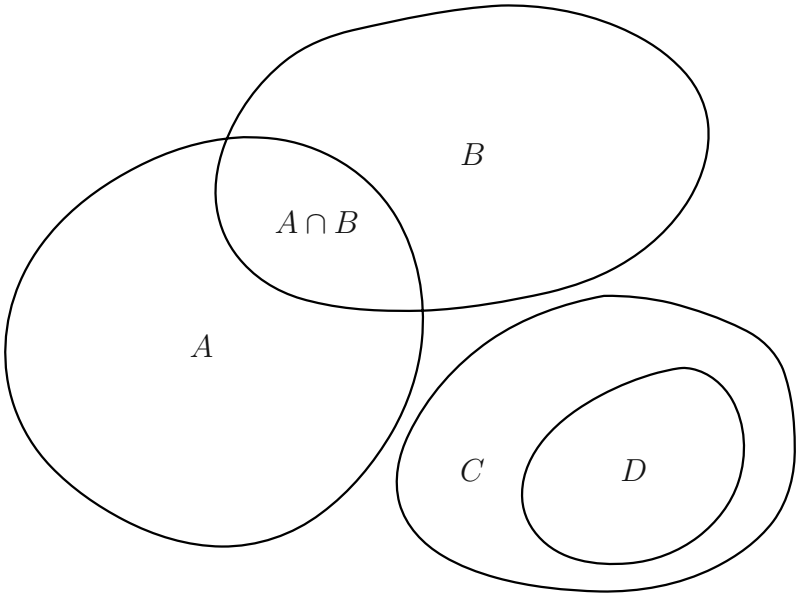


Figure 4.8: Sets with and without intersection.

The following definition facilitates the characterization of random variables and thus eases their handling.

Definition 99 (Probability distribution function) *Given a random variable X (Definition 98) joining the probability space (χ, Σ, μ) (Definition 97) and the measurable space $(\bar{\chi}, \bar{\Sigma})$ (Definition 90), it can be used to define a mapping $\bar{\mu}$ of the following form:*

$$\begin{array}{ccc} \Sigma & \xrightarrow{X} & \bar{\Sigma} \\ \downarrow{\scriptstyle \mu} & \swarrow{\scriptstyle \bar{\mu}} & \\ [0;1] & & \end{array} \tag{4.192}$$

$$\bar{\mu} = \mu(X^{-1}(b)) = \mu \circ X^{-1}(b), \quad \forall b \in \bar{\Sigma} \tag{4.193}$$

$\bar{\mu}$ is an induced measure on $\bar{\chi}$ called the probability distribution function P, also called the cumulative probability function.

A concept closely related to the probability distribution function which allows for a local description, is given next.

Definition 100 (Probability density function) *Given a probability distribution function P (Definition 99) the probability density function p is defined as satisfying the relation*

$$P(X) = \bar{\mu}(X) = \int_{X^{-1}(\bar{\chi})} \mathrm{d}\mu = \int_{\bar{\chi}} p \mathrm{d}\bar{\mu}. \tag{4.194}$$

Definition 101 (Expectation value) *Given a probability space (χ, Σ, μ) (Definition 97) and a random variable X (Definition 98) the expectation value $\langle X \rangle$ is defined by the prescription*

$$\langle X \rangle = \int_{\chi} X \mathrm{d}\mu. \tag{4.195}$$

In case a probability distribution (Definition 100) is given, the expectation value also takes the shape

$$\langle X \rangle = \int_{\bar{\chi}} X p \mathrm{d}\bar{\mu}. \tag{4.196}$$

Among the infinite number of conceivable probability measures a few select are provided in the following, since they have uses in Section 7.2.

Definition 102 (Uniform distribution) *A probability distribution P (Definition 99) on a space χ, which has a constant probability density function p (Definition 100) is called a uniform distribution.*

$$p = \text{const} \tag{4.197}$$
$$P(\chi) = 1 \tag{4.198}$$

Definition 103 (Exponential distribution) *The exponential distribution is associated to the cumulative distribution function (Definition 99).*

$$P(x) = 1 - e^{-\lambda x}, \quad \forall x \geq 0 \tag{4.199}$$

From this the probability density function p (Definition 100) can be derived to read

$$p(x) = \lambda e^{-\lambda x}, \quad \forall x \geq 0 \tag{4.200}$$

Definition 104 (Normal distribution) *A probability distribution function P with a probability density function of the shape*

$$p(x) = a e^{b(x-c)^2}, \qquad a, b, c \in \mathbb{R}, b < 0 \tag{4.201}$$

is called normal or Gaußian. The factors are often associated the variance σ^2, and the mean value μ by the expressions

$$a = \frac{1}{\sqrt{2\pi}\sigma} \tag{4.202a}$$
$$b = -\frac{1}{2\sigma^2} \tag{4.202b}$$
$$c = \mu. \tag{4.202c}$$

If this is the case and furthermore $\mu = 0$ and $\sigma^2 = 1$, it is commonly called standard normal distribution.

The importance of the normal distribution can be linked to the central limit theorem, which states that a series of independently distributed random variables approaches normal distribution in the limit and thus can be approximated using normal distribution.

The normal distribution can also be used to define other distributions as in the following.

Definition 105 (Lognormal distribution) *A random variable Y, which is obtained from a normally distributed (Definition 104) variable X by*

$$Y = e^X \tag{4.203}$$

is called log-normally distributed.

Chapter 5

Dynamics

ποταμοῖσι τοῖσιν αὐτοῖσιν
ἐμβαίνουσιν ἕτερα καὶ ἕτερα
ὕδατα ἐπιρρεῖ

῾Ηράκλειτος ὁ ᾽Εφέσιος

The terminology and methodologies developed for use in mechanics have left a mark on many other aspects of physical investigation, especially since mechanics have been a mainstay of physical development. While the first observations and musings concerning mechanics may be traced back as far as ancient Greece [81][82], the foundations have been laid and formalized by none other than Sir Isaac Newton [83]. His work greatly contributed to if not, along with Leibniz [84] defined, the field of calculus but also defined a new era of mechanics. The high level mathematical formalisms described in Chapter 4 are employed in the following to formulate models of physical reality up to the abstract setting of phase spaces. The first step in this endeavour is to be aware of the used model of the physical world.

The process of establishing an acquaintance with the models of the physical world can be seen as both following a historical path as well as increasing the level of abstraction. Beginning with Newton physics was given a solid enough theoretical background to separate itself from philosophy[1]. Section 5.1 describes Newton's approach and also introduces his theoretical concept of the structure of world, which shall be used throughout this whole work. Building on these foundations, the Lagrangian formalism is introduced in Section 5.2, which, while having been introduced on purely algebraic considerations[2], today already indicates geometric structures. Carrying abstractions further Hamiltonian mechanics, which are described in Section 5.3, show mechanics as geometry in phase space. Upon having described the structures of phase spaces, statistical descriptions, as shown in Section 5.4, are employed in order to deal with an exceeding number of degrees of freedom. How the statistical description can be linked to the tangible world we expect is discussed in Section 5.5. Section 5.6 revisits the evolution in phase space but moves from a local to an integral description. The strength of the concept of a phase

[1] Note the name of his famous work still makes reference to philosophy.
[2] It is said Lagrange prided himself with this fact.

space is shown in Section 5.7, as it is applied in the field of quantum mechanics, which also intends to illustrate that the same structures, once found, may be applicable beyond their initial context, if proper links can be found. The whole chapter should also serve to show that there are several different levels of abstraction and formalism to choose from in the field of physics. It is hoped that the field of scientific computing would take this as an example to also extend the number of choices available.

5.1 Newton's Mechanics and World

Drawing on considerations and abstracts of thinkers such as Thomas Aquinas and Descartes Newton was essential in formulating the stage from which to arrive at the abstraction of phase space. To this end it is important to recall that Newton formulates his famous three laws or axioms of motion:

- *"Corpus omne perseverare in statu suo quiescendi vel movendi uniformiter in directum, nisi quatenus a viribus impressis cogitur statum illum mutare"* – a body remains in a state of rest or uniform motion unless a force acts upon it.
- *"Mutationem motus proportionalem esse vi motrici impressæ, & fieri secundum lineam rectam qua vis illa imprimitur"* – the change of a motion is proportional to and at a right angle to the impressed force.
- *"Actioni contrariam semper & æqualem esse reactionem: sive corporum duorum actiones in se mutuo semper esse æquales & in partes contrarias dirigi"* – to every action there is always an equal action opposed to it: when two bodies act on each other, their actions are always equal and in opposition to each other.

He thus formulated the motion of bodies (or particles) focusing on the concept of forces which are set in a space using an absolute time which is mathematically expressed as an affine space $\mathcal{A}^{n+1}$ (Definition 79) n stands for the degrees of freedom and the additional dimension is added to accommodate time. The absolute nature of time is modelled as a scalar field

$$f : \mathcal{A}^{n+1} \to \mathbb{R}, \tag{5.1}$$

which assigns the time to each point $P \in \mathcal{A}^{n+1}$ (symbolizing an event).

$$t = f(P). \tag{5.2}$$

The differential (Definition 70) of this scalar field f (Definition 61), df is constant in correspondence to the absolute nature of time in Newton's setting and defines a mapping

$$\mathrm{d}f : \mathcal{T}(\mathcal{A}^{n+1}) \to \mathbb{R} \tag{5.3}$$

of the tangent space $\mathcal{T}(\mathcal{A}^{n+1})$ (Definition 48), which determines time spans between two events. Furthermore, the kernel (Definition 22) of df is an n-dimensional Euclidean (Definition 80) subspace $\mathcal{E}^n$ of $\mathcal{A}^{n+1}$ corresponding to all simultaneously occurring events. An affine space with this additional structure is called a Galilean space [72]. A Galilean space has a natural fiber

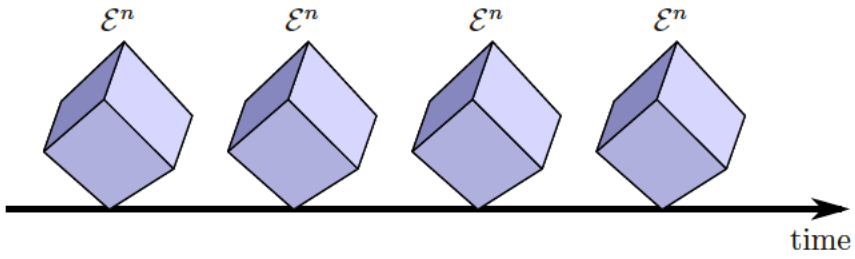


Figure 5.1: Fiber structure of a Galilean space.

bundle (Definition 40) structure of a one-dimensional base space (time), with n-dimensional fibers (space) as is illustrated in Figure 5.1. Each of the fibers contains events which are said to occur at the same time. The evolution of an entity in the affine space with the passing of time is represented as a section of this fiber bundle. While it is simple to naturally denote a one-dimensional base space, with n-dimensional fibers, the reverse is not the case, as while time is defined as absolute, thusly globally defining simultaneous events and thus fixed points in time, it is not clear what fixed points in space should be.

A transformation $\sigma : \mathcal{A}^{n+1} \to \mathcal{A}^{n+1}$, which leaves the structure of a Galilean space invariant, is called a Galilean transformation. Coordinate systems

$$\varphi : \mathcal{A}^{n+1} \to \mathbb{R} \times \mathbb{R}^n, \tag{5.4}$$

in which Newton's axioms hold parametrize the n-dimensional subspaces and are called inertial. Galilean transforms provide mappings between different inertial systems.

In this setting forces and the connected accelerations may be expressed as functions of the form:

$$F : \mathcal{E}^n \times \mathcal{T}(\mathcal{E}^n) \times \mathbb{R} \to \mathbb{R}^n \tag{5.5}$$

$$\ddot{\mathbf{r}} = \mathbf{F}(\mathbf{r}, \dot{\mathbf{r}}, t)\ \mathbf{r} \in \mathcal{E}^n, \dot{\mathbf{r}} \in \mathcal{T}(\mathcal{E}^n) \tag{5.6}$$

A system is determined uniquely by specifying the initial conditions at time t_0 for the position $\mathbf{r}(t_0)$ and the velocity $\dot{\mathbf{r}}(t_0)$. Together with Newton's three laws of motion a multitude of problems can already be considered.

The relevance of Galilean transfoms is connected to the evolution of physical quantities in the system. The laws how to determine (measure) a quantity do not depended on time, thus the evolution of quantities is described by

$$A\left(\mathbf{r}, \dot{\mathbf{r}}, \ddot{\mathbf{r}}, t\right) = A\left(\mathbf{r}(t), \dot{\mathbf{r}}(t), \ddot{\mathbf{r}}(t)\right). \tag{5.7}$$

As Galilean transforms do not interfere with this property, they are of special interest.

However, the formalism has been developed within the Cartesian setting of affine spaces. This leads to complications as the problems to which the formalism is applied increases in sophistication and adds complexity by being formulated directly in coordinate expressions. This leads to a strong mix of the representation in coordinates and the physical content, which is independent from the choice of coordinates. The search to overcome these limitations leads to a generalized reformulation of classical mechanics in the form of Lagrangian mechanics which recasts the single second order equation as two first order equations.

5.2 Lagrangian Formalism

As Newton's ideas and methods were adopted and applied to address problems of increasing sophistication its limits were also made apparent by the increase in the complexity of the used notation. Consequently, further development had the goal of addressing these shortcomings by simplifying calculations and establishing the foundations for a wider variety of treatable problems. Therefore, computational methods and notations to handle the increasing intricacies of the posed problems have been developed by Euler [64], Laplace [85] and Lagrange [86], without altering the fundamentals or the nature of Newton's world. Where Newton's description focuses on forces and accelerations using second order differential equations, the Lagrangian formulation recasts them as a set of first order differential equations. It does so by redirecting attention from the explicit treatment of forces and accelerations to expressions linked to energy. Starting from an expression for Newton's second law

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = \mathbf{f}, \tag{5.8}$$

the term for the force term may be decomposed into a part which can be represented using a potential and a remaining part.

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = -\frac{\partial}{\partial r} V + \hat{\mathbf{f}} \tag{5.9}$$

Furthermore, the momentum $\mathbf{p}$ is linkable to kinetic energy T and expressible as

$$\mathbf{p} = \frac{\mathrm{d}}{\mathrm{d}\dot{\mathbf{r}}} T(\dot{\mathbf{r}}). \tag{5.10}$$

Inserting into (5.9) yields

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}}{\mathrm{d}\dot{\mathbf{r}}} T(\dot{\mathbf{r}})\right) + \frac{\partial}{\partial r} V = \hat{\mathbf{f}}. \tag{5.11}$$

The two terms of energy in this formulation are collected into a single expression as the difference of kinetic and potential energy

$$L = T - V \tag{5.12}$$

called the Lagrangian. As such this Lagrangian contains components from the base manifold (Definition 37) as well as the tangent spaces (Definition 48) and is thus a mapping.

$$L : \mathcal{TM} \to \mathbb{R} \tag{5.13}$$

Based on the Lagrangian, a motion takes the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial}{\partial \dot{\mathbf{r}}} L\right) - \frac{\partial}{\partial r} L = \hat{\mathbf{f}}, \tag{5.14}$$

which is called the Euler-Lagrange equation. While the Euler-Lagrange equation may seem to add to the overall complexity, the step to describe the evolution of a system using the scalar quantities of energy results in an overall simplification as this notation remains invariant to a larger

group of transformations and it becomes easier to apply constraints. The configuration space is now generalized to a manifold, so that the coordinates which previously described a location in the physical world are now extended to abstract degrees of freedom. These abstracted degrees of freedom are no longer required to have physical dimensions such as position or velocity, but may in fact be dimensionless. Regardless of the physical dimension associated to the degrees of freedom, they are specified given as $\mathbf{q}$ and $\dot{\mathbf{q}}$, where $\mathbf{q}$ corresponds to a base manifold $\mathfrak{M}$, $\dot{\mathbf{q}}$ is from the tangent space TM. Thus this description can be summarized to take place in the tangent bundle TM. Where previously a location, a velocity and an acceleration were required, it is now sufficient to provide a point in this tangent bundle, called the configuration space. This does not change the Galilean structure of the world, however.

A trajectory which is used to describe a motion within the manifold $\mathfrak{M}$ (Definition 35) may be associated with a curve (Definition 45) in the tangent bundle TM (Definition 52) of $\mathfrak{M}$ using a lift (Definition 26). Among all of the possible choices to lift a curve γ, the one of the form

$$\gamma \in \mathfrak{M} \to (\gamma, \dot{\gamma}) = \bar{\gamma} \in TM, \tag{5.15}$$

which associates at every point the curve passes through its tangent vector, is called a natural lift.

The curve to be lifted connects a starting point to an end point, thus changing the description from an initial value problem, to a boundary value problem. In order to appreciate the geometric nature of Lagrange's formalism a short exploration of the concepts behind the distance of two points in a more traditional geometric setting is explored.

Using the clearly geometric construction presented in Section 4.8, with the Lagrangian L taking the place of the metric field, it is possible to evaluate the Lagrangian along a lifted curve and, again using the notion of a curve integral (Definition 95), assign values, called actions, to sections of any curve in the following fashion

$$S = \int_{[t_0,t_1]} L(\bar{\gamma}(t), t)\mathrm{d}t = \int_{[t_0,t_1]} L(\gamma, \dot{\gamma}, t)\mathrm{d}t = \int_{[t_0,t_1]} L(q, \dot{q}, t)\mathrm{d}t, \tag{5.16}$$

where the parameters t_0 and t_1 correspond to the points P and Q as indicated in Figure 5.2. In mimicking the definition of distance, this allows to distinguish the various trajectories from each other, thus providing a selection criterion in the form of the extrema which can be derived using a simple homotopy (Definition 32) for variation, as can be shown in a short calculation.

$$S = \frac{\mathrm{d}}{\mathrm{d}s}\Big|_{s=0} \int_{[t_0,t_1]} L(\gamma + s\eta, \dot{\gamma} + s\dot{\eta})\mathrm{d}t = 0 \tag{5.17}$$

Setting

$$\alpha = \gamma + s\eta, \tag{5.18}$$

it follows

$$S = \frac{\mathrm{d}}{\mathrm{d}s}\Big|_{s=0} \int_{[t_0,t_1]} L(\alpha, \dot{\alpha})\mathrm{d}t = \int_{[t_0,t_1]} \left[\frac{\partial}{\partial \gamma} L(\gamma, \dot{\gamma}) \frac{\partial \gamma}{\partial t} + \frac{\partial}{\partial \dot{\gamma}} L(\gamma, \dot{\gamma}) \frac{\partial \dot{\gamma}}{\partial t} \right] \mathrm{d}t = 0. \tag{5.19}$$

Upon shortly examining the last term in this expression, it is found that

$$\int_{[t_0,t_1]} \frac{\partial}{\partial \dot{\gamma}} L(\gamma, \dot{\gamma}) \frac{\partial \dot{\gamma}}{\partial t} \, \mathrm{d}t = - \int_{[t_0,t_1]} \frac{\partial}{\partial t} \left(\frac{\partial}{\partial \dot{\gamma}} L(\gamma, \dot{\gamma}) \right) \frac{\partial \gamma}{\partial t} \, \mathrm{d}t, \tag{5.20}$$

which, when applied, finally results in the expressions

$$\int_{[t_0,t_1]} \left[\frac{\partial}{\partial t} \left(\frac{\partial}{\partial \dot{\gamma}} L(\gamma, \dot{\gamma}) \right) - \frac{\partial}{\partial \gamma} L(\gamma, \dot{\gamma}) \right] \dot{\gamma} \, \mathrm{d}t = 0, \tag{5.21}$$

which can only hold in general, if the integrand itself vanishes. This condition is found to be equivalent to the Euler-Lagrange Equation 5.14. This means that from all possible curves γ_i, which pass through the given points P and Q, as shown in Figure 5.2, the actual realization can be found by the minimization of the action defined in Equation 5.16 as in this case the equations of motion in the form of the Euler-Lagrange Equation are met.

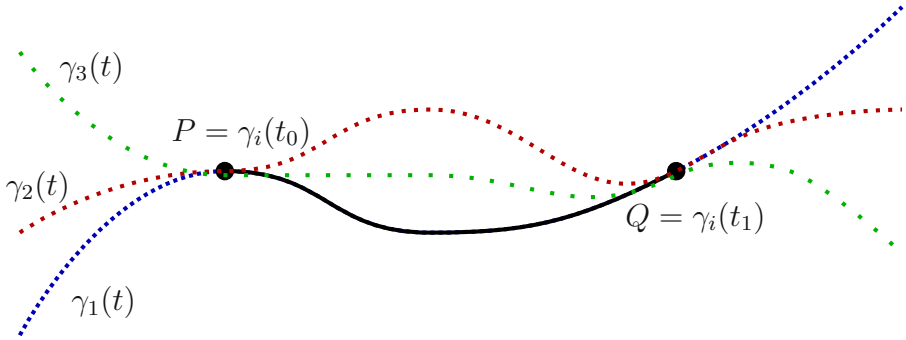


Figure 5.2: Lagrangian evaluated on sections of curves connecting the points P and Q.

5.3 Hamiltonian Formalism

While the Lagrangian formalism already provides extensive capabilities to handle elaborate problems, further refinement is possible resulting in the so called phase space, which also makes concepts available, which can even be carried beyond the confines of classical mechanics.

Motivated by the aim to find a simpler, more symmetric expression of the equation of motion as given in Equation 5.14, it is fruitful to utilize the fact from the settings of the Lagrangian in Equation 5.10 and Equation 5.12, from which follows that the momentum is expressible as

$$\mathbf{p} = \frac{\partial L}{\partial \dot{\mathbf{q}}} \, . \tag{5.22}$$

Adopting this momentum to express the degrees of freedom previously expressed by $\dot{\mathbf{q}}$ using a Legendre map (Definition 53) results in

$$H(\mathbf{q}, \mathbf{p}, t) = \langle \mathbf{p}, \dot{\mathbf{q}} \rangle - L(\mathbf{q}, \dot{\mathbf{q}}, t), \tag{5.23}$$

thus defining a Hamiltonian corresponding to a given Lagrangian. Contrary to the Lagrangian, which is a function on the tangent bundle TM, the Hamiltonian is a function on the co-tangent bundle T^*M, which is called the phase space.

$$H(\mathbf{q}, \mathbf{p}) : T^*M \to \mathbb{R} \tag{5.24}$$

This is the defining expression in the Hamiltonian formalism. Furthermore, the cotangent bundle is equipped with a symplectic structure (see Definition 81), which turns the cotangent bundle T^*M of an n dimensional base manifold into a symplectic manifold (Definition 82) of dimension $2n$. Among the great strengths of the Hamilton formalism is the fact that Hamilton's equations can be expressed using this geometric structure inherent to the phase space.

The symplectic nature of the manifold used ensures that it is always possible, as asserted by Definition 84, to find coordinates such that

$$[\mathbf{p}, \mathbf{p}] = 0, \tag{5.25a}$$
$$[\mathbf{q}, \mathbf{q}] = 0, \tag{5.25b}$$
$$\left[p_i, q^j\right] = \delta_i^j, \tag{5.25c}$$

called canonical coordinates.

The equations of motion as observed in the Lagrangian case take on the simpler, almost symmetric form

$$\dot{\mathbf{p}} = -\frac{\partial H}{\partial \mathbf{q}} \tag{5.26}$$
$$\dot{\mathbf{q}} = \frac{\partial H}{\partial \mathbf{p}} \tag{5.27}$$

called Hamilton's equations. Together they define a vector field $\mathbf{v}_H$ (Definition 62) along with associated integral curves $\gamma_H(t)$ (Definition 63), which all taken together define the phase flow Φ_t (Definition 67). This phase flow is a symplectomorphism (Definition 83), thus leaving the symplectic structure (Definition 81) of the manifold invariant.

The vector field due to the Hamiltonian H can be expressed utilizing the geometrical structure (compare Equation 4.162) inherent to the phase space.

$$\mathbf{v}_H = I\mathrm{d}H = \dot{\gamma}_H \tag{5.28}$$

This vector field provides the opportunity to express the evolution of the system under investigation as the parameter t changes. In case of a function f, which measures a quantity of interest in the phase space

$$f(\mathbf{q}, \mathbf{p}) \in \mathbb{F}(T^*M) : T^*M \to \mathbb{R}, \tag{5.29}$$

the evolution with regard to the parameter t due to the Hamiltonian and its own dependence on the parameter is then simply given by

$$\dot{f} = \mathbf{v}_H(f) + \frac{\partial f}{\partial t}, \tag{5.30}$$

recalling that a vector field is a mapping of the form $\mathbb{F}(T^*M) \to \mathbb{F}(T^*M)$ (see Equation 4.109), which is usually given explicitly using the Poisson bracket (Definition 86) in the form

$$\dot{f} = \frac{\mathrm{d}f}{\mathrm{d}t} = \{f, H\} + \frac{\partial f}{\partial t}\,, \tag{5.31}$$

which indicates how the geometry of the system, defined by the Hamiltonian, is responsible for its evolution.

5.4 Statistical Description – Boltzmann's Equation

So far little heed was paid to the number of entities under consideration as the formalisms are abstract enough to deal with a scaling of the degrees of freedom. When considering N identical but distinct entities in a three-dimensional setting, the associated phase space is simply a Cartesian product (Definition 3) of the single phase spaces, thus forming an overall space of dimension $6N$. While the formalism does not hinder the specification of problems encompassing multiple entities, it becomes increasingly difficult to obtain solutions. Furthermore, when turning to systems composed of particularly large numbers of entities, such as molecules or atoms in gases, it becomes quite unfeasible to deal with them directly, as the sheer amount of boundary and/or initial conditions becomes prohibitive.

Therefore, a description offering a reduction of the overwhelming degrees of freedom is called for. This can be demonstrated using a basic equation describing the evolution of N particles using a density depending on the N phase space coordinates and time

$$\rho(z_1, \ldots, z_N, t) : T^*M^N \times \mathbb{R} \to \mathbb{R} \qquad z_i = (\mathbf{q}, \mathbf{p}) \tag{5.32}$$

and a Hamiltonian H, which depends on all of the particles.

$$H : T^*M^N \to \mathbb{R} \tag{5.33a}$$

$$\frac{\partial \rho}{\partial t} + \{\rho, H\} = 0 \tag{5.33b}$$

By averaging by means of integration (Definition 94) an n particle distribution function is obtained

$$\rho_n(z_1, \ldots, z_n, t) = \int \rho(z_1, \ldots, z_N, t)\mathrm{d}z_{n+1} \ldots \mathrm{d}z_N, \qquad n < N, \tag{5.34}$$

thus constructing a projection (Definition 25)

$$\pi : T^*M^N \to T^*M^n \tag{5.35}$$

such that the higher-dimensional phase space T^*M^N appears as a fiber (Definition 40) over the reduced phase space T^*M^n, since it is certain that

$$T^*M^N = T^*M^n \times T^*M^{(N-n)}, \tag{5.36}$$

An equation governing the evolution of this reduced density function can be obtained by integrating Equation 5.33b, resulting in the expression

$$\frac{\partial \rho_n}{\partial t} + \{\rho_n, H_n\} = (N - n) \int \dot{p_n} \rho_{n+1} \mathrm{d}z_{n+1} \tag{5.37}$$

where the left hand side containing the Hamiltonian H_n depends on the considered n particles, while the right hand side describes the interactions with all the remaining $N - n$ particles by coupling to the next higher distribution function. Therefore this equation is not closed, but gives rise to a hierarchy of equations, the BBGKY, named after the individuals who have independently derived this system, Bogoliubov [87], Born [88], Green [89], Kirkwood [90] and Yvon [91].

Using this formulation, the restriction to just one particle yields an expression for a single particle Hamiltonian, which results in phase space trajectories with deviations from these trajectories attributable to collisions.

$$\frac{\partial \rho_1}{\partial t} + \{\rho_1, H_1\} = Q(\rho_1) \tag{5.38}$$

This also shows the price which has to be paid for the reduction of degrees of freedom. The observed particle will no longer follow a simple curve through phase space and is disturbed due to the scattering term which appears on the right hand side as is illustrated in Figure 5.3. While

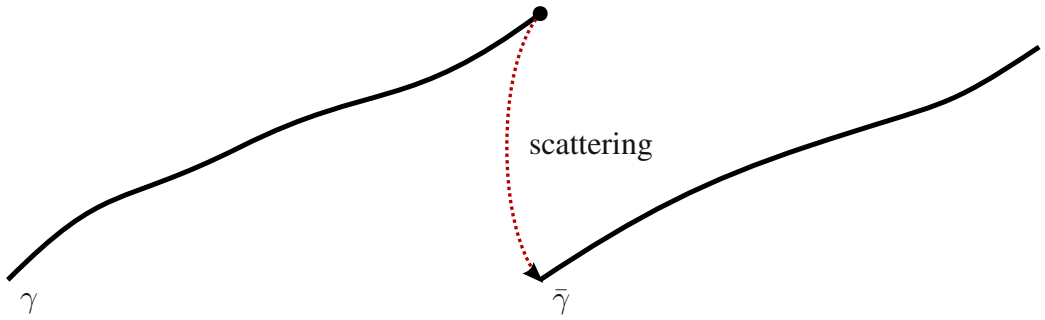


Figure 5.3: The trajectory is interrupted due to scattering.

the derivation naturally describes the collison term with other particles, it can also model the interaction of the particle tracked by Boltzmann's equation with the envrionment, thus allowing for an interpretation as probability of a particle evolving to a given point.

While Boltzmann's equation appears as a simplification here, it is far from easy to obtain solutions to this deceivingly simple equation. Therefore several different techniques have been developed to at least calculate estimates in different contexts and with various levels of accuracy.

5.5 Macroscopic Quantities

While the description using Boltzmann's equation is already a reduction with regard to the degrees of freedom, it is still a statistical description on a microscopic level. It is important to be able to link this modelling on a microscopic level to macroscopic quantities, which are accessible to measurements. The statistical nature already offers a means in the form of expectation values (Definition 101), which also allows to compact the statistics of the micro scale to simple quantities of the macro scale. Thus, a macroscopic quantity X on the manifold $\mathfrak{M}$ (Definition 35) can be obtained from the microscopic description using ρ on the phase space T^*M by

$$\langle X \rangle = \int X \rho \, \mathrm{d}\mathbf{p}. \tag{5.39}$$

This resembles a projection of the form

$$\pi : \rho \in \mathbb{F}(\mathfrak{T}^*\mathfrak{M}) \to \mathcal{T}_s^r(\mathfrak{M}), \tag{5.40}$$

which again reveals a fiber bundle (Definition 40) structure on the base manifold $\mathfrak{M}$ (Definition 35), indicating the powerful versatility of the fiber bundle abstraction.

An important feature of this prescription is that this procedure can not only be applied to already obtained solutions of ρ as a post processing step, but can also be applied in order to derive macroscopic equations from Boltzmann's equation. Several differing procedures are available to provide such a mapping resulting in differing macroscopic equations, ranging from convection-diffusion, also called drift-diffusion, to the Euler equations or the Navier-Stokes equations. This mapping may either be viewed as a defining derivation or as a link between two different descriptions, which have been derived independently or have even been postulated by empiric means.

A basic method uses a collection of weight functions $\chi(\mathbf{p})$ to perform the contraction by integrating (Definition 94) over sub-manifold slices formed by the momentum coordinates $\mathbf{p}$. It thus bears a certain resemblance to the procedure involved in the derivation of the BBGKY, except that the integration is carried out only with respect to the momentum coordinate using the distribution as measure, while preserving the spatial components.

5.6 Boltzmann's Equation in Integral Form

Starting with Boltzmann's equation in differential form

$$\frac{\mathrm{d}\rho}{\mathrm{d}t} = \frac{\partial\rho}{\partial t} + \{\rho, H\} = Q(\rho), \tag{5.41}$$

it is seen that the evolution of the system is connected to the Hamiltonian defined on the phase space. Test particles evolve on trajectories (Definition 45), which comprise the Hamiltonian flow corresponding to the Hamiltonian vector field (Definition 62). Thus trajectories are essential to determine the evolution connected to Boltzmann's equation.

In addition to the trajectories, which are defined by the geometry of the phase space via the Hamiltonian H, it is also required to give a specific description of the collision operator on the right hand side. In the field of semiconductor simulations it is common to describe the collision operator $Q(\rho)$ in the form

$$Q(\rho) = \int S(\mathbf{p}', \mathbf{p}, \mathbf{q})\rho(\mathbf{p}', \mathbf{q}, t) - S(\mathbf{p}, \mathbf{p}', \mathbf{q})\rho(\mathbf{p}, \mathbf{q}, t)\mathrm{d}\mathbf{p}'. \tag{5.42}$$

Using the concept of trajectories on which particles travel, the two components of this integral expression can be interpreted to have physical significance. The first component of the given integral describes the loss of particles on a given trajectory due to collision/scattering events to other trajectories, thus called out-scattering. The second part models the gains to this particular trajectory due to the collision/scattering mechanisms, aptly called in-scattering. This second term can be simplified by setting

$$\int S(\mathbf{p}, \mathbf{p}', \mathbf{q})\mathrm{d}\mathbf{p}' = \lambda(\mathbf{p}, \mathbf{q}). \tag{5.43}$$

Thus Equation 5.42 becomes

$$Q(\mathbf{p},\mathbf{q},t) = \int S(\mathbf{p}',\mathbf{p},\mathbf{q})\rho(\mathbf{p}',\mathbf{q},t)\mathrm{d}\mathbf{p}' - \lambda(\mathbf{p},\mathbf{q})\rho(\mathbf{p},\mathbf{q},t). \tag{5.44}$$

As already indicated the evolution is inherently linked to trajectories γ in phase space which provide a mapping of a curve parameter to the phase space points, represented by the pair $\mathbf{p}$, $\mathbf{q}$.

$$\gamma : \mathbb{R} \to T^*M \tag{5.45a}$$
$$\gamma(r) \to (\mathbf{p},\mathbf{q}) \tag{5.45b}$$

As such the expression for $\lambda(\mathbf{p},\mathbf{q})$ can also be rendered as $\lambda(r)$ as a short form for

$$\lambda(r) = \lambda(\gamma(r)) = \lambda(\mathbf{p},\mathbf{q}), \qquad \gamma(r) = (\mathbf{p},\mathbf{q})\,. \tag{5.46}$$

$\gamma(r)$ represents the trajectory travelled before a scattering event, while the particle continues to travel on a trajectory $\bar{\gamma}(r)$ as indicated in Figure 5.3 afer a scattering event. With the two curves requiring to match at the parameter r_c corresponding to a collision/scattering event in the following manner

$$\gamma(r_c) = (\mathbf{p},\mathbf{q}), \tag{5.47a}$$
$$\bar{\gamma}(r_c) = (\mathbf{p}',\mathbf{q}'), \tag{5.47b}$$
$$\mathbf{q} = \mathbf{q}'. \tag{5.47c}$$

Utilizing these settings, Equation 5.42 can be rendered as

$$\frac{\mathrm{d}\rho(\gamma(t))}{\mathrm{d}t} + \lambda(\gamma(t))\rho(\gamma(t)) = \int S(\mathbf{p}',\gamma(t))\rho(\mathbf{p}',\mathbf{q},t)\mathrm{d}\mathbf{p}' = \tag{5.48a}$$
$$\frac{\mathrm{d}\rho(\gamma(t))}{\mathrm{d}t} + \lambda(\gamma(t))\rho(\gamma(t)) = \int S(\bar{\gamma}(t),\gamma(t))\rho(\bar{\gamma}(t))\mathrm{d}\mathbf{p}'. \tag{5.48b}$$

In the following the dependence of the curves γ on the parameter shall be suppressed, where it facilitates readability without adversely affecting clarity.

Here it is desirable to find an integrating factor, so that the left hand side resembles a total derivative. The integrating factor in this case is

$$e^{-\int_t^\tau \lambda(\xi)\mathrm{d}\xi}, \tag{5.49}$$

as can easily be verified by simple differentiation.

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(e^{-\int_t^\tau \lambda(\xi)\mathrm{d}\xi}\rho(\gamma,t)\right) = e^{-\int_t^\tau \lambda(\xi)\mathrm{d}\xi}\int S(\bar{\gamma},\gamma)\rho(\bar{\gamma})\mathrm{d}\mathbf{p}' \tag{5.50}$$

Applying definite integration the expression for the solution of Boltzmann's equation becomes

$$\rho(\mathbf{p},\mathbf{q},t) = \int_a^t e^{-\int_\tau^t \lambda(\xi)\mathrm{d}\xi}\left(\int S(\bar{\gamma},\gamma(\tau))\rho(\bar{\gamma},\tau)\mathrm{d}\mathbf{p}'\right)\mathrm{d}\tau + e^{-\int_a^t \lambda(\xi)\mathrm{d}\xi}\rho(a) \tag{5.51a}$$
$$= \int_a^t \int e^{-\int_\tau^t \lambda(\xi)\mathrm{d}\xi} S(\bar{\gamma},\gamma(\tau))\rho(\bar{\gamma},\tau)\mathrm{d}\mathbf{p}'\mathrm{d}\tau + e^{-\int_a^t \lambda(\xi)\mathrm{d}\xi}\rho(a). \tag{5.51b}$$

The last term deserves special consideration. It is easy to argue that it represents the initial conditions of the sought function from which the system begins to evolve. It however also accommodates boundary conditions as becomes apparent, when considering that the points of the trajectories corresponding to the parameter a not necessarily lie within the domain of interest. In this case the trajectory within the domain uses the value at the boundary instead as illustrated in Figure 5.4. Thus the term accommodates both, initial as well as boundary conditions for this integral equation [92].

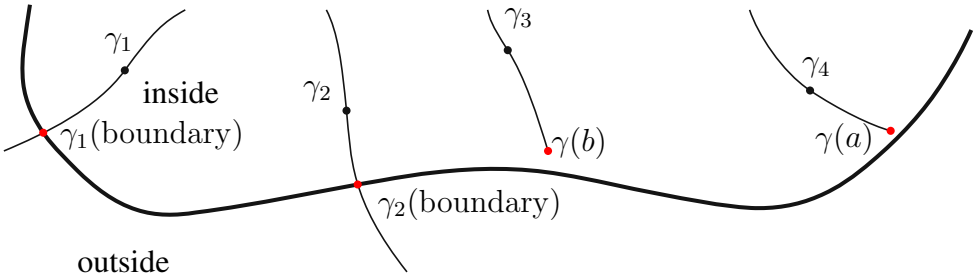


Figure 5.4: Trajectories accommodate initial as well as boundary conditions.

5.7 The Quantum World

While mechanics in its classical form describes the world with determinism, even if predictability is undermined, when turning to statistical methods, this is drastically changed in a quantum mechanical setting. Already well established and tangible concepts such as particles are recast in the setting of quantum mechanics using wave functions. This makes effects describable, which are completely unavailable to the classical setting, without changing the Newtonian structure of the universe with its absolute time.

Quantum mechanics is often introduced in a detached manner with little to no connection to the classical settings supporting utterly alien effects which even border on the bizarre. It is therefore interesting to note that the governing structures so painstakingly uncovered by scientists over the course of time for the classical case are still present in the quantum setting. This comes as no surprise in light that the macroscopic world of every day life follows classical rules, wherefore the descriptions of quantum systems should always yield classical behaviour, when taken to the appropriate limit.

The setting of the quantum world is chosen to be a complex Hilbert space from which functions are chosen to represent the state of a system. The structure of this complex Hilbert space together with a link to the macroscopic world is sufficient to extract the governing equation of the system.

The wave functions Ψ, which are drawn from the complex Hilbert space $\mathcal{H}$ are acted upon mappings, which are often called operators in this context, to describe transitions of states. While the Ψ are capable of encoding the state of a quantum system, they elude observations completely. At first this seems to contradict the requirement that scientific theories must be falsifiable [93] and thus accessible to some form of experiments and measurement. However, observables can be constructed from Ψ in a systematic fashion, which are again accessible to measurements and thus to testing of the theory. Without touching upon the, for many researchers sensitive, topic of the many different interpretations of quantum mechanics [94][95][96][97] the wave functions $\Psi \in \mathcal{H}$ need to meet the following property

$$\langle \Psi | \Psi \rangle = 1 \tag{5.52}$$

as to allow the interpretation as a probability.

The prescription of constructing observables yields expectation values (Definition 101) of previously classically determined entities such as location or momentum. It is thus a reasonable demand that this mapping shall produce entities with their expected structures, e.g., appropriate tensors (Definition 54).

$$A : \mathcal{H} \to T^r_s \tag{5.53}$$

In this fashion a correspondence of operators acting on Ψ and classical observables is established

$$\langle A \rangle = \langle A(\Psi) \rangle = \langle \Psi | \hat{A} | \Psi \rangle, \tag{5.54}$$

where $\langle A \rangle$ is the classical expectation value (Definition 101) and $\hat{A}$ is a operator on elements of the Hilbert space. This defines the observable operators on Ψ.

As in the classical case, a differential form (Definition 59) is derivable with a function A on the Hilbert space $\mathcal{H}$. At a point $\Psi \in \mathcal{H}$ for a tangent vector Φ (Definition 48) using an arbitrary

parameter λ this gives

$$\langle \mathrm{d}A(\Psi)(\Phi)\rangle = \frac{\mathrm{d}}{\mathrm{d}\lambda}\Big|_{\lambda=0}\langle \Psi+\lambda\Phi, \hat{A}(\Psi+\lambda\Phi)\rangle \tag{5.55a}$$

$$= \frac{\mathrm{d}}{\mathrm{d}\lambda}\Big|_{\lambda=0}\Big(\langle \Psi, \hat{A}\Psi\rangle + \langle \Psi, \hat{A}\lambda\Phi\rangle + \langle \lambda\Phi, \hat{A}\Psi\rangle + \langle \lambda\Phi, \hat{A}\lambda\Phi\rangle\Big) \tag{5.55b}$$

$$= \langle \Psi, \hat{A}\Phi\rangle + \langle \Phi, \hat{A}\Psi\rangle. \tag{5.55c}$$

In case $\hat{A}$ is self adjoint with respect to the scalar product, this can be reshuffled to

$$\langle \hat{A}\Psi, \Phi\rangle + \langle \Phi, \hat{A}\Psi\rangle. \tag{5.56}$$

Since a complex Hilbert space $\mathcal{H}$ is also a Kähler space (Definition 85), the Hermitian inner product (Definition 77) of $\mathcal{H}$ may be rendered as a composition of a Riemannian component g (Definition 76) and a symplectic component ω (Definition 81).

$$h(\mathbf{x},\mathbf{y}) = \frac{1}{2\hbar}g(\mathbf{x},\mathbf{y}) + \frac{1}{2\hbar}\omega(\mathbf{x},\mathbf{y}) \tag{5.57}$$

Thus the preceding expression may be modified to

$$\frac{1}{2\hbar}g(\hat{A}\Psi,\Phi) + \frac{1}{2\hbar}g(\Phi,\hat{A}\Psi) = \frac{1}{\hbar}g(\hat{A}\Psi,\Phi), \tag{5.58}$$

from which, due to the relation between symplectic and Riemannian part as required by the Kähler structure, it follows further

$$\frac{1}{\hbar}g(\hat{A}\Psi,\Phi) = -\frac{i}{\hbar}\omega(\hat{A}\Psi,\Phi). \tag{5.59}$$

Which motivates, in allusion to the Hamiltonian vector fields (Definition 62) of the classical case, the association of a Hamiltonian vector field $\mathbf{v}_{\hat{A}\Psi}$ with an operator $\hat{A}$ by

$$\mathbf{v}_{\hat{A}}(\Psi) = -\frac{i}{\hbar}\hat{A}\Psi, \tag{5.60}$$

which finally enables to conclude using the interior product (Definition 72)

$$\langle \mathrm{d}A(\Psi)(\Phi)\rangle = \omega(\mathbf{v}_{\hat{A}\Psi},\Phi) = (i_{\mathbf{v}_{\hat{A}\Psi}}\omega)(\Phi). \tag{5.61}$$

As can be observed, the classical expectation value can be linked to the symplectic structure inherent to the complex Hilbert space $\mathcal{H}$.

Thus the structures governing the evolution of the classical case, in the form of Hamiltonian vector fields, are also recognizable in a quantum system. While all of the Hilbert space is endowed with the symplectic structure, the normalization requirement confines the selection of states to the unit hyper-sphere. However, this restriction is not sufficient to remove all ambiguity, since it still remains in the form of uni-modular factors

$$\Psi' = c\Psi, \qquad |c| = 1. \tag{5.62}$$

This remaining ambiguity is removed by further restricting the Hilbert space to a projective space $\mathcal{HP}$. This projective space may now be regarded as a phase space connected to a physical system.

In case of the Hamiltonian operator, the prescription 5.60 of assigning vector fields to operators yields exactly Schrödinger's equation as can be seen by

$$\mathbf{v}_{\hat{H}}\Psi = -\frac{i}{\hbar}\hat{H}\Psi \tag{5.63a}$$

$$i\hbar\dot{\Psi} = \hat{H}\Psi, \tag{5.63b}$$

which is the main governing equation of the quantum system. Thus the Hamiltonian vector field defined in this fashion is indeed responsible for the evolution of the quantum system in the Schrödinger picture of quantum mechanics, where operators remain constant and the states evolve.

While a function $\Psi \in \mathcal{PH}$ captures a state completely, descriptions usually employ a representation using either position $\mathbf{r}$ or momentum $\mathbf{p}$ representations, which may be obtained by (utilizing Bra-Ket notation introduced in Section 4.6.2)

$$\Psi(\mathbf{r}, t) = \langle \mathbf{r}|\Psi(t)\rangle \tag{5.64}$$

and

$$\Phi(\mathbf{p}, t) = \langle \mathbf{p}|\Phi(t)\rangle. \tag{5.65}$$

The change between these two representations is possible using a Fourier transform

$$\Phi(\mathbf{p}, t) = (2\pi\hbar)^{-\frac{n}{2}} \int \Psi(\mathbf{r}, t)\, e^{-\frac{i}{\hbar}\langle \mathbf{p},\mathbf{r}\rangle} \mathrm{d}\mathbf{r} \tag{5.66a}$$

$$\Psi(\mathbf{r}, t) = (2\pi\hbar)^{-\frac{n}{2}} \int \Phi(\mathbf{p}, t)\, e^{\frac{i}{\hbar}\langle \mathbf{p},\mathbf{r}\rangle} \mathrm{d}\mathbf{p}, \tag{5.66b}$$

where n is the spatial dimension of the problem under investigation.

These descriptions account for half the degrees of freedom implicitly, while exposing only the other half. It is, however, possible to construct a representation, which explicitly recovers all degrees of freedom. The density operator and the density matrix are such representations. The density operator is given by

$$\hat{\rho} = |\Psi\rangle\langle\Psi|. \tag{5.67}$$

By evaluating this operator at two different positions in the form

$$\rho(\mathbf{r}_1, \mathbf{r}_2) = \langle \mathbf{r}_1|\hat{\rho}|\mathbf{r}_2\rangle = \langle \mathbf{r}_1|\Psi\rangle\langle\Psi|\mathbf{r}_2\rangle = \int \Psi^*(\mathbf{r}_2)\Psi(\mathbf{r}_1) \tag{5.68}$$

the density matrix $\rho(\mathbf{r}_1, \mathbf{r}_2)$ is obtained, which may be seen as a measure of correlation between two positions $\mathbf{r}_1$ and $\mathbf{r}_2$ and retains all state information encoded in Ψ, as may be reinforced when reformulating Equation (5.54) as

$$\langle A\rangle = \langle\Psi|\hat{A}|\Psi\rangle = \iint \langle\Psi|\mathbf{r}_1\rangle\langle\mathbf{r}_1|\hat{A}|\mathbf{r}_2\rangle\langle\mathbf{r}_2|\Psi\rangle \mathrm{d}\mathbf{r}_1\mathrm{d}\mathbf{r}_2 \tag{5.69a}$$

$$= \int \langle\mathbf{r}_2|\Psi\rangle \left(\int \langle\Psi|\mathbf{r}_1\rangle\langle\mathbf{r}_1|\hat{A}|\mathbf{r}_2\rangle \mathrm{d}\mathbf{r}_1\right) \mathrm{d}\mathbf{r}_2 \tag{5.69b}$$

$$= \int \langle\mathbf{r}_2|\Psi\rangle\langle\Psi|\hat{A}|\mathbf{r}_2\rangle \mathrm{d}\mathbf{r}_2 \tag{5.69c}$$

$$= \int \langle\mathbf{r}_2|\hat{\rho}\hat{A}|\mathbf{r}_2\rangle \mathrm{d}\mathbf{r}_2. \tag{5.69d}$$

Utilizing such a representation, which provides information about the correlation of all of the points in the problem domain, it is possible to also construct a representation which uses the position $\mathbf{r}$ and the momentum $\mathbf{p}$, or wave vector $\mathbf{k}$ associated by

$$\mathbf{p} = \hbar\mathbf{k}, \tag{5.70}$$

to describe the quantum system and thus mapping it to a phase space similar to the classical settings. This is provided by the so called Wigner-Weyl transform [98][99], where the Wigner transform of an operator is given by

$$A(\mathbf{q},\mathbf{p}) = \int_{-\infty}^{\infty} \langle \mathbf{q} - \frac{\mathbf{r}}{2} | \hat{A} | \mathbf{q} + \frac{\mathbf{r}}{2} \rangle \mathrm{e}^{-\frac{i}{\hbar}\langle \mathbf{p},\mathbf{r}\rangle} \mathrm{d}\mathbf{r} \tag{5.71}$$

In the case of the density operator, a quasi-probability distribution function f is obtained,

$$f(\mathbf{q},\mathbf{p}) = \int_{-\infty}^{\infty} \langle \mathbf{q} - \frac{\mathbf{r}}{2} | \hat{\rho} | \mathbf{q} + \frac{\mathbf{r}}{2} \rangle \mathrm{e}^{-\frac{i}{\hbar}\langle \mathbf{p},\mathbf{r}\rangle} \mathrm{d}\mathbf{r}, \tag{5.72}$$

which mirrors the distribution function of the classical case but, is no longer positively definite and thus can not be strictly interpreted as a probability density function, since it is no longer a measure but a signed measure (Definition 91). It is, however, possible to compute expectation values (Definition 101) in the usual manner using

$$\langle X \rangle = \int X f \mathrm{d}\mathbf{q}\mathrm{d}\mathbf{p} \tag{5.73}$$

for a quantity X.

Since the transform linking the representations in terms of ρ and f is linear, the evolution of the system may be expressed in either of them equivalently. The structures inherent to the space of quantum mechanics also allow to formulate the evolution of a physical quantity, such as for the $\hat{\rho}$ as

$$\dot{\hat{\rho}} = \left[\hat{\rho}, \hat{H}\right] + \frac{\partial}{\partial t}\hat{\rho}, \tag{5.74}$$

which takes on a similar shape in the quantum phase space using the quasi-probability distribution function. It has to be noted, however, that the Lie bracket (Definition 69) applied in the context of operators in the Hilbert space has a slightly warped shape in the quantum phase space, due to the transformation given in Equation 5.71. The deformation takes on the form [100]

$$\{f,g\}_M = \frac{2}{\hbar}\sin\frac{\hbar}{2}\left(\frac{\partial}{\partial\mathbf{q}}\frac{\partial}{\partial\mathbf{p}} - \frac{\partial}{\partial\mathbf{p}}\frac{\partial}{\partial\mathbf{q}}\right) f(\mathbf{q},\mathbf{p}) g(\mathbf{q},\mathbf{p}). \tag{5.75}$$

Using this deformed bracket, the evolution in quantum phase space can be described as

$$\dot{f} = \{f,H\}_M + \frac{\partial}{\partial t} f, \tag{5.76}$$

which mimics the classical case as expressed in Equation 5.31. It also follows that the classical Poisson bracket and hence classical evolution can be recovered from the prescription given in Equation 5.75 in the limit $\hbar \to 0$, thus this construct meets a basic property demanded from a quantum description.

Part III

Methods

Chapter 6

Components

> Everything makes sense a bit at a time. But when you try to think of it all at once, it comes out wrong.
>
> Terry Pratchet

Sifting through scores of old, badly maintained code of tools for scientific computing showed several recurring themes and topics from the different fields of mathematics presented in Chapter 4. However, only very few of the implementations seemed to have any regard for reusability. At times there did not seem to be a discernible theme of purpose at all. Turning to what little documentation was available, it was found to be akin to the black box of a crashed air plane, horribly mangled and beyond use. Since the greater use of the code was still available, it was possible to reverse engineer the intent of the local code, but at a much greater expense than necessary. The bad state of the source code and the great effort required to divine its intended purpose cannot be attributed to a single factor. The implementations were designed with the imperative as well as object-oriented paradigms in mind. And while they were realized with considerable forethought as well as skill in programming, they also passed through the hands of several different programmers of very diverse levels of awareness of the abstractions used in the initial writing as well as skill. Thus the initially object-oriented components, with their hierarchy and data encapsulation were soon subverted in order to make access easier for purely imperative constructs. This trend seems to have been accelerated by bursts of sloppiness due to haste such as resulting due to approaching deadlines, leaving obviously deeply flawed constructions behind. This is of course nothing, which anyone would want to document, if it is done consciously, while the interesting implications do not come to mind, if it is done unconsciously. The resulting monstrous heap of code is an unwieldy nightmare to maintain and extend. This situation was confounded by a lack of unit testing infrastructure which made diagnostics virtually impossible on a manageable scale.

Naturally, the prevention of such a dreadful deterioration of new projects in the same fashion was among the main questions resulting from this encounter. An answer to this may be found in the nature of the object-oriented paradigm and its facilities for code reuse, which do not easily allow for non-intrusive extensions after the initial design. As the initial design cannot encompass all subsequent use cases and scenarios, such extensions seem unavoidable. Thus intrusive changes are unavoidable, which may, however, change the behaviour of the entire object, if not performed with a modicum of diligence. This scenario also makes it much more appealing to

simply attempt to circumvent the now bothersome mechanisms of clean object-oriented designs, instead to improving it; especially, when the initial design may no longer be clear.

Therefore, key components, whose concepts have been found strewn throughout the legacy code bases, have been identified and their mathematical descriptions and definitions isolated, as shown in Chapter 4. Using these mathematical concepts options for implementation have been sought and explored relying on the generic programming paradigm. The rational behind this is that mathematical concepts are quite robust to the evolution of methodology as long as they are well defined. Furthermore, the generic implementations offer much less incentive to be subverted to be extended, since extensions should be possible unintrusively. Thus it should be much more common for code to be simply phased out, as it is replaced by more adequate implementations. Where the result of the extraction of basic mathematical concepts found useful for scientific computing has been described in Chapter 4, their mapping to the realm of digital computers is described in the the following. As the compilation of definitions was a theoretical contribution, the description of example realizations here is of a practical nature. It puts well known data structures in their contexts and supplies additions, where needed.

Sets are an essential building block used in computing, since the memory used to store values in digital computing constitutes a set. Each of the memory cells within the whole are distinguishable from one another and form a discrete topological space (Definition 29). The discrete nature of a digital computer's memory not only forces the topology to be Hausdorff (Definition 33), but also results in the fact that every mapping (Definition 20) among memory cells is continuous (Definition 30). The distinct existence of memory cells would be incomplete, if they could not each store values. A short examination reveals that this structure matches the definition of a fiber bundle (Definition 40) with the memory as the base space and their contents as the respective fibers. The management and partitioning (Definition 2) of a system's memory is commonly a central task of an operating system's kernel. It is facilitated by the use of addresses, which identify all of the memory locations, which is easily recognized as the use of an atlas (Definition 36), effectively identifying the structure as a manifold (Definition 35). The atlas may be comprised by a single chart, but may also be more complicated as in the case of paged virtual memory [101] for instance. While it is common for this structure to be ignored, or go unexplored as unnecessarily complex, it still is present even in these basic primitive layers of machines. Any higher abstractions dealing with topological spaces or manifolds are essentially mappings from the intended structures to this low level manifold. Thus special care of encoding their structure in a fashion suitable to a representation in the lowest levels of the used machines has to be taken. It furthermore illustrates the worth of having definitions available, with which to describe encountered structures.

Additional components were added to an already existing generic scientific simulation environment (GSSE) [57][28][102] to enable an efficient realization of topological space concepts given in Chapter 4. The environment provides a variety of compile time and run time interfaces to arbitrary-dimensional topological spaces, e.g., sets, graphs, implicit and explicit grid and mesh structures.

6.1 Sets

On a higher level the notion of sets remains interesting to developers. Various classes of containers model topological spaces, and thus sets, with different requirements and applications [57]. Since each memory location is distinct, low level containers may include several items with the same value without it being considered as a duplicate, which contradict the notion of a set. If it is required to obtain a set of distinct values, an addition relation (Definition 4) for comparison of the stored values is required to realize such a data structure, since without it, it is impossible to judge two items for equality, thus indicating duplicates. Set containers are available for both, run time as well as compile time, as is illustrated using the C++ `std::set`, `mpl::set`, and `fusion::set` representatives

Example set data structures.

```
std::set<Key, Compare, Allocator>
mpl::set<t1, t2, ..., tn>
fusion::set<T1, T2, ..., Tn>
```

The run time version taken directly from C++'s STL [38] requires not only a data type, but also the specification of a comparison relation which is used in this implementation to impose an ordering of the inserted elements and uses this mechanism to ensure uniqueness. It also allows to specialize the memory layout via an allocator. The STL further provides several set operations in the form of algorithms such as `includes`, `set_union`, `set_intersection` and `set_difference`. In contrast to the run time implementation, the featured compile time facility provided by the MPL [58] needs neither an additional relation, since types are inherently required to be distinguishable, nor an allocator, since no memory is required. Finally, a hybrid approach combining a compile time set with run time values is also available in the form of `fusion::set` available from the Boost Fusion [103] library.

To enable a consistent compile time and run time set interface, the GSSE offers 0D cell complex data types which can be used in multiple application areas, e.g., sparse and dense polynomial containers [104] as given in the next Chapter 6.2. The following code snippet demonstrates an application of the internal data layout of a set and implements a data structure equivalent to a `std::vector` and can be specified as a dense memory arrangement of any data type element, e.g., a `double`. The set properties are then handled by the C++ standard library, where a free function `insert()` controls the set property.

Example topology data structures.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<0> >
        , gsse::pair<tag_storage,   double >
    > complex_0D_t;
```

Orthogonal properties related to a set definition, in contrast to C++'s standard library, can be clearly observed due to the separation of topological structure, as displayed by `tag_dimension`, and data storage `tag_storage`. The availability of such basic generic data structures modelling the concept of a set on a higher semantic level allows to considerably simplify the development of more complex algorithms which require to act on sets.

6.2 Data Types

While it may be possible to derive a certain amount of pleasure from the mere storage of items in various memory locations, it falls far short from the intents and purposes of digital computers, which rather boast the capability of performing operations on the data items at a speed far surpassing the capabilities of average humans. Algebraic operations as outlined in Section 4.2 are among the most important operations to be carried out by a machine. In its most crude form digital computers represent data items as a binary pattern within their memory. This pattern is in essence subject to interpretation to determine not only the value but also the operations available. Both of these two notions are intimately related and are therefore formally combined in the concept of data types. Usually several simple data types are inherently defined as part of programming languages, such as for integers and rational numbers. Higher level programming languages, especially those considered to support the object-oriented paradigm, allow the definition of custom data types and the specification of available operations for these newly defined types. Several programming languages furthermore allow to overload operators for these custom types, thus allowing the expression of semantics in a fashion comparable to the built in types.

However, the limited, finite nature of the used machines results in restrictions to the data types which can be implemented. It especially restricts the modelling of mathematical structures, such as the real numbers $\mathbb{R}$. Thus, while implemented data types may approximate these algebraic structures, they cannot mimic these structures in their entirety. As a consequence it is of considerable importance to be aware of the limitations of the used data types and the errors incurred due to the approximation.

An example of the separation of the algebraic properties from basic numerical data types can be given using the notion of polynomials (Definition 19). The definitions specifies that the coefficients should be drawn from a ring (Definition 13) but does not further specify the nature of the variable x. Similarly the implementation can focus on separating the type of the variable x from the type of the coefficients, since the the operators for addition (subtraction) and multiplication can be implemented completely irrespective of x; it is only necessary to derive formal powers. Any operations regarding the polynomials need to deal with an important decision with respect to the representation and storage of the coefficients on a digital computer. A very simple manner to store coefficients of polynomials is to store them in an array, or a similarly indexable container. Such that a polynomial of the form

$$p(x) = x^5 + 7x^2 + 2x^1 + 1.3x^0 \tag{6.1}$$

can be associated with an expression of the form

Simple storage of a polynomial's coefficients.

```
std::vector<double> poly;
poly[0] = 1.3;
poly[1] = 2;
poly[2] = 7;
poly[3] = 0;
poly[4] = 0;
poly[5] = 1;
```

This easy method of storage, which is intuitively similar to the ubiquitous positional notation of numbers, unfortunately requires the storage of all the coefficients between the minimal and maximal power within the polynomial, since the power the coefficient is associated to is stored implicitly by the position within the container. Such a method of storage shall therefore be referred to as implicit. The efficiency of implicit storage depends on the intended field of application, which dictates the number of 0 coefficients, which still need to be stored. On the other hand memory can be allocated in as a contiguous block, wherefore it is also known as dense.

The limitations due to inefficiency of implicit representation increases as the number of zero coefficients increases. While, in a mathematical setting it is easy to deal with an infinite number of coefficients, especially, if only a finite number is non-zero. The polynomial of Equation 6.1 can also be represented in the following, completely equivalent, manner.

$$c_i = 0, \forall i \in \mathbb{N}, i \neq 0, 1, 2, 5 \tag{6.2a}$$

$$c_0 = 1.3, c_1 = 2, c_2 = 7, c_5 = 1 \tag{6.2b}$$

The above notation, however, provides a model for a different approach to the storage of coefficients, in the following referred to as explicit or sparse storage. The name is due to the fact that the power to which a coefficient belongs to needs to be specified explicitly. Using an STL container this can be accomplished using

Sparse/Explicit storage of coefficients.

```
std::map<long, double> poly;
poly[0] = 1.3;
poly[1] = 2;
poly[2] = 7;
poly[5] = 1;
```

where in contrast the dense case all unspecified values simply do not exist, which makes it easier to deal with polynomials where many of the coefficients are zero, without incurring memory overhead, at the cost of requiring more intricate memory access mechanisms.

By using the GSSE, the different STL data structure definitions are unified into a common data specification compile time program. The next code snippet presents the equivalent dense and sparse container definitions.

Example topology data structures.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<0> >
        , gsse::pair<tag_storage,   double >
    > container_dense_t;

typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<0> >
        , gsse::pair<tag_storage,   double >
        , gsse::pair<tag_index,     long>
    > container_sparse_t;
```

The examples given so far have made use of data types from the STL and GSSE. While these containers are quite capable of storing the coefficients, they are not equipped with operators to deal with the operations of addition and multiplication of polynomials associated with the stored values. This can be remedied by directly implementing operators for the used containers, e.g., the `std::vector` or the `std::map`. While this is definitely a viable option to model the algebraic structure of polynomials, it is not only completely lacking elegance, but also incurs serious problems. Such an approach has the profound side effect that now every container of the chosen type could be treated as a polynomial with respect to addition and multiplication, even if this is by no means intended. The situation then escalates, when different algebraic entities also utilize the same type of container, but with different algebraic structures which should, however, be represented using syntactically and formally identical operators, such as multiplication for vectors (Definition 16).

It is therefore prudent to implement custom data types, for which the specific operations are defined, to avoid these grave repercussions. The custom data type may contain one of the previously described containers for dense or sparse storage of coefficients, however, as an implementation detail, which is encapsulated and thus can be easily exchanged and adapted as the situation requires without adversely affecting the interface. Proceeding in this manner is consistent with the object-oriented paradigm as described in Section 3.1.2.

A generic library for the handling of polynomials using run time as well as compile time mechanisms [104] is available, which defines not only the basic algebraic structure of polynomials, but also provides facilities for integration and differentiation polynomials in a flexible and efficient manner. As a compile time example the derivative of a second-degree polynomial is calculated:

$$\frac{\mathrm{d}(10x^2 + 4.5x + 3)}{\mathrm{d}x} = 20x + 4.5$$

The type list represents the type of each coefficient starting from the zero to the second degree coefficient.

```
typedef gsse::vector_ct <double,
                         double,
                         int> coeffs;

poly    p(x, coeffs(3.0, 4.5, 10));
diffed d = diff(p, x);
```

The combination of providing custom data types along with the ability to overload operators enable the realization of specialized domain specific embedded languages (DSELs), which can greatly increase the level of abstraction as well as ease of notation, but does not impede the run time performance. By evaluating the resulting assembler code of this compile time program, the binary only contains the final coefficients `4.5, 20`.

6.3 Topology

From among the different qualifications resulting from the combination of a set and a relation, the concept of a partially ordered set (poset) is of special importance for the implementation of concepts relating to topology (Definition 28). Where the topology of the underlying structure cannot be changed, it can be employed to model different topologies on top of it. The topological spaces required for manifolds (Definition 35) as well as fiber bundles (Definition 40) are of particular interest in this context. They can be represented using a collection of connected cells forming a CW complex (Definition 34).

As topology is constructed on the foundations of set theory, it is only natural that the specification of topological structures retains a semblance to the structures of sets. However, since topology requires as well as provides additional information, the interface must accommodate this fact. Here the most prominent requirements are dimension and global space properties.

The GSSE code snippet in Section 6.1 already presents the definition of a homogeneous interface to a set. Here, topological spaces are now introduced by using the same GSSE data structure specification as used before. So far different data structures and higher abstraction always required to introduce new types. These types mostly demanded further code adaptation, e.g., different data containers and hence iteration modifications. By using GSSE specifications, all data structure and topological properties remain consistent and no traversal modification is necessary. The important difference here is that data structures are not restricted to 0D cell types, but can instantiate arbitrary-dimensional CW complex spaces, at compile time and run time.

Example explicit 2D simplex mesh structure.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<2> >
        , gsse::pair<tag_cell,      gsse::cell_simplex >
        , gsse::pair<tag_complex,   gsse::complex_explicit >
    > cw_complex_t;
```

The given tags instantiate a 2D simplex cell complex, where the internal cell structure (cell topology) is calculated at compile time and available at run time without performance losses. Arbitrary explicit and implicit cell calculation algorithms can be implemented and used.

Also the implicit and explicit nature of polynomials, given in Section 6.2, can be found in this data type definitions, where computer storage capacities are formed by implicit and explicit declarations.

Example implicit 4D hypercube grid structure.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<4> >
        , gsse::pair<tag_cell,      gsse::cell_hcube >
        , gsse::pair<tag_complex,   gsse::complex_implicit >
    > cw_complex_t;
```

Implicit higher-dimensional topological structures can be identified by dense grids, where no explicit cell indices are stored in memory. Instead all necessary grid topology is calculated on demand.

As introduced in Section 4.2 algebraic concepts enable operators on structures. Using a mapping between two algebraic structures, cell boundaries can be easily introduced. These operators and mappings benefit from the abstract topological specification and operate on spaces of arbitrary dimension and topology.

Given the 4D hypercube structure, the boundary from the boundary, a quadrilateral-on-cell operation, is obtained by the following piece of code, which allows for the calculation of the boundary at compile time:

```
typedef
      gsse::boundary<4, 2, gsse::cell_hcube>
boundary_t;
```

To allow for such a concise formulation of the task of boundary extraction, several supporting constructs need to be implemented, as was illustrated in the example shown in Section 3.3. At the core of the shown boundary operation is the availability of detailed information regarding the cell's structure. This structure can be made available using hard coded look up tables; this, however, results in a loss of flexibility, as new look up tables need to be incorporated manually. Alternatively, it may be possible to determine the structure algorithmically. Any such algorithm naturally needs to take into account the type of the cell (e.g., if it is a cuboid or simplex cell). Since the combinatorical problem of determining all structural components increases dramatically [105] with dimension, a run time algorithm may become prohibitively expensive in case of high performance applications. Therefore, the availability of an algorithm at compile time which can accommodate a change of dimensionality without the need to manually provide and adjust static tables is desirable, even at the expense of increased compile times.

Where boundary operations are essential for numerical schemes employed for the solution of partial differential equations, the extraction of iso-surfaces is an invaluable tool for the processing of results obtained from any such calculations.

Marching cubes [106] has been a key algorithm for the rapid extraction of surfaces. It however not only has issues with topological ambiguity, but has also been handicapped by patenting issues [107]. Both of these issues have been addressed by the development of the marching tetrahedra [108], which suffers neither patent issues nor topological ambiguity.

The following presents a means utilizing generic programming methods and paradigms, which allows to forgo the use of a hard coded look-up table and instead utilize higher level representations of the underlying topological information or determine a look-up table as needed. The procedure is realized using the capabilities already present in the GSSE In this fashion the procedure is generalized to arbitrary dimensions. This allows to supply an algorithm for surface extraction which is suitable for data of any supplied dimension, since it will adapt to the supplied dimension.

The original n-cube cell is decomposed into simplices. It is easy to obtain all the edges from the simplices generated in this fashion. The edges are essential for the surface extraction process, since the vertices which are subsequently used to spawn the extracted surface representation are derived directly from them, by means of interpolation. While it may be tempting to discard the simplex information, since vertex generation is so tightly coupled to the edges, the simplices are essential for the generation of surface elements as they eliminate ambiguity. For, while the generation of surface elements within any given simplex can be done consistently, the lone set of vertices in an n-cube leaves too many degrees of freedom undetermined, which is also the source

for ambiguity, when using the marching cubes algorithm [109]. While this situation is easily containable in the three-dimensional case, the effort required in higher dimensions increases dramatically [105].

The task of determining the simplices filling an n-cube does not have a unique solution, as is already apparent in the case of a three-dimensional cube, which admits a filling consisting of either five or six tetrahedra. While the generation of a minimal number of elements is desirable, the minimization of the element count is not a driving motivation, but symmetry needs to be considered in order to enable a seamless repetition of neighbouring cells. In the particular case of the three-dimensional cube this condition singles out the decomposition into six tetrahedra. In more general terms the composition in which all tetrahedra share the space diagonal which crosses the n-cube cell meets the required symmetry conditions.

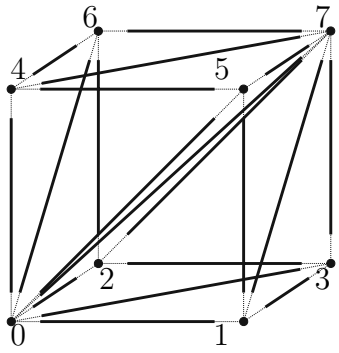


Figure 6.1: Vertices forming a 3-cube. All of the shown edges belong to tetrahedra obtained by the described algoirthm.

The proposed algorithm utilizes information available from the n-cube. When utilizing the topological mechanisms available in the GSSE, very detailed information regarding the arrangement of lower dimensional components is already available. The proposed algorithm therefore reuses as much of this information as possible.

Starting with an n-cube cell $\mathbf{Q}_0$; first the space diagonal $\mathbf{D}_0$ is extracted and its vertices stored for simplex construction, before the boundary elements of the n-cube $\partial\mathbf{Q}_0$, which are themselves n-cubes $\mathbf{Q}_i \in \partial\mathbf{Q}_0$ again, are queried. From each of the boundary elements $\mathbf{Q}_i$, which are of course of lower dimension than before, $\dim \mathbf{Q}_0 = \dim \mathbf{Q}_i + 1$, the space diagonal $\mathbf{D}_i$ is extracted and its vertices added to the simplex again before recursing to once more determining the boundary elements $\mathbf{Q}_{ij} \in \partial\mathbf{Q}_i$ of the current cell $\mathbf{Q}_i$. The procedure is applied recursively, until the cells are themselves edges and thus cannot support a distinct diagonal. The space diagonals, thusly generated in conjunction with the edge boundary of the n-cube form the complete set of edges required to extract a surface representation from the n-cube. Simplices are determined while selecting the space diagonals and aggregating the encountered vertices.

An example using a 3-cube is provided to further illustrate the procedure of the algorithm. Considering an n-cube cell formed by vertices as illustrated in Figure 6.1. The n-cube cell is given as $\mathbf{Q}_0 = \{0, 1, 2, 3, 4, 5, 6, 7\}$. The first step is to extract the diagonal $\mathbf{D}_0 = \{0, 7\}$. The vertices $\{0\}$ and $\{7\}$ are two initial vertices of all the tetrahedra, which will be constructed, thus this edge is shared among all the generated tetrahedra. The boundary elements $\partial\mathbf{Q}_0 = \cup\mathbf{Q}_i = \{\{0, 1, 2, 3\}, \{4, 5, 6, 7\}, \{0, 2, 4, 6\}, \{1, 3, 5, 7\}, \{0, 1, 4, 5\}, \{2, 3, 6, 7\}\}$ are then each considered in turn, giving the diagonals $\{0, 3\}$, $\{4, 7\}$, $\{0, 6\}$, $\{1, 7\}$, $\{0, 5\}$, $\{2, 7\}$, respectively. As can be seen every one of the additional diagonals contains a new vertex to build a tetrahedron, when com-

paring to the already selected $\{0, 7\}$ vertices. Adding this vertex the list of partial simplices is $\{0, 3, 7\}$, $\{0, 4, 7\}$, $\{0, 6, 7\}$, $\{0, 1, 7\}$, $\{0, 5, 7\}$, $\{0, 2, 7\}$, which are now missing a single vertex for completion. As no further diagonals can be generated no further recursion is initiated. However, the vertices of the edges are examined to complete the forming simplices. This finally gives the tetrahedra to be $\{0, 1, 3, 7\}$, $\{0, 2, 3, 7\}$, $\{0, 4, 5, 7\}$, $\{0, 4, 6, 7\}$, $\{0, 2, 6, 7\}$, $\{0, 1, 5, 7\}$.

The simplices are traversed by decomposing them into their edge components. Thereby the membership relations between the simplices and the edges are available naturally. The overall complexity of the algorithm is broken down due to the reuse of the GSSE's topological operations. However, edges may be evaluated several times, as they belong to several different simplices, which comes as overhead if left unaddressed.

As each of the generated simplices is traversed the vertices originating from the edges are used for the formation of surface elements immediately, thus once all simplices have been traversed the surface has been extracted.

The problem of splitting an n-cube into simplices is relatively expensive in terms of computational effort and grows considerable as dimension increases. It is therefore prudent to reuse the extracted topological structures as much as possible. Fortunately, the topological decomposition of an n-cube into simplices is required to be performed only once and can be reused multiple times once the information has been initialized. Furthermore, traversal of the individual simplices for every cell can be omitted, when the information is condensed into an associative container, which mimics the hard coded look-up tables already in use. However, instead of only guiding subsequent formation of elements one by one, the generic approach may be used to provide all of the resulting surface elements in its own container. Since the GSSE can provide topological information it is feasible to implement the entire algorithm using meta programming techniques [31][32].

Thus it is possible to concisely access a surface extraction in the following manner:

```
tempalte<typename InputDomainType, typename SurfaceDomainType,
         typename ParameterPackageType>
void surface_extractor(InputDomainType& input_domain,
                       SurfaceDomainType& output_surface,
                       const ParameterPackageType& parameters)
{
  typedef typename get_cell_type<InputDomainType>::type
                                                    cell_type;
  typedef typename get_cell_type<SurfaceDomainType>::type
                                                      sim_type;
  marching_simplices<cell_type, sim_type,
                     ParameterPackageType>::type
                     march(parameters);

  gsse::traverse<at_cl>
  [
     insert(output_surface, march(_1))
  ](domain);
}
```

It utilizes the functional paradigm in the spirit of Boost Phoenix [110][34] which is enabled by GSSE facilities for traversal and extends the GSSE by a surface extraction mechanism. The parameter pack can be used to supply a prescription for interpolation, so as to provide an additional degree of adaptivity.

```
template<typename ContainerType>
struct generator
{
    ContainerType return_template;
    template<typename InitialisationType>
    generator(InitialisationType*)
    {  ...  }
    template<typename QuadCellType>
    ContainerType operator()(const QuadCellType& quad) const
    {  ...  }
};
```

The `InitialisationType` encodes the instructions to initialize the `return_template` at construction time. Subsequent calls to the `operator()` use this information to assemble and return the correct surface elements by remapping the representation, which is local to the cell, to the global indices. By initializing the member variable in this fashion in a templated constructor, where the type is utilized for control enables the structs to share the same type and thus be easily stored in the associative container, while at the same time returning different values at evaluation time.

Since it is not possible to invoke the templated constructor explicitly, the compiler needs to deduce the type automatically, a dummy pointer argument is used, whereby any instantiation of the passed type can be avoided by using

```
generator( (type*) 0 )
```

For compatibility with the containers in the STL, a default constructor is also required.

It should go unnoted that the size of the associative container increases quickly with increasing dimension, if symmetry is not utilized to reduce the number of different cases. This can be accomplished by establishing a canonical ordering which is then mapped to the rest by permutations corresponding to the symmetries.

6.4 Fibers

As has been indicated in the introduction to this chapter, fiber bundles (Definition 40) are inherently used in digital computers. In the same manner as the crude structures are assembled to construct increasingly complex topologies (Definition 28), these topologies can then be used to further provide fiber bundle structures beyond the bare basics. A very simple data structure available in the STL library, which illustrates the use of a fiber bundle, is:

`std::map` is an explicit fiber bundle.

```
std::map<Key, Data, Compare, Allocator>
```

The `Key` type represents the base space, while the `Data` type specifies the data type of the fibers. This implementation also relies on a relation for comparison of elements to be available and also allows for control of the allocation of memory.

Modelled after this run time example, there is also a compile time container available:

`mpl::map` is the compile time equivalent to `std::map`.

```
mpl::map<p1, p2, ..., pn>
```

Here, the types such as `p1` are expected to be models of compile time pairs (`mpl::pair`), which link the data types of the fibers to the base space data types.

Where these two examples are each completely contained within its particular realm of run time and compile time respectively, it is also possible to bridge these two worlds. Such an implementation is provided as part of the Boost Fusion [103] library in the form of `fusion::map` which links a compile time data type to a run time value:

Boost Fusion allows to mix run time and compile time.

```
fusion::map<T1, T2, ..., Tn>
```

This illustration is not to be interpreted that the other containers are not also fiber bundles, however, a `std::vector` for example corresponds to an implicit base space, thus masking the underlying structure.

The availability of a topological base spaces along with fiber bundles allows to easily construct further topological structures, even more exotic varieties such as Möbius strips [111][75]. The interplay of these two very basic concepts is thusly essential to the construction of highly complex topologies, as presented in the following code snippet. The topological base space is modelled by an explicit (sparse) container structure, where integers are used to address each element. The fiber space is also modelled by an explicit (sparse) container structure, but the fiber bundle sections are accessed by strings. The final mapping to a `double` concludes the necessary concepts from Definition 61.

Example explicit 0D cell fiber bundle.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<0> >
        , gsse::pair<tag_storage,   double >
        , gsse::pair<tag_index_bs,  long>
        , gsse::pair<tag_index_fs,  std::string>
    > fb_scalar_field_t;
```

From this vantage point, the standard containers found in the libraries (STL, MPL and Fusion) appear as special cases compared to more complex topologies. The need for more complex topologies arises from applications requiring the modelling of manifolds (Definition 35) as arise from topics elaborated in Chapter 5 for instance.

The ability to model manifolds in a convenient and powerful manner together with the fiber bundle concept also provides a clear strategy of how to realize fields such as given by Definition 61, Definition 62 or most generally Definition 65. Since each of these fields is in essence the section of a fiber bundle (Definition 41), it is only required to associate a single entity of the type

corresponding to the field to every element of the base space, as given in the code snippet by `gsse::vector_ct<>`.

Example vector field example.

```
typedef gsse::map<
          gsse::pair<tag_dimension, mpl::int_<0> >
        , gsse::pair<tag_storage,   gsse::vector_ct<double,
                                                    double,
                                                    double> >
        , gsse::pair<tag_index_bs,  long>
        , gsse::pair<tag_index_fs,  std::string>
    > vector_field_t;
```

The existence of any kind of quantity which is to be distributed over all of the topological base space can also serve as a gauge to further qualify how the collection of topological cells subdividing the whole topological space should be shaped. The collection of subdivisions is usually called a mesh and the task of the generation of meshes is called meshing [112][113]. The estimation and judgement of the quality of the mesh not only depends on the quantity being resolved but also the application the mesh is intended for [114].

6.5 Geometric Structures

As explored in Section 4.6, curves (Definition 45) are a versatile entity in the context of differential geometry as well as in physical applications as seen in Section 5.3. Geometric structures such as curves are defined on top of topological spaces, whose discrete representation has already been outlined in Section 6.3. The structure of fiber bundles as described in Section 6.4 allows for a simple storage of vector fields (Definition 62), which are intrinsically connected to their integral curves (Definition 63). This connected notion should be carried over from the continuous setting to the world of digital computers.

However, since the data items, such as vector fields, are stored only on discrete points, associated with the CW complex (Definition 34), a curve can also only be approximated. Returning to a very basic notion of the curve, as a mapping (Definition 20) associating a single parameter to a set of points in a manifold (Definition 37), it becomes apparent that an approximation of a curve can be achieved by storing a set of points with an additional mapping relating the CW complex and the curve.

Thus an integral curve, as a one-dimensional differentiable manifold, has its own representation, which is initially completely independent of the topological space it traverses. However, a mapping must exist and also be explicitly available, in order to use a curve within the discrete digital world. This mapping is readily found, when turning to atlases (Definition 36) and their charts. Thus by calculating the coordinates of points, it is possible to relate the points of the curve with its embedding manifold. However, in a manifold the obtained coordinates are valid only locally and worthless, since they degenerate to a mere collection of numbers, without knowledge of the chart in which they are obtained. A simple example is obtained by specifying points using barycentric coordinates within each of the cells. Without knowledge which cell to reference it is not possible to correctly locate the point in the CW complex. In the case of an affine space (Def-

inition 79), the obtained coordinates have global scope and no further information needs to be specified, beside the knowledge that it is indeed an affine space.

The points comprising the curve can either be represented using a CW complex but can also be given in an implicit manner, using a prescription to determine points from a parameter. In the case of integral curves, the derivative with respect to the parameter of the mapping used for point calculations must match the given vector field. The simple yet powerful structure of polynomials (Definition 19) makes them well suited for this task. In order to avoid issues with high polynomial orders due to Runge's phenomenon [115] interpolation is usually restricted to locally using low orders of polynomials, thus modelling the curve piecewise by several polynomials. This piecewise representation of a curve has become known as a spline.

Since the local nature of splines corresponds to the local nature of charts, they are well suited to directly provide coordinates for points of the curve, making them directly applicable in the previously described scheme.

The identification of points using coordinates seems to alleviate the problem of dealing with points belonging to two manifold structures, the solution suffers from imperfections due to the numerical issues connected with the calculation of the coordinates. It thus degrades the reliability of determining the identity of points, as it moves it from a problem related to set theory and topology to a geometric setting.

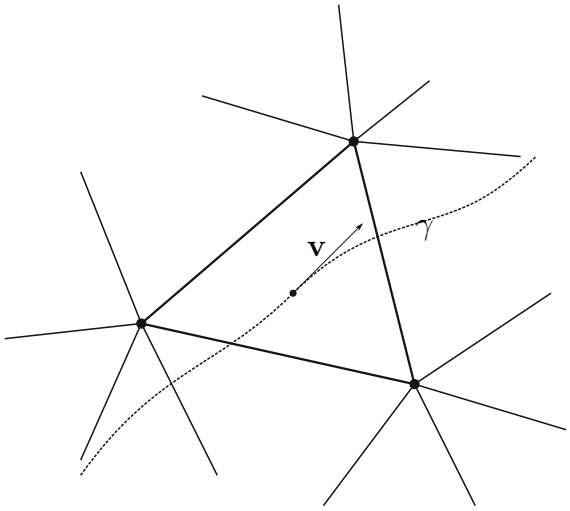


Figure 6.2: A curve γ passes through several elements, thus as the curve's parameter is varied the location corresponds to different cells.

In case the description uses local expressions, corresponding to cells, it is important to identify, when a curve moves from one cell to the next (illustrated in Figure 6.2), as it indicates the limit of its validity. Thus, the intersection of a curve with the boundary of a cell is worth to be investigated further.

In the case of a Riemannian base manifold and hence the availability of a mapping between p-forms and vectors due to an isomorphism as indicated in Equation 4.156, the boundaries of tetrahedral and hypercube cells can be geometrically represented using a normal vector $\mathbf{n}$ to the boundaries surface using a simple formula

$$(\mathbf{n}, \mathbf{x}) = d, \tag{6.3}$$

where $\mathbf{x}$ represents a point to be tested.

Given the curve has a local polynomial representation in a parameter η

$$\gamma(\eta) = \sum_i \mathbf{c}_i \eta^i, \tag{6.4}$$

where the coefficients c_i are connected to the current cell, e.g., derived from quantities associated to this cell. Inserting Equation 6.4 into Equation 6.3, the intersections are obtainable as

$$\left(\mathbf{n}, \sum_i \mathbf{c}_i \eta^i \right) - d = 0. \tag{6.5}$$

Linearity allows to reformulate

$$\sum_i \left(\mathbf{n}, \mathbf{c}_i \right) \eta^i - d = 0 \tag{6.6}$$

showing the resulting polynomial in the parameter η, whose coefficients are given by the inner product of the curve parameters and the normal vector, whose roots indicate the possible intersections. The correct intersection can then be chosen from the calculated roots.

As a machine does not possess any intuitive notion which boundary component may be intersected and which not, the intersection needs to be evaluated for every boundary component (sub-cell) of the currently investigated cell. The boundary sub-cells and their face normals thereby need to be available from the base topological framework.

The following piece of code illustrates an implementation of the outlined algorithm using the GSSE infrastructure.

Listing 6.1: Determination of points of transit from cell to cell. Data type definitions.

```
typedef gsse::boundary<DIM, DIMB, CellTopology> GFacetOnCell;
typedef gsse::result_of::boundary<DIM, DIMB, CellTopology>::type
    GFacetOnCell_result;

gsse::container<3, numeric_t> ci;
```

These data types are defined and evaluated at compile time, where `DIM` represents the global manifold dimension and `DIMB` represents the corresponding facet sub dimension. The type `CellTopology` describes the discrete cell representation of the underlying manifold as explicit mesh, e.g., a simplex mesh or a cube grid, as outlined is Section 6.3. Finally, the `ci` need to be available or calculated form values defined on the base topology, as sketched in Section 6.4, to represent the curve γ according to Equation 6.4. Here a cubic polynomial, thus of degree 3, is chosen for illustration. An application of this procedure, where the coefficients $\mathbf{c}_i$ are determined from physical considerations, is available in Section 7.2.1.

Where the previous snippet of code presented a compile time structure, the next piece of code is evaluated at run time. The `new_cell` along with the `roots` of the curve parameter are calculated from the given current `cell`, its boundaries and their geometry, and the `ci` as indicated by the global mesh structure.

Listing 6.2: Determination of points of transit from cell to cell. Run time calculation.

```
GFacetOnCell            facet_on_cell;
GFacetOnCell_result     facet_result = facet_on_cell(cell);
Cell new_cell;

for (size_t i = 0; i < gsse::size(facet_result); ++i)
{
  gsse::vec_t nvec = gsse::normal_vector(facet_result [i]);
  numeric_t   d    = gsse::in(nvec, facet_result[i][0] );

  gsse::polynomial<3,numeric_t>   poly(0, -d);
  gsse::algorithm::traverse (poly,
                             gsse::in( nvec, gsse::acc(ci)) );
  gsse::vec_t roots = gsse::poly_root_calculator( poly );

  if (roots_valid (roots) )
  { new_cell = gsse::set::not_in (
                          cell_on_facet( facet_result[i]) , cell
                          );
    break;
  }
}
```

The local description due to the use of splines limits the order of the used polynomials, thereby also limiting complications connected to the calculation of the roots of the polynomial. Once the next cell has been determined, the next polynomial of the spline corresponding to the new cell can be used.

6.6 Integration

Where many concepts and notions are defined in a purely local fashion, others, such as expectation values (Definition 101) or distances (as presented in Section 4.8), have a non-local nature which is accounted for by utilizing integrals for their description. In many cases the determination of the result of integration can make use of the fundamental theorem of calculus or its generalization of Stokes's theorem.When this is to be used to obtain a value connected to the integrated expression, it relies on the result being expressible in the limited terms of elementary functions, which, unfortunately, severely limits the range of applicability. For this reason several alternate methods which are not limited by elementary functions have been developed.

6.6.1 Interpolating Integration

A possible venue to provide a means of evaluating integrals which do not match an elementary function, is by using approximations based on selected elementary functions to calculate the value of the integral. Different strategies, sharing the approach of relying on polynomials (Defini-

tion 19) and their properties have been developed, since polynomials are not only very adaptable, but also easy to integrate, as the polynomial functions are closed under integration. Common to all of these methods is that an estimate for the integral is obtained by sampling the integrand at several discrete points obtained by subdivision of the integration domain.

A simple strategy is to approximate the integrand directly by constructing an approximate polynomial which is then integrated. In order to avoid high orders of polynomials, which have proven to be numerically problematic, local approximations of low polynomial order can be used. Following this idea using equidistant subdivisions leads to the Newton-Cotes method. The use of different degrees of polynomials for local approximation leads to rectangular, trapezoid, and quadratic estimates.

It should not go without notice that it is possible to construct schemes of higher order by using linear polynomials, which give rise to the trapezoid quadrature procedure. This is accomplished by applying Richardson extrapolation (see Appendix C) to the trapezoid quadrature to improve its accuracy and is known as the Romberg method, Romberg rule or Romberg quadrature. By repeating the application of Richardson extrapolation, it is possible to obtain a quickly converging algorithm, which furthermore allows for the reuse of previously computed values. The fast convergence of the algorithm, however, relies on the existence of derivatives of an order corresponding to the order of approximation.

A different approach for efficient quadrature turn to utilizing orthogonal polynomials, to derive a different method of numerical integration, which is known as Gauß quadrature. It draws on the properties of orthogonal polynomials to precompute weighting coefficients w_i associated with positions ξ_i to formulate the integral as

$$\int_D f(x)\mathrm{d}x = \sum_i w_i f(\xi_i). \tag{6.7}$$

The quality of the approximation depends on an appropriate choice of the orthogonal polynomials based on the function to be integrated. The weights and positions for several orthogonal polynomials have been precalculated for the case of one-dimensional integration [116].

Several variants of all of these algorithms have been developed, as well as methods differing in their particulars, but using the same approaches [117], resulting in a selection of numerical integration schemes from which to choose. There is, however, no best method of quadrature in general, especially if there is no a priori knowledge if the integrand. Thus the creation of options from which to choose from is of high importance in this field. This has proved to be a set of powerful tools in one dimension, however, as the dimension of the integral increases, these methods become less attractive, due to an exponential increase in required sampling points and function evaluations [118] to maintain quality of approximation, which drastically increases the computational burden. Therefore, alternatives to these methods should also be considered, if multi-dimensional integration is required or desired.

6.6.2 Stochastic Approach to Integration

A different approach to the subject of integration, which does not deteriorate as the dimension of the integration domain increases has also been developed. The stochastic nature of the procedure has earned it the name of Monte Carlo method [119][120][121][122][123][124]. Since

it is not easily possible to evaluate multi-dimensional integrals by simple deterministic methods described in Section 6.6.1. Monte Carlo methods provide a simple and elegant solution which becomes increasingly attractive as the dimensionality of the integration domain increases. The task of determining the value of a given integral (Definition 94)

$$I = \int_{\Omega} f \mathrm{d}y \tag{6.8}$$

is reexpressed to illustrate the general idea behind the stochastic integration methodology.

$$I = \int_{\Omega} f(x)\mathrm{d}y = \int_{\Omega} \frac{f(x)}{p(x)} p(x)\mathrm{d}y, \tag{6.9}$$

$$p(x) \neq 0 \leftrightarrow f(x) \neq 0 \tag{6.10}$$

In this way the standard integral given in Equation 6.8 can be viewed as an expectation value of a random variable (Definition 98), if $p(x)$ is viewed as a probability density function (Definition 100). For $p(x)$ to be considered a probability density, it has to meet the following restrictions.

$$\int_{\Omega} p(x)\mathrm{d}y = 1 \tag{6.11a}$$

$$p(x) \geq 0 \quad \forall x \in \Omega \tag{6.11b}$$

The reinterpretation as an expectation value (Definition 101) in conjunction with a probability density also provides an intuitive way for moving from the continuous realms of the initial problem domain to a discrete one, such as present in digital computers.

Calling on the law of large numbers, it is possible to relate several discrete realizations modelled after the probability density $p(x)$ sampling the integration domain Ω to the expectation value I as the sample mean approaches the expectation value.

$$I = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{N} \frac{f(x_i)}{p(x_i)} \tag{6.12}$$

It should not go unnoted that the (expectation) value I does not depend on the choice of the distribution function. The variance

$$\sigma = \lim_{N\to\infty} \sqrt{\sum_{j=0}^{N} \left(\frac{f(x_j)}{p(x_j)} - \frac{1}{N} \sum_{i=0}^{N} \frac{f(x_i)}{p(x_i)}\right)^2}, \tag{6.13}$$

however, is not independent of this choice, which can be used as an estimate for the quality of the estimate for a given N using the three σ rule, which ensures that the true value of the integral lies within the interval of $\mu \pm 3\sigma$ with a probability of 0.997 percent.

Using the C++ programming language and relying on already available libraries [110] the central part of the implementation of a Monte Carlo integrator may be accomplished in a very concise manner, as the next listing of code demonstrates.

```
using bacc = boost::accumulators;
bacc::accumulator_set<numeric_type,
        bacc::stats<bacc::tag::mean,
        bacc::tag::error_of<bacc::tag::mean> > > estimator;

random_number_generator_type random_source;
function eval;

for (size_t i(0); i < number_of_samples; ++i)
{
  estimator(eval(random_source() ));
}
```

It uses the `boost::accumulator` library to conveniently keep track of the mean of the evaluated points, thereby constructing an estimator for the true value of the integral under consideration. This estimator contains the mean value, which is the main estimate along with the corresponding statistical error. The library's facilities make it simple to realize this behaviour by simply specifying the corresponding tags, `boost::accumulators::tag::mean` and `boost::accumulators::tag::error_of<>` respectively. Furthermore, it requires a source of random variables (see Appendix B), provided here in the form of `random_source`. The function to be integrated itself is represented as a function object `eval`. The estimator is then fed with function evaluations corresponding to a sequence of random numbers.

As can be seen, the dimensionality of the problem does not enter into this implementation explicitly. It is completely encapsulated within the invocation of the `eval` function object, which relies on an appropriate random number generator.

When considering parallelization of the loop in this code, special care must be taken on the one hand of the estimator, which internally keeps track of the mean value and thus must be handled appropriately. On the other hand, the generation of random numbers in parallel may also result in issues, when left unchecked.

Chapter 7

Dynamics in Action

> He that will not apply new
> remedies must expect new evils.
>
> Francis Bacon

All of the descriptions of dynamics given in Chapter 5 can be rendered using the simple concept of curves (Definition 45) associated with the evolution of the system, whether they appear merely as a means to track evolution or are seen as an essential component such as representatives of the flow (Definition 67) driving evolution.

The following expositions will focus on the use of the Hamiltonian formalism, where curves, also called trajectories, are an essential component of modelling, since the whole phase space is subject to evolution due to the Hamiltonian flow. Since here an individual curve, which is an integral curve (Definition 63) of the Hamiltonian vector field (Definition 62), is a representative of the flow, the evolution of any point along the curve is described by a simple shift along the curve. Since the integral curves of a Hamiltonian vector fields never intersect (with the exception of singularities), the evolution of every point is uniquely determined by knowing the integral curve.

Thus, using a discretization of the domain of interest as described in Section 6.5, it is possible to derive the evolution of a dynamic system in the environment of digital computers. However, the theoretical and practical considerations hampering direct utilization still apply.

Thus the reduction such as due to Boltzmann's equation becomes a natural choice for further considerations. Boltzmann's equation, however, also proved to be elusively difficult to solve in a rigorous manner. This resulted in intense investigation of properties which could be extracted without actually solving the equation, or the gauging of the appropriateness of approximations.

7.1 Equilibrium and Macroscopic Quantities

Faced with severe difficulties of finding solutions of Boltzmann's equation, Boltzmann himself was able to derive the shape of the equilibrium state of the distribution ρ in the shape of the

Maxwell-Boltzmann distribution

$$\rho_{\text{equ}} = \alpha \mathrm{e}^{\beta H(\mathbf{p},\mathbf{q})}, \tag{7.1}$$

where $H(\mathbf{p},\mathbf{q})$ is the system's Hamiltonian, β is connected to the average energy and thus the temperature T and α a factor additionally accounting for the number of degrees of freedom.

Furthermore, Hilbert's investigations [125] of the dynamics of Boltzmann's equations using an expansion into a power series of the form

$$\rho = \sum_{i=-1}^{\infty} \lambda^i \Psi_i \tag{7.2}$$

revealed that in the case the collision term $Q(f)$, and by extension $S(\gamma,\bar{\gamma})$, is symmetric, the evolution described by Boltzmann's equation can be captured by using five distinct functions. These can incidentally be linked to the macroscopic quantities of density, temperature and velocity. Thus establishing that the evolution of systems described under these circumstances can be characterized using hydrodynamic descriptions [125].

The interconnection between the theory of Boltzmann's description and the macroscopic transport mechanisms becomes more apparent in lieu of the theory in Section 5.5, which allows to derive a series of macroscopic quantities as well as equations connecting these quantities and governing their evolution. In this context, Boltzmann's statement regarding the shape of the equilibrium provides a guide for admissible assumptions during further derivations, while Hilbert's theoretical result provides a strong argument how many quantities and hence equations must be considered.

When the conditions regarding symmetry are satisfied, the derivation of further quantities is not only not furthering the cause of obtaining better approximations, but on the contrary introduces unwarranted complications due to the computational effort which needlessly squanders resources. Thus the crunching of numbers can be reduced by employing information, which has been arrived at through completely theoretical means.

Macroscopic Description of Electrons in Semiconductors

While Boltzmann conducted his research in the field of gas dynamics, his equation has found applications beyond this initial field. One of these fields is the description of electron dynamics in semiconductors. Where the scattering term in gas dynamics mainly considers interactions between different particles, the case of semiconductors focuses on the interaction of an electron with the surrounding semiconductor crystal. Thus, while in the field of gas dynamics the energy contained in the gas can be considered a conserved quantity, especially if not considering any boundaries, this is no longer appropriate in the case of solid state physics, since the electrons interact with the crystal lattice and thus gain or remove energy form the considered system. This necessitates a change in the symmetry of the scattering operator $Q(f)$[1].

Since the symmetry considerations have been involved in the determination of the required number of equations, a reexamination of the previously sufficient set of equations is in order in this context. Thusly motivated, a ladder of equations has been derived following the scheme sketched

[1] Noether's theorem specifies that any symmetry results in a conserved quantity [126].

in Section 5.5, which exceeds the previously established systems in complexity [127][128][129]. The complexity is further increased, as any system of equations derived from Boltzmann's equation must be solved self consistently with a coupled Poisson equation, due to the charge of the involved particle. The involvement of charge, furthermore, leads to a doubling of effort, as positive and negative particles may be considered.

From among the derivable sets of equations the most widely used is also the simplest, which can also be easily derived on empirical considerations. The equations describe convection-diffusion, also called drift-diffusion, problem.

$$n = \int \rho \, \mathrm{d}\mathbf{p} \tag{7.3}$$

$$\mathbf{J} = \mu(\frac{1}{\mathrm{k_B}}\mathrm{grad}\,(n\,T) + q\mathbf{E}\,n) = -\frac{q}{m}\int \rho\mathbf{p}\,\mathrm{d}\mathbf{p} \tag{7.4}$$

$$\frac{\partial}{\partial t}\,n - \frac{1}{q}\mathrm{div}\,(\mathbf{J}) = -R \tag{7.5}$$

n is the number of electrons, T the temperature, $\mathbf{J}$ the electron current, m and q are the electron mass and charge, respectively, $\mathbf{E}$ is the electric field, and R a term describing recombination with holes.

This model has been in use from early days of the semiconductor industry, when computational resources were far scarcer than they are today. For this reason it was of greatest importance for the applicability of this approach to develop a discretization scheme which maintains the level of accuracy with as coarse a spatial discretization as possible. Thus the computational effort has been considerably reduced, by accounting for the exponential distribution of electrons and at the same time consistently modelling the potential linearly [130].

Aiming to increase the accuracy of the physical description, the same discretization scheme has also been applied to the equations derived using higher powers of $\mathbf{p}$. This, however, is only of limited success, as the increased computational effort is additionally coupled with problems of obtaining a solution at all. Thus the more complex models have, so far, not found great adoption. While the problems encountered in dealing with higher order transport models may very well be attributed to their heightened complexity alone, it cannot be neglected that the applied discretization scheme, which mimics the highly successful Scharfetter-Gummel discretization of the drift diffusion model, is no longer consistent even in the case of a hydrodynamic, in this context also often called energy transport, description. Thus as the physical models of increasing complexity are additionally hampered by inconsistencies in their mapping from the continuous to the digital world, numerical difficulties are hardly surprising. While these may be unavoidable due to added degrees of freedom, even the most rigorous theoretical derivation of models will be greatly handicapped and impaired, when the final translation to a discrete setting is fragmented, especially in the unforgiving world of non-linear partial differential equations.

Despite the problems encountered in further refinement, the value of this approach to obtaining approximations to Boltzmann's equation should not be neglected. Especially when considering the case of the drift-diffusion model, the required computational resources are comparatively modest compared to alternate methods of tackling the problem presented by Boltzmann's equation.

7.2 Tracing Phase Space – Components from Theory

Where Section 7.1 described the dynamical system due to Boltzmann's equation using differential equations, an approach using the integral form of Boltzmann's equation as introduced in Section 5.6 is provided here. The presented integral equation is approached using the formalism described in Appendix A to approximate solutions. In contrast to the contraction to macroscopic quantities, as described in Section 7.1 this approach makes direct use of the phase space and its geometry by casting trajectories through it. As already outlined in Section 5.4, trajectories alone are no longer sufficient for an adequate description but a mechanism to change between trajectories is also required. The theory outlined in Section 5.6 is first theoretically further refined in order to provide guidelines usable for implementation, reusing components from Section 6.

The continuous model of the evolution of the dynamical system described in Section 5.6, needs only minor adaptations in order to be made accessible to the discrete world of digital computers. The adaptation is briefly outlined to provide a guide of how independent components may be assembled to be applied to a dynamical system.

Since Boltzmann's equation can be cast in the form of an integral equation, it is possible to construct a resolvent series. Recalling Equation 5.51b:

$$\rho(\mathbf{p},\mathbf{q},t) = \int_a^t \int e^{-\int_\tau^t \lambda(\xi)\mathrm{d}\xi} S(\bar{\gamma},\gamma(\tau))\rho(\bar{\gamma},\tau)\mathrm{d}\mathbf{p}'\mathrm{d}\tau + e^{-\int_a^t \lambda(\xi)\mathrm{d}\xi}\rho(a) \tag{7.6}$$

Proceeding as outlined in Appendix A by recursive insertions leads to:

$$\int_a^t\int_a^{t'}\iint e^{-\int_\tau^t \lambda(\gamma_1(\xi))\mathrm{d}\xi} S(\gamma_2(t'),\gamma_1(t'))e^{-\int_{\tau'}^{t'} \lambda(\gamma_2(\xi))\mathrm{d}\xi} S(\gamma_3(t''),\gamma_2(t'')) \tag{7.7}$$
$$e^{-\int_a^{t''} \lambda(\gamma_3(\xi))\mathrm{d}\xi}\rho(\gamma_3(a),a)\mathrm{d}\mathbf{p}'\mathrm{d}\mathbf{p}''\mathrm{d}t'\mathrm{d}t''$$

It contains two scattering events, hence uses three different trajectories (Definition 46), which are connected at their end points by the condition that they agree in their $\mathbf{q}$ component using the corresponding parameter, while allowing discontinuous transitions in their $\mathbf{p}$ components, as already established in Equation 5.47. Recalling it in adapted form it reads:

$$\gamma_i(t) = (\mathbf{p},\mathbf{q})\,, \tag{7.8a}$$
$$\gamma_{i+1}(t) = (\mathbf{p}',\mathbf{q}) \tag{7.8b}$$

Another important fact is that the sequence of trajectories here, actually moves back in time from the point of interest to the initial point. Thus the algorithms utilizing this approach directly are aptly called backward algorithms [131].

The trajectories encountered here correspond to the integral curves (Definition 63) of the Hamiltonian vector field (Definition 62). Each trajectory thus is a representative of the Hamiltonian flow (Definition 67), which governs the evolution of the system as a whole. Thus it becomes apparent that the solution is obtained by tracing the trajectories which follow the basic structures which define the geometry of the problem domain.

In the spirit of applying a Monte Carlo approach for the evaluation of the integrals, as described in Section 6.6.2 the factors

$$\frac{\lambda(\gamma_i(t))}{\lambda(\gamma_i(t))}, \tag{7.9}$$

which leave the value unchanged but allow for easier interpretation are introduced. After introduction of these factors and slight reordering the following is obtained:

$$\begin{aligned}\int_a^t\int_a^{t'}\iint \lambda(\gamma_1(t'))e^{-\int_{t'}^{t}\lambda(\gamma_1(\xi))\mathrm{d}\xi}\frac{S(\gamma_2(t'),\gamma_1(t'))}{\lambda(\gamma_1(t'))} \\ \lambda(\gamma_2(t''))e^{-\int_{t''}^{t'}\lambda(\gamma_2(\xi))\mathrm{d}\xi}\frac{S(\gamma_3(t''),\gamma_2(t''))}{\lambda(\gamma_2(t''))} \\ e^{-\int_a^{t''}\lambda(\gamma_3(\xi))\mathrm{d}\xi}\rho(\gamma_3(a),a)\mathrm{d}\mathbf{p}'\mathrm{d}\mathbf{p}''\mathrm{d}t'\mathrm{d}t''\end{aligned} \tag{7.10}$$

This identifies the terms of the form

$$\lambda(\gamma_1(t'))e^{-\int_{t'}^{t}\lambda(\gamma_1(\xi))\mathrm{d}\xi} = p_{\exp}(\lambda(\gamma_1(t')),t') \tag{7.11}$$

as the probability distribution function corresponding to the exponential distribution (Definition 103). While it is highly tempting to also identify the terms of the form

$$\frac{S(\gamma_2(t'),\gamma_1(t'))}{\lambda(\gamma_1(t'))} \tag{7.12}$$

with a probability distribution (Definition 99), it is not the case as the integration defining λ is not carried out with the corresponding states. However, such a term can easily by introduced recalling Equation 5.43

$$\int S(\mathbf{p},\mathbf{p}',\mathbf{q})\mathrm{d}\mathbf{p}' = \lambda(\mathbf{p},\mathbf{q}) \tag{7.13}$$

and analogously setting

$$\int S(\mathbf{p},\mathbf{p}',\mathbf{q})\mathrm{d}\mathbf{p} = \bar{\lambda}(\mathbf{p}',\mathbf{q}). \tag{7.14}$$

Inserting this expression provides

$$\frac{\int S(\mathbf{p},\mathbf{p}',\mathbf{q})\mathrm{d}\mathbf{p}'}{\lambda(\gamma_1(t'))}\frac{S(\gamma_2(t'),\gamma_1(t'))}{\int S(\mathbf{p},\mathbf{p}',\mathbf{q})\mathrm{d}\mathbf{p}'} = \frac{\bar{\lambda}(\gamma_2(t'))}{\lambda(\gamma_1(t'))}\frac{S(\gamma_2(t'),\gamma_1(t'))}{\bar{\lambda}(\gamma_2(t'))} = \frac{\bar{\lambda}(\gamma_2(t'))}{\lambda(\gamma_1(t'))}p_{\mathrm{scatt}}(t'). \tag{7.15}$$

Thus setting

$$p_{\mathrm{scatt}} = \frac{S(\gamma_2(t'),\gamma_1(t'))}{\bar{\lambda}(\gamma_2(t'))} \tag{7.16}$$

corresponds to a random variable of uniform distribution (Definition 102). Application of this to Equation 7.10 yields:

$$\begin{aligned}\int_a^t\int_a^{t'}\iint p_{\exp}(\lambda(\gamma_1(t')),t')\frac{\bar{\lambda}(\gamma_2(t'))}{\lambda(\gamma_1(t'))}p_{\mathrm{scatt}}(t')p_{\exp}(\lambda(\gamma_2(t'')),t'')\frac{\bar{\lambda}(\gamma_3(t''))}{\lambda(\gamma_2(t''))}p_{\mathrm{scatt}}(t'') \\ e^{-\int_a^{t''}\lambda(\gamma_3(\xi))\mathrm{d}\xi}\rho(\gamma_3(a),a)\mathrm{d}\mathbf{p}'\mathrm{d}\mathbf{p}''\mathrm{d}t'\mathrm{d}t''\end{aligned} \tag{7.17}$$

The final remaining exponential can be expressed as an additional integral over the parameter again representing an exponential distribution, thus finally obtaining

$$\int_a^t \int_a^{t'} \int_a^{t''} \iint p_{\exp}(\lambda(\gamma_1(t')), t') \frac{\bar{\lambda}(\gamma_2(t'))}{\lambda(\gamma_1(t'))} p_{\text{scatt}}(t') \\ p_{\exp}(\lambda(\gamma_2(t'')), t'') \frac{\bar{\lambda}(\gamma_3(t''))}{\lambda(\gamma_2(t''))} p_{\text{scatt}}(t'') \\ p_{\exp}(\lambda(\gamma_3(t''')), t''') \rho(\gamma_3(a), a) \mathrm{d}\mathbf{p}' \mathrm{d}\mathbf{p}'' \mathrm{d}t' \mathrm{d}t'' \mathrm{d}t'''. \tag{7.18}$$

While this expression is particular to the second term of the iteration, all following terms follow the same pattern and are expressible as a series of multiplications of probabilities along with a corresponding density.

The key components required for the approximation of the described dynamical system can be found from Equation 7.18 to be:

- Sections of trajectories moving through phase space
- Transitions from one trajectory to another
- Random number generation (see Appendix B)

7.2.1 Sections of Trajectories

Trajectories have already been introduced in Section 6.5 under the name of curves which shall be specialized for the context of dynamics. While trajectories corresponding to a Hamiltonian flow (Definition 67) fibrate (Definition 39) the whole phase space, only sections are required for the treatment of the dynamical system described by Boltzmann's equation.

Nevertheless, the sections of the trajectories under investigation are integral curves (Definition 63) of the Hamiltonian vector field (Definition 62) defined on the phase space. Considering the phase space as the co-tangent bundle leads to a split of this vector field, into two components which are still tangential to the curve. However, the two separated components are no longer each a vector field. Where the component corresponding to $\dot{\mathbf{q}}$ may be immediately identified with a vector field corresponding to the velocity, the part representing $\dot{\mathbf{p}}$ is a tensor field (Definition 65) of type $\mathcal{T}_1^0(\mathcal{V})$, thus a field of 1-forms, not a vector field. This field is given by the 1-forms corresponding to the force acting at each point of the manifold. While the nature of the components seems to have changed when projecting to the base manifold structures, both fields remain to be an expression of the Hamiltonian which defined the symplectic geometry.

The extent of the section of the trajectory traversed corresponds to a random variable using the exponential distribution described in Equation 7.11. A random variable (Definition 98) with the desired properties can be generated from a uniformly distributed random variable by inversion of the distribution function as outlined in Appendix B. Recalling Equation 7.11

$$\lambda(\gamma) e^{-\int_{t'}^{t} \lambda(\gamma(\xi)) \mathrm{d}\xi}, \tag{7.19}$$

the cumulative distribution function to this probability density distribution reads

$$1 - e^{-\int_{t'}^{t} \lambda(\gamma(\xi)) \mathrm{d}\xi}. \tag{7.20}$$

Equating this expression with a uniformly distributed random variable $\bar{r}$, e.g., from the interval $[0; 1[$, allows to formulate

$$\bar{r} = 1 - e^{-\int_{t'}^{t} \lambda(\gamma(\xi))\mathrm{d}\xi}, \tag{7.21}$$

$$r = e^{-\int_{t'}^{t} \lambda(\gamma(\xi))\mathrm{d}\xi}. \tag{7.22}$$

The new variable $r = 1 - \bar{r}$ is again a uniformly distributed random variable from the interval $[0; 1[$. This expression can readily be inverted to yield

$$\ln(r) = -\int_{t'}^{t} \lambda(\gamma(\xi))\mathrm{d}\xi = \Lambda(t') - \Lambda(t). \tag{7.23}$$

The domain of interest and the quantities associated with it are represented in a discrete manner in the digital computer as described in Section 6.3 and Section 6.4. The discrete nature has repercussions on the validity of the considered trajectory, such that it models only the local flow (Definition 66) instead of the original global flow. The border of validity of any given local trajectory is due to the finite size of the subdivision into cells (Definition 34 and discussed in Section 6.3), which form the base space of the fiber bundle (Definition 40 as implemented in Section 6.4) carrying the quantities which determine the shape of the trajectories.

Since the trajectories are nothing else than curves through phase space, the shape of the the trajectories is completely defined by the coefficients c_i, which were abstractly defined in Section 6.5. Here, they can be associated to physical quantities. c_0 corresponds to the position $\mathbf{q}$ of a particle, while c_1 is linked to $\mathbf{p}$. Finally, c_2 represents $\dot{\mathbf{p}}$. Thus the geometry of the trajectory is still tightly tied to the entities of the original symplectic structure.

The discrete nature makes it necessary to check, if the trajectory leaves the originating cell with any chosen value of the parameter and therefore actually serves as an upper limit for how far the trajectory is traversed, since the trajectory needs to be interrupted in case the parameter carries the new position beyond the initial cell, as already outlined in Section 6.5. Since the trajectories are one-dimensional, it is sufficient to compare the parameters corresponding to the intersection of a trajectory with a current cell's boundary to the value of the parameter obtained from the random distribution.

7.2.2 Trajectory Hopping

Both descriptions of Boltzmann's equation as provided in Section 4.9 and Section 5.6 acknowledge the defining nature of the scattering term on the right hand side. Thus the scattering term has to be mapped adequately to the realm of digital computing in order to expect the calculated result to be accurate. Furthermore, the scattering term is, together with the geometry defining Hamiltonian, the point of entry for physical modelling into the mathematical structure. Thus, the inclusion of different scattering mechanisms needs not only to be adequate from a mathematical point of view but also must withstand a physicist's scrutiny.

The scattering kernel S is therefore modelled as being composed from different scattering mechanisms S_i such that

$$S(\bar{\gamma}, \gamma) = \sum_i S_i(\bar{\gamma}, \gamma). \tag{7.24}$$

Thus, recalling Equation 7.14

$$\int S(\bar{\gamma},\gamma)\mathrm{d}\mathbf{p} = \int \sum_i S_i(\bar{\gamma},\gamma)\mathrm{d}\mathbf{p} = \sum_i \int S_i(\bar{\gamma},\gamma)\mathrm{d}\mathbf{p} = \sum_i \bar{\lambda}_i(\bar{\gamma}) = \bar{\lambda}(\bar{\gamma}) \tag{7.25}$$

and by Equation 7.16, which reads

$$p_{\mathrm{scatt}} = \frac{S(\bar{\gamma},\gamma)}{\bar{\lambda}(\bar{\gamma})} = \sum_i \frac{S_i(\bar{\gamma},\gamma)}{\bar{\lambda}(\bar{\gamma})} = \sum_i \frac{\bar{\lambda}_i(\bar{\gamma})}{\bar{\lambda}(\bar{\gamma})}\frac{S_i(\bar{\gamma},\gamma)}{\bar{\lambda}_i(\bar{\gamma})} = \sum_i \frac{\bar{\lambda}_i(\bar{\gamma})}{\bar{\lambda}(\bar{\gamma})} p_{\mathrm{scatt},i} \tag{7.26}$$

the probabilities for each of the individual scattering mechanisms $p_{\mathrm{scatt},i}$ can be related to the total scattering probability p_{scatt}. This relation also allows the selection of a given scattering mechanism by generating a uniformly distributed random variable r_{scatt} from the interval $[0;1[$. Since it is possible to uniquely identify the contribution λ_i of every scattering mechanism by its index i, it is possible to select a scattering mechanism by choosing an index i_{scatt}. When choosing this index i_{scatt}, the following inequality holds.

$$\sum_i^{i_{\mathrm{scatt}}} \frac{\bar{\lambda}_i(\bar{\gamma})}{\bar{\lambda}(\bar{\gamma})} \leq r_{\mathrm{scatt},i} < \sum_i^{i_{\mathrm{scatt}}+1} \frac{\bar{\lambda}_i(\bar{\gamma})}{\bar{\lambda}(\bar{\gamma})} \tag{7.27}$$

Equivalently, it is possible to forgo normalization for the selection, since it is to a common factor $\bar{\lambda}$ and expressing it in terms of rates as

$$\sum_i^{i_{\mathrm{scatt}}} \bar{\lambda}_i(\bar{\gamma}) \leq \lambda_{\mathrm{scatt},i} < \sum_i^{i_{\mathrm{scatt}}+1} \bar{\lambda}_i(\bar{\gamma}). \tag{7.28}$$

The nature of continued summation determines the scattering probability p_{scatt} along with the selection of the scattering mechanism i_{scatt}, which determines the beginning of the new trajectory, by means of the rejection technique as is outlined in Appendix B.

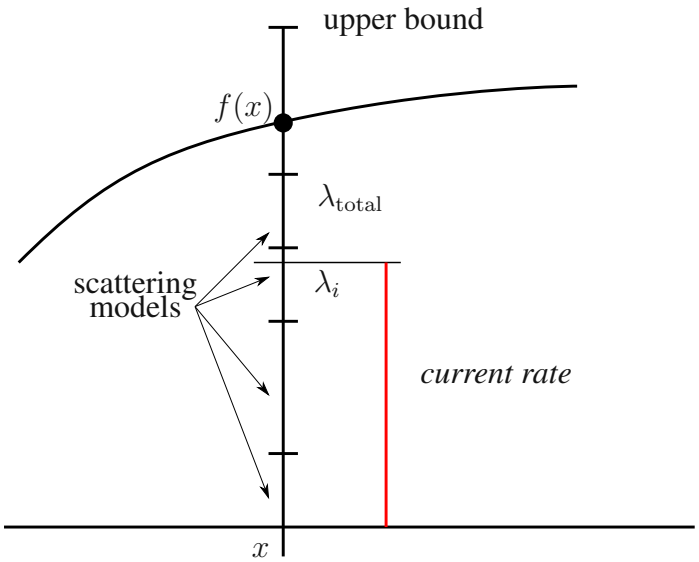


Figure 7.1: Schematic illustration of the rejection technique to evaluate scattering mechanisms.

Figure 7.1 provides an illustration of this particular case. At a given point x a *current rate* is chosen randomly using a distribution from the interval $[0; 1[$ multiplied by an upper bound for all the scattering. Then the scattering models are evaluated at this point x and the individual rates, thusly calculated are summed in order to determine the interval λ_i, which contains the *current rate*. The scattering mechanism associated with this interval is then chosen as the new state from which the next trajectory will propagate.

During the simulation the scattering models are evaluated repeatedly in order to choose the next state of the electron under consideration. Therefore high performance is among the requirements placed on the evaluation of the collision model.

7.2.3 Bridging Two Worlds

Applications implementing scattering mechanisms encountered in the field of solid state electronics, which change the states of electrons have been available quite some time. The foundations for the presented code have been been written in plain ANSI C, where all available scattering mechanisms are implemented as individual functions, which are called subsequently. The scattering models require a variable set of parameters, which leads to non-homogeneous interfaces in the functions representing them. To alleviate this to some extent global variables have been employed completely eliminating any aspirations of data encapsulation and posing a serious problem for attempts for parallelization to take advantage of multi-core CPUs. The code has the very simple and repetitive structure:

```
double sum = 0;
double current_rate = generate_random_number();

if (A_key == on)
{
  sum = A_rate(state, parameters);

  if (current_rate < sum)
  {
   counter->A[state->valley]++;
   state_after_A (st, p);
   return;
  }
}
if (B_key == on)
{      ...     }
...
```

Extensions to this code are usually accomplished by copy and paste, which is prone to simple mistakes by oversight, such as failing to change the counter which has to be incremented or calling the incorrect function to update the electron's state.

Furthermore, at times the need arises to calculate the sum of all the scattering models (λ_{total} in Figure 7.1), which is accomplished in a different part of the implementation, thus further opening the possibility for inconsistencies between the two code paths.

The decision which models to evaluate is performed strictly at run time and it would require significant, if simple, modification of the code to change this at compile time, thus making highly optimized specializations very cumbersome.

The functions calculating the rates and state transitions, however, have been well tested and verified, so that abandoning them would be wasteful.

Object oriented programming directly offers run time polymorphism by means of virtual inheritance. Unfortunately current implementations of inheritance use an intrusive approach for new software components and tightly couple a type and the corresponding operations to the super type. In contrast to object-oriented programming, current applications of generic programming are limited to algorithms using statically and homogeneously typed containers, but offer highly flexible, reusable, and optimizeable software components.

As can be seen, both programming types offer different points of evaluation. Run time polymorphism based on run time concepts [132] tries to combine the virtual inheritance run time modification mechanism with the compile time flexibility and optimization.

Inheritance in the context of run time polymorphism is used to provide an interface template to model the required concept, where the derived class must provide the implementation of the given interface. The following code snippet

```
template<typename StateT> struct scatter_facade
{
 typedef StateT state_type;
 boost::shared_ptr<scattering_concept> scattering_object;

 struct scattering_concept
 {
  virtual ~scattering_concept() {} ;
  virtual numeric_type rate(const state_type& input) const = 0;
  virtual void transition(state_type& input) = 0;
 };

 template<typename T> struct scattering_model :
  scattering_concept
 {
  T scattering_instance;
  scattering_model(const T& x) : scattering_instance(x) {}
  numeric_type rate(const state_type& input) const ;
  void transition(state_type& input) ;
 };

 numeric_type rate(const state_type& input) const;
 void transition(state_type& input) ;
 template<typename T>
 scatter_facade(const T& x) :
     scattering_object(new scattering_model<T>(x)) {}
 ~scatter_facade() {}
};
```

therefore introduces a `scattering_facade` which wraps a `scattering_concept` part. The virtual inheritance is used to configure the necessary interface parts, in this case `rate()` and `transition()`, which have to be implemented by any scattering model. In the given example the `state_type` is still available for explicit parametrization.

To interface this novel approach a core structure is implemented which wraps the implementations of the scattering models:

```
template<typename ParameterType>
struct scattering_rate_A
{
 ...
 const ParameterType& parameters;

 scattering_rate_A (const ParameterType& parameters):
  parameters(parameters) {}

 template <typename StateType> numeric_type
 operator() (const StateType& state) const
 {
  return A_rate(state, parameters);
 }
};
```

By supplying the required parameters at construction time it is possible to homogenize the interface of the `operator()`. This methodology also allows the continued use of the old data structures in the initial phases of transition, while not being so constrictive as to hamper future developments.

The functions for the state transitions are treated similarly to those for the rate calculation. Both are then fused in a `scattering_pack` to form the complete scattering model and to ensure consistency of the rate and state transition calculations as can be seen in the following part of code:

```
template<scattering_rate_type, transition_type, parameter_type>
struct scattering_pack
{
 scattering_rate_type<parameter_type>  rate_calculation;
 transition_type<parameter_type>  state_transition;

 scattering_pack  (const parameter_type& parameters) :
  rate_calculation(parameters), state_transition(parameters)
 {}

 template<typename StateType>
 numeric_type rate(const StateType& state) const
 {
  return rate_calculation(state);
 }
```

```
 template<typename StateType>
 void transition(StateType& state)
 {
  state_transition(state);
 }
}
```

The blend of run time and compile time mechanisms allows the storage of all scattering models within a single container, e.g. `std::vector`, which can be iterated over in order to evaluate them.

```
typedef std::vector<scatter_facade_t<state_type>>
        scatter_container_type ;

scatter_container_type scatter_container ;
scatter_container.push_back(scattering_model) ;
```

For the development of new collision models easy extendability, even without recompilations, is also a highly important issue. This approach allows the addition of scattering models at run time and to expose an interface to an interpreted language such as, e.g., Python [29].

In case a highly optimized version is desired, the run time container (here the `std::vector`) may be exchanged by a compile time container, which is also readily available from the GSSE and provides the compiler with further opportunities for optimizations at the expense of run time adaptability.

While the described approach initially slightly increases the burden of implementation, due to the fact that wrappers must be provided, it gives a transition path to integrate legacy codes into an up to date frame while at the same time not abandoning the experience associated with it. The invested effort allows to raise the level of abstraction, which in turn allows to increase the benefits obtained from the advances in compiler technologies. This in turn inherently allows an optimization for several platforms without the need for massive human effort, which was needed in previous approaches.

In this particular case, encapsulating the reliance on global variables of the functions implementing the scattering models to the wrapping structures, parallelization efforts are greatly facilitated, which are increasingly important with the continued increase of computing cores per CPU.

Furthermore the results can easily be verified as code parts are gradually moved to newer implementations, the only stringent requirement being link compatibility with C++. This test and verification can be taken a step further in case the original implementation is written in ANSI C, due to the high compatibility of it to C++. It is possible to weave parts of the new implementation into the older code. Providing the opportunity to get a comparison not only of final results, but of all the intermediates as well.

Such swift verification of implementations allows to also speed up the steps necessary to verify calculated results with subsequent or contemporary experiments, which should not be neglected, in order to keep physical models and their numerical representations strongly rooted in reality.

The correct operation of the run time concepts depends on the faultless operation of the compiler being used within the rules set down by the C++ language specification [133]. Under this crucial assumption the correct compilation of the code ensures that the code will behave according to the guarantees introduced by the run time concept. While this provides an indication for correctness it cannot be considered absolute proof, especially since the C++ language and its compilers does not have a special focus on validation as is the case in, e.g., Ada[134][135], which could certainly make a stronger case.

7.3 Among Quanta

Where previous comments regarding implementations dealt with classical settings, the following will touch upon the subject of the intangible world of quantum mechanics, which was ever so briefly discussed in Section 5.7. In contrast to the classical description the quantum world allows non-local coherent phenomena. The influence of these effects usually diminishes, when going beyond small scales of time and space, until finally, the macroscopic world appears in its classical beauty.

7.3.1 A Coherent Phase Space Illustration

The setting of quantum mechanics shall serve as an example of the application of the presented abstractions of implementation. The case of the simple infinite square quantum well, illustrated in Figure 7.2, is well suited to provide an expressive example. Using Schrödinger's equation (Equation 5.63) the solution to the infinite square well is found to read

$$\Psi(q) = \sqrt{\frac{2}{L}} \sin\left(n\frac{2\pi}{L}\left(q + \frac{L}{2}\right)\right). \tag{7.29}$$

Subjecting this analytic expression to the Wigner transformation given in Equation 5.71, again

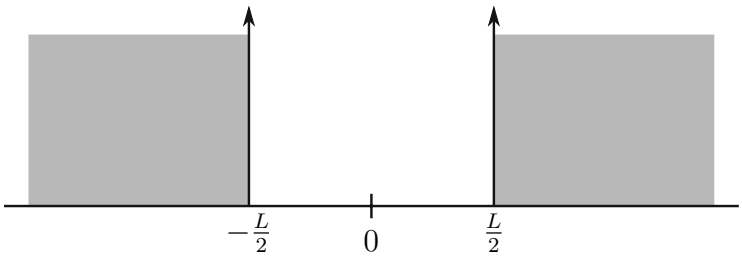


Figure 7.2: Infinite square quantum well.

yields an analytic expression for the Wigner function f_{W} defined on the phase space, which takes the shape

$$f_{\mathrm{W}}(q,p) = \frac{1}{\pi\hbar L}\left(\frac{2\cos\left(\frac{2\pi}{L}nq\right)\sin\left(2\frac{p}{\hbar}\left(\frac{L}{2}-|q|\right)\right)}{2\frac{p}{\hbar}} + \frac{\sin\left(\left(2\frac{p}{\hbar}+\frac{2\pi}{L}n\right)\left(\frac{L}{2}-|q|\right)\right)}{2\frac{p}{\hbar}+\frac{2\pi}{L}n} + \frac{\sin\left(\left(2\frac{p}{\hbar}-\frac{2\pi}{L}n\right)\left(\frac{L}{2}-|q|\right)\right)}{2\frac{p}{\hbar}-\frac{2\pi}{L}n}\right). \tag{7.30}$$

As this is a phase space illustration the obtained representation now depends not only on the spatial location $\mathbf{q}$, but also on a momentum $\mathbf{p}$ as is consistent with the outlines provided in Section 5.7 and originally in Section 5.3. This expression can easily be evaluated in a point wise manner, to yield a scalar field (Definition 61) defined on the phase space. The evaluation in the discrete setting of the digital computer proceeds by iterating over all points, which represent the phase space and evaluating an expression encapsulating the prescription corresponding to Equation 7.30.

This procedure is illustrated in the following snippet of code which uses the traversal capabilities provided by the GSSE as well as functional mechanisms in the spirit of Boost Phoenix [34]. It should be noted that the described procedure is fit to be used with structured grids as well as unstructured meshes, since topology based traversal is handled by the GSSE. Furthermore, it is also important to note that the traversal of the phase space nodes is completely independent of the dimension of the phase space. The functional expression used to compute the value of the scalar field needs to be able to deal with the dimensionality it is provided with. In this particular case the analytical prescription to obtain the phase space description corresponding to a particle in the infinite square quantum well can be extended to, in theory, arbitrary dimensions, albeit visualization of the resulting phase spaces is no longer easily possible.

```
auto const cwf = calc_wigner_function<numeric_t>(L, order);

gsse::traverse()
[
  std::cout << cwf( gsse::acc0 , gsse::acc1 ) << std::endl
](phase_space);
```

The results obtained in this fashion are illustrated in Figure 7.3 and Figure 7.4 for different values of n, corresponding to different Eigenstates.

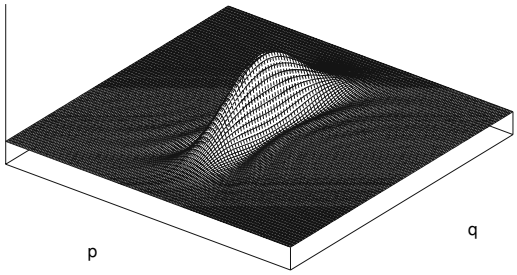


Figure 7.3: Wigner function in the phase space corresponding to the first level ($n = 1$) in an infinite square quantum well.

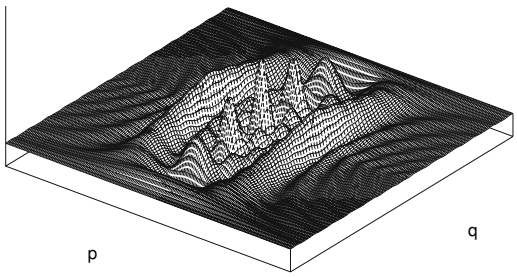


Figure 7.4: Wigner function in the phase space corresponding to the seventh level ($n = 7$) in an infinite square quantum well.

7.3.2 Phase Space Capturing Quantum Effects

The bound states of a contained particle already are profound expression of quantum behaviour. Quantum mechanics using wave functions, however, allows for additional effects. The following example shall illustrate that such effects also find their expression, when using the phase space description. To this end a particle which is represented by a wave packet of the following form

$$\Psi(q) = \left(\frac{1}{2\pi\Delta q^2}\right)^{1/4} \mathrm{e}^{-\frac{(q-q_0)^2}{4\Delta q^2}} \mathrm{e}^{\frac{\mathrm{i}}{\hbar}p_0 q}, \tag{7.31}$$

where q_0 and p_0 are the central position and the mean momentum respectively and Δq represents the width of the wave packet, shall be considered. Subjecting Equation 7.31 to the Wigner transform (Equation 5.71) readily yields the phase space representation which can be given analytically as

$$f_{\mathrm{W}}(q,p) = \frac{1}{\pi\hbar}\mathrm{e}^{-\frac{(q-q_0)^2}{2\Delta q^2}} \mathrm{e}^{-\frac{(p-p_0)^2}{2\Delta p^2}}. \tag{7.32}$$

Again using the traversal capabilities of the GSSE, the evaluation of this expression on the phase space is identical to the previous case with the exception of supplying an appropriate functor in accordance with Equation 7.32. Figure 7.5 provides an illustration of this wave packet in phase space, thus depending on both location $\mathbf{q}$ and momentum $\mathbf{p}$, centred at the origin.

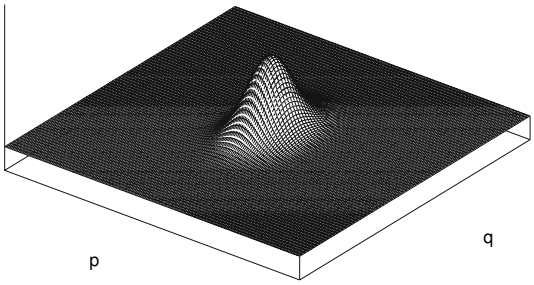


Figure 7.5: Phase space representation of a wave packet.

Moving beyond a single particle and thus a single wave packet, two particles in proximity to one another are considered. Proceeding by simply adding the two pseudo density functions resulting from the wave packets, a result as in Figure 7.6 is obtained, which corresponds to classical expectations regarding the interactions of particles.

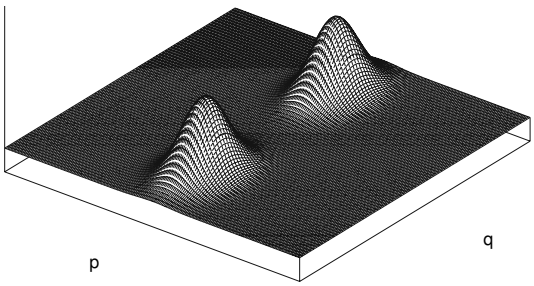


Figure 7.6: Two distinct wave packets placed symmetrically around the origin.

The quantum mechanical setting, however, also allows for a different case, which can be explored by adding the wave functions comprising the two individual wave packets before applying the transformation to phase space. Proceeding in this manner yields the expression

$$f_{\mathrm{W}}(q,p) = \frac{1}{\pi\hbar\left(1+\mathrm{e}^{-\frac{q_0^2}{\Delta q^2}}\right)}\left(\mathrm{e}^{-\frac{(q+q_0)^2}{2\Delta q^2}} + \mathrm{e}^{-\frac{(q-q_0)^2}{2\Delta q^2}} + 2\mathrm{e}^{-\frac{q^2}{2\Delta q^2}}\cos\left(\frac{2q_0}{\hbar}p\right)\right)\mathrm{e}^{-\frac{(p-p_0)^2}{2\Delta p^2}}, \tag{7.33}$$

which, encompasses additional features, as can be seen in Figure 7.7 which has again been obtained using the traversal scheme. Markedly different characteristics when compared to Figure 7.6 are clearly observable in between the two wave packets, which represent a feature of quantum mechanics.

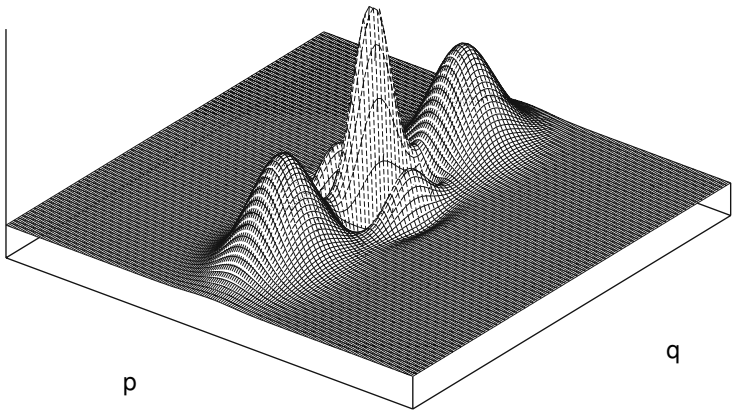


Figure 7.7: Two wave packets interacting.

7.4 Bordering Two Worlds Again

While both of the extremes, the quantum setting as well as the classical world in their pure forms are well covered by powerful descriptions with respect to both, theory as well as application, the realm between them remains problematic, since both descriptions are driven to extremes of their validity.

In an attempt to bridge the gap between the two worlds in the setting of solid state physics, the following scheme has been developed which calculates an estimate for the modification of a coherent solution as obtained by quantum mechanics, under the influence of classical scattering.

The following primarily uses the wave vector $\mathbf{k}$ instead of the momentum $\mathbf{p}$ in the descriptions, as is customary in solid state physics. Furthermore, it may be beneficial to forgo the use of the full structure as presented in Equation 5.75 in Section 5.7 and decompose the description in the following manner

$$\left(\frac{\partial}{\partial t} + \frac{\hbar k_x}{m}\frac{\partial}{\partial x}\right) f_{\mathrm{w}}(x,\mathbf{k}) - \int V_{\mathrm{w}}(x,{k_x}' - k_x) f_{\mathrm{w}}(x,k_x',k_{yz})\mathrm{d}{k_x}' = 0, \tag{7.34}$$

where V_{w} is called the Wigner potential [99][136]. This is the basis for further simplification. Under the assumption that V_{w}locally varies at most linearly, a classical limit for this expression can be recovered [137] in the form

$$\int V_{\mathrm{w}}(x,{k_x}' - k_x) f_{\mathrm{w}}(x,k_x',k_{yz})\mathrm{d}{k_x}' = -\frac{\mathrm{q}}{\hbar}\mathbf{E}\frac{\partial}{\partial k_x} f_{\mathrm{w}}(x,k_x,k_{yz}), \tag{7.35}$$

where q is the charge and $\mathbf{E}$ the electric field. Furthermore, the considerations are given here for one spatial dimension. The remaining two dimensions are assumed to be in equilibrium thus imposing a form for the coherent Wigner function f_{w}^{c} of

$$f_{\mathrm{w}}^{c}(x,\mathbf{k}) = f_{\mathrm{w}}^{c}(x,k_x)\frac{\hbar^2}{2\pi\mathrm{mkT}}\mathrm{e}^{-\frac{\hbar^2\left(k_y'+k_z'\right)}{2\mathrm{mkT}}} \tag{7.36}$$

using Maxwell-Boltzmann statistics, reminiscent of the Equilibrium provided in Equation 7.1. The same procedure is applicable to more than one dimensions, the requirement of a quantum mechanical solution, which is not easily available due to excessive numerical costs of computing a full density matrix required as input, make it difficult in reality.

7.4.1 Wigner Boltzmann Equation

The Wigner Boltzmann equation [138] is obtained by combining elements of Equation 7.34 describing coherent transport and Boltzmann's equation, as given in Equation 5.38 which includes scattering. The combination of the scattering operator and the coherent quantum expression in one dimension can be given as

$$\begin{aligned}\xi\frac{\partial}{\partial x} f_{\mathrm{w}}(x,\mathbf{k}) = &\int V_{\mathrm{w}}(x,k_x' - k_x) f_{\mathrm{w}}(x,k_x',k_{yz})\mathrm{d}k_x' + \\ &\int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}\mathrm{d}k_x' - \lambda(\mathbf{k}) f_{\mathrm{w}}(x,\mathbf{k}),\end{aligned} \tag{7.37}$$

where $\mathbf{k}$ is specifically decomposed into the components k_x and k_{yz}.

The deviation f_{w}^{Δ} of the true Wigner distribution f_{w} from the purely coherent $f_{\mathrm{w}}^{\mathrm{c}}$ is expressed as

$$f_{\mathrm{w}}^{\Delta}(x,\mathbf{k}) = f_{\mathrm{w}}(x,\mathbf{k}) - f_{\mathrm{w}}^{c}(x,k_x). \tag{7.38}$$

This correction to the coherent case then satisfies the following equation

$$\begin{aligned}\xi\frac{\partial}{\partial x} f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}) = &\int V_{\mathrm{w}}(\mathbf{x},\mathbf{k}'-\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' \\ &+ \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' - \lambda(\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}) \\ &+ \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' - \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}),\end{aligned} \tag{7.39}$$

which is treatable in the same fashion as Boltzmann's equation in Section 5.6 to obtain an integral representation. To this end the equation is rewritten to read

$$\begin{aligned}\left[\xi\frac{\partial}{\partial x} + \lambda(\mathbf{k})\right] f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}) = &\int V_{\mathrm{w}}(\mathbf{x},\mathbf{k}'-\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' \\ &+ \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' \\ &+ \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}' - \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}),\end{aligned} \tag{7.40}$$

which again admits an integrating factor of the form $e^{-\int \lambda(\tau)\mathrm{d}\tau}$. Thus the integral form of the correction equation becomes

$$\begin{aligned}\left[f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k})\,\mathrm{e}^{-\int_t^0 \lambda(\tau)\mathrm{d}\tau}\right]_{t_b}^{0} = &\int_{[t_b,0]} \int V_{\mathrm{w}}(\mathbf{x},\mathbf{k}'-\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}'\mathrm{e}^{-\int_t^0 \lambda(\tau)\mathrm{d}\tau}\mathrm{d}t \\ &+ \int_{[t_b,0]} \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}'\mathrm{e}^{-\int_t^0 \lambda(\tau)\mathrm{d}\tau}\mathrm{d}t \\ &+ \int_{[t_b,0]} \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}'\mathrm{e}^{-\int_t^0 \lambda(\tau)\mathrm{d}\tau}\mathrm{d}t \\ &- \int_{[t_b,0]} \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k})\mathrm{e}^{-\int_t^0 \lambda(\tau)\mathrm{d}\tau}\mathrm{d}t.\end{aligned} \tag{7.41}$$

A further reduction can be applied to this expression by using the assumption that the zeroth order of correction already provides a reasonably good approximation. This assumption shall be checked with the implementation, given in Figure 7.11. Thus, foregoing the terms including f_{w}^{Δ} on the right hand side, which corresponds to keeping only the zeroth order term, the expression reduces to contain only terms including f_{w}^{c} which is assumed to be known from coherent calculations.

$$\begin{aligned}f_{\mathrm{w}}^{\Delta}(\mathbf{x},\mathbf{k}) = &\int_{[t_b,0]} \int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k}')\mathrm{d}\mathbf{k}'\mathrm{e}^{-\lambda(\mathbf{k})(-t)}\mathrm{d}t \\ &- \int_{[t_b,0]} \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(\mathbf{x},\mathbf{k})\mathrm{e}^{-\lambda(\mathbf{k})(-t)}\mathrm{d}t\end{aligned} \tag{7.42}$$

The restriction to one dimension of this expression is obtained by integrating over k_{yz}, thus giving

$$f_{\mathrm{w}}^{\Delta}(x,k_x) = \int_{[t_b,0]} \int\int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(x,\mathbf{k}') \mathrm{d}\mathbf{k}' \mathrm{e}^{-\lambda(\mathbf{k})(-t)} \mathrm{d}k_{yz}\mathrm{d}t \\ - \int_{[t_b,0]} \int \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(x,\mathbf{k}) \mathrm{e}^{-\lambda(\mathbf{k})(-t)} \mathrm{d}k_{yz}\mathrm{d}t. \tag{7.43}$$

Inverting the passage of the parameter in the trajectories, results in a description of the form

$$f_{\mathrm{w}}^{\Delta}(x,k_x) = \int_{[0,|t_b|]} \int\int S(\mathbf{k}',\mathbf{k}) f_{\mathrm{w}}^{c}(x-v_x t,\mathbf{k}') \mathrm{d}\mathbf{k}' \mathrm{e}^{-\lambda(\mathbf{k})t} \mathrm{d}k_{yz}\mathrm{d}t \\ - \int_{[0,|t_b|]} \int \lambda(\mathbf{k}) f_{\mathrm{w}}^{c}(x-v_x t,\mathbf{k}) \mathrm{e}^{-\lambda(\mathbf{k})t} \mathrm{d}k_{yz}\mathrm{d}t, \tag{7.44}$$

which will be used for further derivations from this point forward. Additionally t_b will be set to correspond to a positive value of the parameter without further particular indication thereof.

Rearranging and enhancing the last expression to extract representations identifiable as probabilities as was done for the derivations following Equation 7.10 gives

$$f_{\mathrm{w}}^{\Delta}(x,k_x) = \int_{[0,t_b]} \int\int \left\{ \frac{S(\mathbf{k}',\mathbf{k})}{\bar{\lambda}(\mathbf{k}')} \right\} \left\{ \lambda(\mathbf{k}) \mathrm{e}^{-\lambda(\mathbf{k})t} \right\} f_{\mathrm{w}}^{c}(x-v_x t,\mathbf{k}') \mathrm{d}\mathbf{k}' \mathrm{d}k_{yz}\mathrm{d}t \\ - \int_{[0,t_b]} \int \left\{ \lambda(\mathbf{k}) \mathrm{e}^{-\lambda(\mathbf{k})t} \right\} f_{\mathrm{w}}^{c}(x-v_x t,\mathbf{k}) \mathrm{d}k_{yz}\mathrm{d}t, \tag{7.45}$$

which again employs the same basic components recognizable as probabilities as in Section 7.2. Thus although the algorithm is different, the components for trajectories (Definition 45) and scattering mechanisms, which fitted the backward Monte Carlo algorithm can again be used for calculations. Each of the integrals is evaluated separately. The Monte Carlo procedure for the first integral is summarized as follows:

- Initialize an electron state at a selected point of the discretized simulation domain.
- Evaluate the scattering expression for the chosen state.
- Follow the trajectory initialized from the after scattering state to the next scattering event.
- Add the value of f_{w}^{c} to an estimator corresponding to the point at the end of the trajectory.

The second integral term lacks the expression for the scattering probability, thus the algorithm can be given as:

- Initialize an electron state at a selected point of the discretized simulation domain.
- Follow the trajectory initialized from the after scattering state to the next scattering event.
- Add the value of f_{w}^{c} to an estimator corresponding to the point at the end of the trajectory.

The procedures for both integrals are performed using a subdivision of phase space into a structured grid, which defines the topology, as discussed in Section 6.3, and an estimator is associated with every grid node. This grid is then traversed and the two outlined algorithms are applied starting at every grid node, thus sampling phase space. The initialization of the electron states at every point has to take into account the decomposition into k_x and k_{yz} components. Therefore, the yz components are initialized assuming equilibrium, while the x component is initialized corresponding to the phase space position indicated by the grid. The grid is also used to hold parameters for the trajectories, thus making local modelling available, as described in Section 6.5. The local scattering mechanisms can reuse the know how of already existing codes, either directly, by extending the old code base, or by applying the methodology described in Section 7.2.3 to facilitate incorporation into a new frame work.

7.4.2 Approximation to Quantum Effects in Semiconductor Devices

The derived correction is applied to a resonant tunnelling diode (RTD), as it combines quantum mechanical features with classical influence due to scattering. The resonant effect is clearly due to quantum phenomena, yet the dimensions of the device may be such that scattering will disturb any purely coherent solution.

The structure of the device is given Figure 7.8, where the quantum well at the centre of the device is 4nm wide. The barriers have a thickness of 1nm and a height of 3eV. The active device is surrounded by several microns of contact region on both sides, where the distribution of charge carriers is undisturbed by the operation or structure of the device. This apparently inert region of the device has proved itself to be very important for the accuracy of the calculations due to the presented algorithm as is demonstrated in the later Figure 7.13.

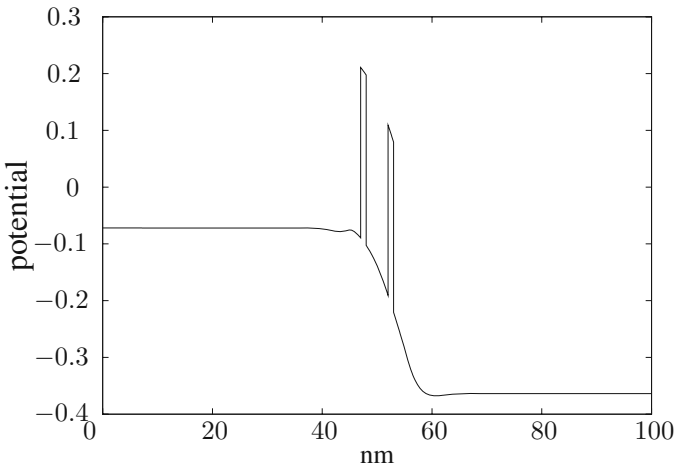


Figure 7.8: Potential profile of the simulated RTD. The 4nm wide quantum well is surrounded by 1nm barriers with a height of 0.3eV, applied is 0.3V bias.

The derivations critically depend on data corresponding to the coherent quantum case represented by f_{w}^{c}. As outlined in Section 5.7 it can be obtained from a density matrix representation by means of transform as shown in Equation 5.71. This density matrix can be efficiently provided using a Non-Equilibrium Green's Function (NEGF) [139][140][141] procedure. After the input has been subjected to the transformation, thus resulting coherent Wigner function representation serves as direct input for the Monte Carlo algorithm. An example of the Wigner representation corresponding to an RTD device is depicted in Figure 7.9.

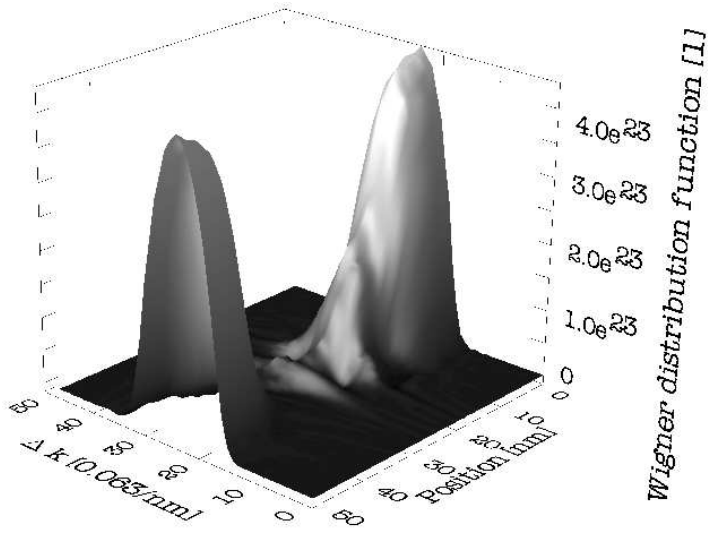


Figure 7.9: The coherent Wigner function f_{w}^{c} obtained from transforming the coherent density matrix provided by a NEGF procedure.

The stochastic nature of the Monte Carlo approach makes verification and debugging of implementations tedious. Therefore, simple test cases are of great importance to quickly detect and eliminate anomalies. One such possibility presents itself, when recording the length of trajectories, since every point is used to generate a trajectory which is either absorbed at the boundary or terminated by a scattering event, it is expected that trajectories will be shorter in the boundary regions. An investigation using histograms, shows that this is indeed the case as shown in Figure 7.10.

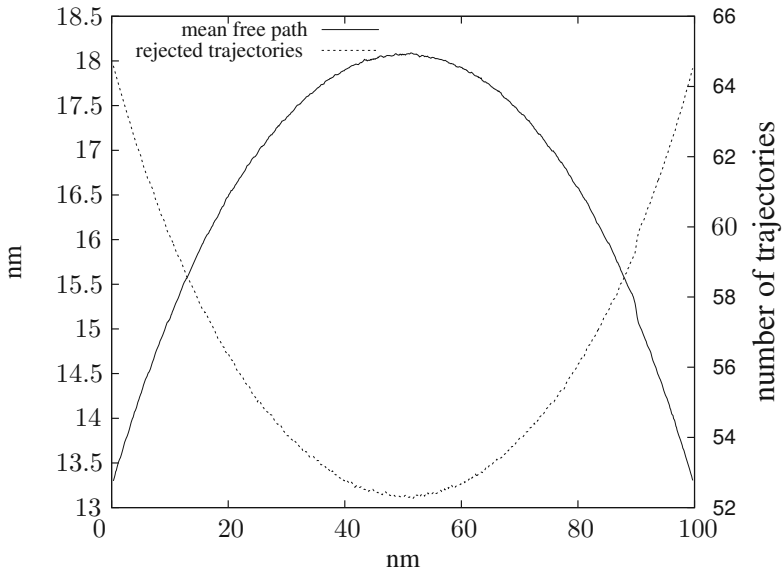


Figure 7.10: Histogram of the trajectories which leave the device and mean free path for trajectories which end in the device.

Furthermore, the assumption that the correction is mainly influenced by the first scattering event can be gauged, by continuing the evolution past the first collision and continuing the trajectory, until it reaches a boundary.

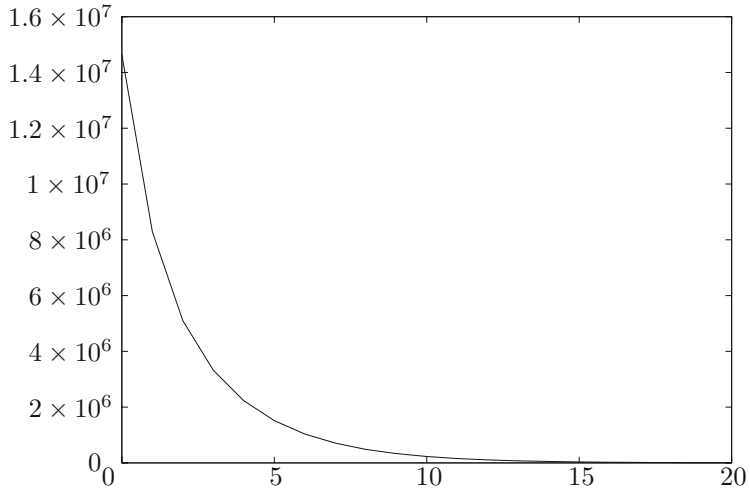


Figure 7.11: Number of trajectories still within the device versus the number of scattering events.

Figure 7.11 shows a steep decline in the number of trajectories still continuing within the domain of interest as the number of scattering events increases, corresponding to approximately an order of magnitude per scattering event.

Finally, having established a means to estimate the validity of the assumption of considering only the first order terms the procedure can be applied and the results are presented in Figure 7.12.

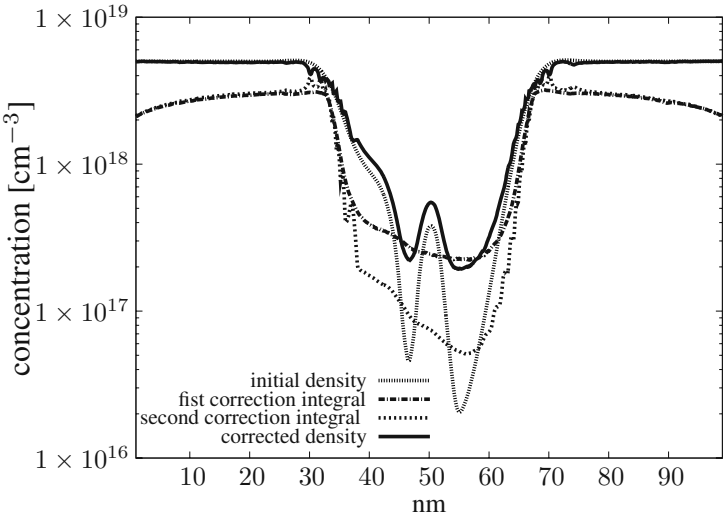


Figure 7.12: Density distribution in the device under $0.3V$ bias, presented in logarithmic scale.

The initial concentration, as obtained calculated from the density matrix is adapted by the scattering, increasing the concentration in the quantum well. It should be pointed out, however that this calculation is not self consistent that is the electric field is kept constant throughout the simulation and not updated due to the shift of charge carriers. The procedure can be refined to include

such updates, by using the newly obtained distribution to calculate a new electric field and also calculating a new density matrix. This new data can then be reapplied to the same procedure. The importance of adhering to the assumptions used in the theoretical derivations of the Monte Carlo procedure is illustrated in Figure 7.13.

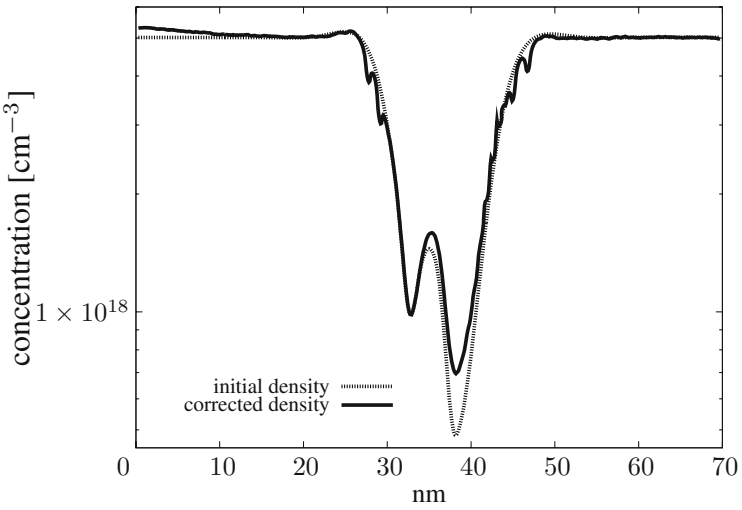


Figure 7.13: An RTD with too short contact regions results in erroneous results at the contacts.

The derivation assumes that the contact regions are in equilibrium, which is not met precisely in the input for the calculation depicted in Figure 7.13. The discrepancy is not visible directly in the given figure, as there is seemingly no disturbance due to the flatness of the contacts. However, applying the derived algorithm reveals that curiously the correction increases in the left contact, which by initial assumption should remain its flat equilibrium configuration. Examining the boundary conditions, revealed a deviation from equilibrium. When correcting this irregularity by extending the bounding contact regions, the results shown in Figure 7.12 are obtained.

Chapter 8

Outlook

dum spiro spero

Marcus Tullius Cicero

Abstractions in the theoretical and practical aspects of sciences and engineering have been developed for extensive periods of time. The comparatively young field of scientific computing has also made progress in attempting to ascend the ladder of developing abstractions. In contrast to the field of mathematics, where abstractions do not result in overhead, the field of scientific computing needs to take care not to incur penalties with respect to limited resources of both memory or computational power due to abstractions. Methodologies, such as static polymorphism have been developed, which together with the ongoing refinement of tools are not only capable of avoiding abstraction penalties, but are even capable of outperforming low level implementations.

The increased abstraction levels also ease to implement higher mathematical abstractions. The C++ programming language, while being far from perfect, offers capabilities to not only establish the desired high abstraction levels at high execution performance. Furthermore it allows the almost seamless reuse of legacy implementations within an abstracted environment, thereby speeding development as well as improving migration capabilities. The reuse of already established pieces of software is not restricted to previous C++ implementations, but by providing a thin wrapping layer extensible to all implementations in languages which can produce link compatible object code. Thus enabling the development of unifying interfaces, which, while abstract, can be flexibly tailored to accommodate new use cases quickly. The high abstraction aims for the employment of concepts in programming. While concepts have not yet been formalized and included into the newest C++ standard, the idea of a concept may already be utilized using the available language features, thereby also highlighting the requirements which are expected from a standardized form of concepts.

It has been attempted to present that an abstract approach to scientific problems is nothing, which has to be newly developed for scientific computing. Mathematics has been developed and used by scientists in order to provide structuring not only as a goal onto itself but also as a tool for other sciences such as physics. It is the aim to show the feasibility of utilizing mathematical abstractions as guidelines for realizations of physical problems in the domain of scientific computing. This should not, however, be a mapping expressing mere existence, for this is clearly beyond doubt as it has already been established on numerous occasions, but is to allow to facilitate the use of advances in all involved fields of research. Thus it is hoped to have given an idea

how to provide a link of the kind as sketched in Figure 2.3. It is clear that such a link cannot be unique, at least from the current level of knowledge. Therefore, a mathematical compendium has been compiled out of the necessity to establish frames of reference. It also intends to show that once powerful concepts are in place, problems can be expressed and treated very concisely without sacrificing generality.

The field of physics has then been explored to illustrate that even in this discipline, which predates scientific computing, there is no single view to any given problem, but that multiple descriptions are available. While the evolution of the modelling techniques and methodologies has progressed, the old descriptions may still find applications in fitting contexts. It is such a trend which is also envisioned for the field of scientific computing; to embrace new facilities that develop in the form of programming paradigms as well as supporting tools and thus allow for an increasing level of abstraction, sophistication and reuse. This should of course not happen at the cost of completely abandoning the already existing implementations and code bases, as Section 7.2.3 was intended to illustrate, by showing the integration of old and new implementations. In this it is similar to the attempts to bridge the realms of classical mechanics and quantum mechanics, as very briefly explored in Section 7.4. While this section may be short, it nevertheless constitutes scientific progress in its own right, even if everything else should be considered to be without value or trivial, it should not be discarded with the rest, since it surpasses the sum of its parts and not only provides insights into a world between worlds, but does so using a newly derived algorithm.

It is crucially important that it is exactly the, for many reasons, implicit assumptions, which can cause major problems for the development of generic implementations. Therefore, even a reiteration of the structures, which are so easily assumed to be ubiquitous is required in order to be able to make them available explicitly. To make a bold claim: the development of generic implementation is mainly concerned with making implicit things explicit. To this end it is first necessary to be aware of the implicit assumptions in the same manner that top down design is only possible if "the top" is known. If this is considered as overly simple, it is astounding how often it is ignored.

By adding possibilities of explicit specification the number of choices is increased which may not be popular in general, when ever more facilities are expected to work using point and click actions. While this is not an unreasonable attitude in order to cope with the ever increasing number of choices an individual may be faced, it is only the choices which allow to reap the most benefit from scientific computing. Since it is always possible to provide convenience layers, which shield potential users from the grizzly internals, while it is not easily possible to do the same in reverse. It is hoped, that the initial example provided in Section 3.3 as well as the examples provided throughout Chapter 6 were able to establish this notion while at the same time serving as examples how to map mathematical structures to digital computers.

An aim for future developments is to provide a means to apply algorithms beyond their initial field of conception. As Chapter 5 elucidated for example, the described mechanics are in essence a geometric theory, thus geometric algorithms should be applicable, using appropriate generalizations. But the intent is then not to produce just another specialized implementation, for a new particular case, but rather formulate a truly generic version of the algorithm, even if specializations can later be again linked to it transparently. The resulting geometry library should be capable of seamlessly dealing not only affine geometry, which is commonly encountered, but also with Riemannian and symplectic geometries efficiently. Furthermore, extensions to include

even more different geometries such as, e.g., hyperbolic geometries should also be made easy by enforcing an explicit separation of basic concepts. The envisioned setting would in itself form a foundation for employing algorithms by providing required infrastructure, thereby enabling identical methods to be applied for, e.g., the minimization of action and shortest path calculation. It would be an important step of making the abstract concepts of differential geometry much more tangible in the field of scientific computing.

These concluding remarks had the intent of once more illuminating the work, the ideas, and the concepts presented in all of the preceding pages, while at the same time providing directions for further questing. If any of the goals have been successful, or all efforts of writing as well as reading have been for naught, is, in the end, up to the reader ...

Appendix A

Resolvent for Integral Equations

While the computation of integrals in itself is a worthy application of Monte Carlo theory, especially in the case of high dimensionality of the integration domain, it can be employed to obtain solutions, or at least estimates, to integral equations.

Recalling that Fredholm integral equations of the second kind

$$\phi(s) = f(s) + \lambda \int_B K(s,t)\phi(t)\mathrm{d}t \tag{A.1}$$

have a solution of the form

$$\phi(s) = f(s) + \lambda \int_B R(s,t,\lambda)f(t)\mathrm{d}t = f(s) + \lambda \sum_{i=0}^{\infty} \int_B K_{i+1}(s,t)\lambda^i f(t)\mathrm{d}t. \tag{A.2}$$

as outlined in Section 4.9. This resolvent series includes repeated integrals which are accessible to Monte Carlo integration methods.

Beginning with examining the integrals in the resolvent series and rewriting them as expectation values gives

$$\lambda \int_B R(s,t,\lambda)f(t)\mathrm{d}t = \lambda \int_B p(t)\frac{R(s,t,\lambda)f(t)}{p(t)}\mathrm{d}t \tag{A.3}$$

$$= \lambda \int_B p(t)\frac{\sum_{i=0}^{\infty} K_{i+1}(s,t)\lambda^i f(t)}{p(t)}\mathrm{d}t \tag{A.4}$$

$$= \lambda \int_B p(t)\sum_{i=0}^{\infty}\frac{K_{i+1}(s,t)}{p(t)}\lambda^i f(t)\mathrm{d}t. \tag{A.5}$$

Examining the terms in order of increasing summation index shows

$$\frac{K_1(s,t)}{p(s,t)} + \frac{K_2(s,t)}{p(s,t)}\lambda + \ldots \tag{A.6}$$

Recalling Equation 4.184 gives

$$K_n(s,t) = \int_B K_{n-1}(s,t_1)K(t_1,t)\mathrm{d}t, \qquad K_1(s,t) = K(s,t), \tag{A.7}$$

where further integrals appear, when expanding K_2

$$\frac{K_2(s,t)}{p(t)}\lambda = \frac{1}{p(t)}\int_B K(s,t_1)K(t_1,t)\mathrm{d}t_1\lambda, \tag{A.8}$$

which can again be evaluated in a similar fashion as an expectation value (Definition 101) with a random variable (Definition 98).

$$\frac{K_2(s,t)}{p(s,t)} = \frac{1}{p(t)}\int_B p(t_1)\frac{K(s,t_1)K(t_1,t)}{p(t_1)}\mathrm{d}t_1\lambda \tag{A.9a}$$

$$= \int_B p(t_1)\frac{K(s,t_1)K(t_1,t)}{p(s,t)p(t,t_1)}\mathrm{d}t_1\lambda \tag{A.9b}$$

The recursion relation A.7 allows to reuse the obtained result in the subsequent terms and further recursion reveals that the term for $K_n(s,t)$ evaluated using Monte Carlo methods (see Section 6.6.2) takes the shape

$$K_n(s,t) = \int_{B^n} \frac{K(s,t_1)}{p(s,t_1)}\left(\prod_{i=1}^{n-2}\frac{K(t_i,t_{i+1})}{p(t_i,t_{i+1})}\right)\frac{K(t_{n-1},t)}{p(t_{n-1},t)}\mathrm{d}t_1\ldots\mathrm{d}t_{n-1} \tag{A.10}$$

Thus every term in the series requires an additional random number to be generated in order to calculate an expectation value. The notion of the procedure is depicted in Figure A.1.

Figure A.1: The general Monte Carlo scheme due to Equation A.10.

Appendix B

Random Number Generation

> Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.
>
> Johann von Neumann

While it is mathematically sufficient to state a random number follows a specific probability density distribution function (Definition 100), this needs to be explicitly taken care of in a computing settings. The deterministic nature of digital computers is in stark contrast to any required randomness which therefore has to be introduced in an explicit manner. Specialized hardware devices which generate random numbers using various physical phenomena described by statistical means are among the available solutions to this issue. It is, however, also desirable to have a workable algorithmic solution. The reliance on a deterministic algorithm results in certain deficiencies of the generated numbers, as the algorithmic description is directly opposed to the concept of randomness. It thus also becomes necessary to consider the exact requirements the generated random numbers have to meet. This of course depends on the field of application, such as Monte Carlo methods or cryptography. In the context of simulations, the efficient, high performance generation of random numbers is as much a concern as the knowledge of the deficiencies. The exact reproducibility of a pseudo random sequence, which constitutes a mortal sin in the field of cryptography [142][143], is not only perfectly acceptable in the context of Monte Carlo methods, where it may actually be considered a boon, as it eases the development and debugging of algorithms.

The pseudo random sequences generated in an algorithmic fashion experience a period after which the sequence repeats itself. Applications utilizing any such sequence need to be aware of this limitation and should accomplish their calculations before the period is exhausted [144].

Furthermore, care has to be taken, if multiple pseudo random generators are used in order to avoid erroneous results due to unchecked correlations in the generated sequences. This issue is of increased significance due to the general trend towards parallelization [145].

The generation of sequences of pseudo random numbers depends on the desired probability density distribution function. Since it is impractical to provide a pseudo random generator for any conceivable probability distribution function, a possibility to derive pseudo random sequences

with differing probability density functions from one another is highly desirable.

One way to accomplish this is using the basic expression

$$y = F(x), \tag{B.1}$$

finding an inverse function

$$x = G(y) = F^{-1}(y), \tag{B.2}$$

and using it to calculate the desired probability distribution. This method, however, obviously requires that $F(x)$ to be invertible, which is not only a theoretical restriction, but may also pose computational challenges.

As inversion may not be a viable course of action, different procedures have been developed such as the rejection technique, which is also called the rejection sampling method or Neumann's method. Using an input pseudo random variable x of an available distribution f, it calculates a pseudo random number y of an arbitrary output distribution g. For this task it requires a rejection and acceptance rule $h(x)$, which determines, if a pseudo random number x is viable or not. In case the rule is met, the random number with the desired distribution is calculated using

$$y = g(x). \tag{B.3}$$

The efficiency of such a procedure depends on a fitting choice of the acceptance rule h.

When the desired probability distribution function g is representable in the form

$$g(x) = f(x)\lambda(x), \tag{B.4}$$

where $f(x)$ is the input probability distribution density function connected to a variable x and λ a bounded function

$$\lambda(x) \leq c. \tag{B.5}$$

With these settings the algorithm takes the form:

1. Generate a pseudo random variable x according to f .
2. Generate a pseudo random variable α – usually the distribution is chosen to be uniform .
3. If

$$f(x) > c\alpha, \tag{B.6}$$

accept x as a realization of a random variable distributed according to g. Otherwise, reject x and restart the procedure.

Appendix C

Richardson Extrapolation

Digital computers cannot, in general, evaluate values with infinite precision and thus need to rely on approximations. It is, of course, among the goals of any scheme resulting in such an approximate value to have as little an error as possible. Numerical schemes to reduce this error, such as the Richardson extrapolation presented here, are therefore of great interest.

The Richardson extrapolation scheme assumes that a function f can be expanded around a point of evaluation depending on a parameter h in a form

$$f(h) = f(0) + c_1 h + \frac{1}{2} c_2 h^2 + \dots, \tag{C.1}$$

such that the point of evaluation corresponds to the parameter $h = 0$. Expanding the function at two different values of the parameter h and $\frac{h}{s}$ gives

$$f(h) = f(0) + c_1 h + \frac{1}{2} c_2 h^2 + \dots \tag{C.2a}$$

$$f(\frac{h}{s}) = f(0) + c_1 \frac{1}{s} h + \frac{1}{2} c_2 \frac{1}{s^2} h^2 + \dots. \tag{C.2b}$$

Multiplying Equation by s and subtracting Equation allows to eliminate the terms of order h

$$s f(\frac{h}{s}) - f(h) = (s-1)\, f(0) - \frac{1}{2} c_2 \frac{1}{s^2} h^2 + \dots. \tag{C.3}$$

Reordering this expression shows the approximation for $f(0)$ explicitly as

$$f(0) = \frac{s f(\frac{h}{s}) - f(h)}{s-1} + c_2' h^2 + \dots. \tag{C.4}$$

An approximation which now is of order h^2 instead of the original order h. This procedure can be applied repeatedly to further increase the quality of the approximation.

Applications using digital computers usually utilize factors s corresponding to powers of 2. This allows the following settings

$$R(i,0) = f(\frac{h}{2^i}) \tag{C.5a}$$

$$R(i,j) = \frac{s^j R(i,j-1) - R(i-1,j-1)}{2^i - 1}, \tag{C.5b}$$

which further illustrate the recursive nature of the procedure.

Bibliography

[1] Κλαύδιος Πτολεμαος[1]. *Μαθηματική Σύνταξις*[2].

[2] Nicolaus Copernicus (Nikolaus Kopernikus). *De revolutionibus orbium coelestium*. J. Petreium, Norimbergæ, 1543.

[3] H. A. Lorentz. Versuch einer Theorie der electrischen und optischen Erscheinungen in bewegten Körpern. 1895.

[4] Albert Einstein. Prinzipielles zur allgemeinen Relativitätstheorie. *Annalen der Physik*, 55:241–244, 1918.

[5] Πλάτων[3]. *Πολιτεία*[4].

[6] Willhelm Schickard (1623). Schickardi machina arithmetica. *Kepler biography*, 1718.

[7] Charles Babbage. Observations on the Application of Machinery to the Computation of Mathematical Tables. *Mem. Astron. Soc*, 1:311–314, 1822.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, August 2006.

[9] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ANSI Fortran X3.9-1966*, 1966.

[10] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 1539-1:2010*, 2010.

[11] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 1989.

[12] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 9899:1999*, 1999.

[13] GNU. *GNU Compiler Collection (GCC)*. http://gcc.gnu.org/.

[14] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.

[1] Klaudios Ptolemaos
[2] Mathematike Syntaxis
[3] Platon
[4] Politeia

[15] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An Extended Comparative Study of Language Support for Generic Programming. *J. of Functional Programming*, 17(2):145–205, March 2007.

[16] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic* . North-Holland Pub. Co., Amsterdam, 1958.

[17] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.

[18] J. Backus. Can Programming be Liberated from the Von Neumann Style?: A Functional Style and its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978.

[19] D. R. Musser and A. A. Stepanov. Generic Programming. In *Proc. of the ISSAC'88 on Symb. and Alg. Comp.*, pages 13–25, London, UK, 1988. Springer.

[20] J. C. Dehnert and A. A. Stepanov. Fundamentals of Generic Programming. *Springer series Lecture Notes in Computational Sciene (LNCS)*, pages 1–11, 1998.

[21] T. Geraud and A. Duret-Lutz. Generic Programming Redesign Pattern. In *Proc. of the 5th Conf. on Pattern Lang. of Progr. (EuroPLoP 2000)*, Irsee, Germany, 2000.

[22] G. Dos Reis and J. Jarvi. What is Generic Programming? In *Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf.*, San Diego, CA, USA, October 2005.

[23] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[24] S. Peyton Jones and J. Hughes (Eds.). Haskell 98: A Non-Strict, Purely Functional Language. Technical report, February 1999.

[25] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2005.

[26] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[27] Python Software Foundation. *Python Programming Language*. http://www.python.org/.

[28] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr. Parallel Library-Centric Application Design by a Generic Scientific Simulation Environment. In *Proc. of the POOSC*, Paphos, Cyprus, July 2008.

[29] P. Schwaha, R. Heinzl, F. Stimpfl, and S. Selberherr. Synergies in Scientific Computing by Combining Multi-Paradigmatic Languages for High-Performance Applications. In *Proc. of the POOSC*, Paphos, Cyprus, July 2008.

[30] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, 1985.

[31] T. L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[32] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley, 2004.

[33] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A Comparative Study of Language Support for Generic Programming. In *Proc. of the 18th Annual ACM SIGPLAN*, pages 115–134, New York, NY, USA, 2003. ACM Press.

[34] Boost. *Boost Phoenix 2.0*. http://www.boost.org/libs/spirit/phoenix.

[35] P.J. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[36] M. Zalewski and S. Schupp. Change Impact Analysis for Generic Libraries. In *Proc. of 21st IEEE Intl. Conf. on Software Maintenance*, September 2006.

[37] Douglas Gregor. *High-level Static Analysis for Generic Libraries*. Dissertation, Rensselaer Polytechnic Institute, May 2004.

[38] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, Boston, MA, USA, 1998.

[39] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, MA, USA, 2001.

[40] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. G. Siek. Algorithm Specialization in Generic Programming - Challenges of Constrained Generics in C++. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, New York, NY, USA, June 2006. ACM Press.

[41] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, and G.D. Reis ADN A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. *SIGPLAN Not.*, 41(10):291–310, 2006.

[42] J. Siek, J. G. Siek, and Andrew Lumsdaine. Essential Language Support for Generic Programming. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[43] A. Priesnitz and S. Schupp. From Generic Invocations to Generic Implementations. In *Proc. of the POOSC*, Technical Report, Nantes, France, July 2006.

[44] J. Siek and A. Lumsdaine. Mayfly: A Pattern for Lightweight Generic Interfaces, July 1999.

[45] L. Lee and A. Lumsdaine. Generic Programming for High Performance Scientific Applications. In *JGI '02: Proc. of the 2002 joint ACM-ISCOPE Conf. on Java Grande*, pages 112–121, New York, NY, USA, 2002. ACM Press.

[46] J. Järvi, A. Lumsdaine, D. Gregor, M. Kulkarni, D. Musser, and S. Schupp. Generic Programming and High-Performance Libraries. In *Next Generation Software Program Workshop, Santa Fe*, 2004.

[47] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic Programming and High-Performance Libraries. *Intl. J. of Parallel Prog.*, 33(2), June 2005.

[48] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A High Performance Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, page 61, Umea, Sweden, June 2006.

[49] R. Heinzl, M. Spevak, and P. Schwaha. Concepts for High Performance Generic Scientific Computing. In *Proc. EEICT 2006*, volume 4, pages 446–450, Brno, Czech Rep., April 2006.

[50] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. Concepts for High Performance Generic Scientific Computing. In *Proc. of the 5th MATHMOD*, volume 1, Vienna, Austria, February 2006.

[51] R. Heinzl, P. Schwaha, and S. Selberherr. A High Performance Generic Scientific Simulation Environment. In B. Kaagström et al., editor, *Lecture Notes in Computer Science*, volume 4699/2007, pages 996–1005. Springer, Berlin, June 2007.

[52] M. Schordan, R. Heinzl, and S. Selberherr. Characterization and Performance Evaluation of Generic Programming Styles in C++. In *Library Centric Sofware Design, OOPSLA*, Portland, OR, USA, October 2006.

[53] R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr. A Parallel Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, Trondheim, Norway, May 2008.

[54] F. Putze, P. Sanders, and J. Singler. MCSTL: The Multi-Core Standard Template Library. In *Proc. Symposium on Principles and Practice of Parallel Programming*, pages 144–145, New York, NY, USA, 2007. ACM.

[55] G. Berti. *Generic Software Components for Scientific Computing*. Dissertation, Technische Universität Cottbus, 2000.

[56] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser. A Generic Approach to Scientific Computing. In *Proc. of the ICCAM Conf.*, page 137, Leuven, Belgium, July 2006.

[57] R. Heinzl. *Concepts for Scientific Computing*. Dissertation, Technische Universität Wien, 2007.

[58] Boost. *Boost MPL*. http://www.boost.org.

[59] Erwin H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[60] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proc. of the 19th annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[61] Hans Jörg Dirschmid. *Höhere Mathematik – Matrizen und Lieare Gleichunen*. Manz, 1998.

[62] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[63] 孫武[5]. 孫子兵法[6]. approx. 6th century BC.

[64] Leonhard Euler. *Methodus inveniendi lineas curvas maximi minimive proprietate gaudentes, sive solutio problematis isoperimetrici latissimo sensu accepti*. 1744.

[65] Bernhard Schutz. *Geometrical Methods of Mathematical Physics*. Cambridge University Press, 1980.

[66] Marián Fecko. *Differential Geometry and Lie Groups for Physicists*. Cambridge University Press, 2006.

[67] Oliver Davis Johns. *Analytical Mechanics for Relativity and Quantum Mechanics*. Oxford University Press, 2005.

[68] Jerrold E. Marsden, Ralph Abraham, and Tudor Ratiu. *Manifolds, Tensor Analysis and Applications*. Springer, 3 edition, 2002.

[69] Vladimir I. Arnold. *Mathematical Methods of Classical Mechanics*. Springer, 1989.

[70] Ivan Kolář, Peter W. Michor, and Jan Slovák. *Natural Operations in Differential Geometry*. Springer, 1993.

[71] Serge Lang. *Fundamentals of Differential Geometry*. Springer, 1998.

[72] Hans Jörg Dirschmid. *Tensoren und Felder*. Springer, 199.

[73] *Wikipedia*. http://www.wikipedia.org.

[74] Michiel Hazewinkel. *Encyclopaedia of Mathematics*. Springer, 2002.

[75] R. Heinzl, P. Schwaha, C. Giani, and S. Selberherr. Modeling of Non-Trivial Data-Structures with a Generic Scientific Simulation Environment. In *Proc. of the 4th High-End Visualization Workshop*, pages 5–13, Tyrol, Austria, June 18–22 2007.

[76] David Hestenes. Mathematical Viruses. In *Algebras and their Applications in Mathematical Physics*, pages 3–16. Kluwer, 1991.

[77] Paul Adrien Maurice Dirac. A New Notation for Quantum Mechanics. In *Proceedings of the Cambridge Philosophical Society*, volume 35 of *Proceedings of the Cambridge Philosophical Society*, page 416, 1939.

[78] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[5] Sunzi/Sun Tsu/Sun Tzu/Sun Tse

[6] The Art of War

[79] Josiah Willard Gibbs. *Elementary Principles in Statistical Mechanics*. New York : C. Scribner, 1902.

[80] Werner Karl Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 43(3):172 – 198, 1927.

[81] Ἀρχιμήδης[7]. *De planorum equilibris*. approx. 250 BC.

[82] Ἀριστοτέλης[8]. *Φυσικῆς ἀκροάσεως*[9]. approx. 350 BC.

[83] Isaac Newton. *PhilosophiæNaturalis Principia Mathematica*. 1687.

[84] Gottfried Willhelm Leibniz. *Nova Methodus pro Maximis et Minimis*. 1684.

[85] Pierre-Simon marquis de Laplace. *Exposition du système du monde*. 1796.

[86] Joseph Louis Lagrange. *Mécanique Analytique*. 1788.

[87] N. N. Bogoliubov. Kinetic Equations. *Journal of Experimental and Theoretical Physics*, 16(8):691–702, 1946.

[88] M. Born and H. S. Green. A General Kinetic Theory of Liquids I. The Molecular Distribution Functions. *Proc. Roy. Soc*, 1946.

[89] Melville S. Green. Boltzmann Equation from the Statistical Mechanical Point of View. *Journal of Chemical Physics*, 25, 1956.

[90] John G. Kirkwood. The Statistical Mechanical Theory of Transport Processes II. General Theory. *The Journal of Chemical Physics*, 14, 1946.

[91] J. Yvon. Theorie Statistique des Fluides et l'Equation et l'Equation d'Etat. *Actes sientifique et industrie*, (203), 1935.

[92] Mihail Nedjalkov, Dragica Vasileska, Ivan Dimov, and Goce Arsov. Mixed initial-boundary value problem in particle modeling of microelectronic devices. *Monte Carlo Methods Appl.*, (4):299–331, 2007.

[93] Karl Raimund Popper. *Logik der Forschung*. 1935.

[94] Hugh Everett. "Relative State" Formulation of Quantum Mechanics. *Reviews of Modern Physics*, 29(3):454–462, July 1957.

[95] David Bohm. A Suggested Interpretation of the Quantum Theory in Terms of "Hidden" Variables. I. *Phys. Rev.*, 85(2):166–179, Jan 1952.

[96] David Bohm. A Suggested Interpretation of the Quantum Theory in Terms of "Hidden" Variables. II. *Phys. Rev.*, 85(2):180–193, Jan 1952.

[97] Eugene Wigner and Henry Margenau. Symmetries and Reflections, Scientific Essays. *American Journal of Physics*, 35(12):1169–1170, 1967.

[7] Archimedes
[8] Aristoteles
[9] Physikes akroaseos

[98] Hermann Weyl. Quantenmechanik und Gruppentheorie. *Zeitschrift für Physik A Hadrons and Nuclei*, 46(1 - 2):1–46, November 1927.

[99] Eugene Wigner. On the Quantum Correction For Thermodynamic Equilibrium. *Phys. Rev.*, 40(5):749–759, Jun 1932.

[100] José E. Moyal and Maurice S. Bartlett. Quantum Mechanics as a Statistical Theory. In *Proceedings of the Cambridge Philosophical Society*, volume 45, pages 99–124, 1949.

[101] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3 edition, 1998.

[102] R. Heinzl and P. Schwaha. A Generic Topology Library. *Science of Computer Programming*, 2009.

[103] Boost. *Boost Fusion 2.0*. http://www.boost.org/libs/fusion.

[104] R. Sonderfeld and R. Heinzl. A Generic and Self-Optimizing Polynomial Library. In *Proceedings of the 8th workshop on Parallel/High-Performance*, Genova, Italy, July 2009.

[105] A. J. Zomorodian. Topology for Computing. In *Cambridge Monographs on Applied and Computational Mathematics*, 2005.

[106] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[107] Harvey E. Cline and William E. Lorensen. System and Method for the Display of Surface Structures Contained within the Interior Region of a Solid Body. US Patent no. 4710876, 1985.

[108] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70, 1990.

[109] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[110] Boost. *Boost C++ Libraries*. http://www.boost.org.

[111] August Ferdinand Möbius. *Zur Theorie der Polyëder und der Elementarverwandtschaft*. Oeuvres Compl'tes, 1861.

[112] M. Bern and D. Eppstein. Mesh Generation and Optimal Triangulation. In *Comp. in Eucl. Geometry, Lect. Notes on Comp.*, volume 1, 1992.

[113] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, 2001.

[114] J. R. Shewchuk. What is a Good Linear Element? Interpolation, Conditioning, and Quality Measures. In *Proc. IMR*, pages 115–126, 2002.

[115] Carl Runge. Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik*, 46:224–243, 1901.

[116] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, 1964.

[117] Philip J. Davis and Philip Rabinowitz. *Methods of Numerical Integration*. Academic Press, New York,, 2 edition, 1982.

[118] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.

[119] Stanislaw Ulam, Robert Davis Richtmyer, and Johann von Neumann. Statistical Methods in Neutron Diffusion. *Los Alamos Scientific Laboratory report LAMS–551*, 1947.

[120] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44:335–341, 1949.

[121] Nicholas Metropolis. The Beginning of the Monte Carlo Method. *Los Alamos Science*, (15), 1987.

[122] Ilya M. Sobol. *A Primer for the Monte Carlo Method*. CRC-Press, 1994.

[123] I. Dimov. *Monte Carlo Methods For Applied Scientists*. World Scientific Publishing Company, 2005.

[124] Jun S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer, 2008.

[125] David Hilbert. *Grundzüge einer allgemeinen Theorie der linearen Integralgleichungen*. B. G. Teubner, Leipzig, 1912.

[126] Emmy Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918.

[127] T. Grasser, H. Kosina, M. Gritsch, and S. Selberherr. Using Six Moments of Boltzmann's Transport Equation for Device Simulation. *J. of Applied Physics*, 90(5):2389–2396, 2001.

[128] Robert Kosik. *Numerical Challenges on the Road to NanoTCAD*. Dissertation, Technische Universität Wien, 2004.

[129] Martin Vasicek. *Advanced Macroscopic Transport Models*. Dissertation, Technische Universität Wien, 2009.

[130] D.L. Scharfetter and H.K. Gummel. Large-Signal Analysis of a Silicon Read Diode Oscillator. *IEEE Trans. Electron Dev.*, 16(1):64–77, 1969.

[131] C. Jacoboni, P. Poli, and L Rota. A New Monte Carlo Technique for the Solution of the Boltzmann Transport Equation. *Solid State Electronics*, pages 523–526, 1988.

[132] P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup. Runtime Concepts for the C++ Standard Template Library. In *Proc. of the ACM SAC'08*, pages 171–177, Fortaleza, Ceara, Brazil, March 2008.

[133] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14882:1998*, 1998.

[134] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 8652:1987*, 1987.

[135] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 8652:1995*, 1995.

[136] David K. Ferry and Harold L. Grubin. Modeling of Quantum Transport in Semiconductor Devices. volume 49 of *Solid State Physics*, pages 283 – 448. Academic Press, 1996.

[137] Carlo Jacoboni, Andrea Bertoni, Paolo Bordone, and Rossella Brunetti. Wigner-Function Formulation for Quantum Transport in Semiconductors: Theory and Monte Carlo Approach. *Mathematics and Computers in Simulation*, 55(1–3):67 – 78, 2001.

[138] Mihail Nedjalkov, Hans Kosina, Siegfried Selberherr, Christian Ringhofer, and David K. Ferry. Unified Particle Approach to Wigner-Boltzmann Transport in Small Semiconductor Devices. *Phys. Rev. B*, 70(11):115319, September 2004.

[139] George Green. *An Essay on the Application of Mathematical Analysis to the Theories of Electricity and Magnetism*. Nottingham, 1828.

[140] Lars Hörmander. *The Analysis of Linear Partial Differential Operators : Distribution Theory and Fourier Analysis*, volume 1. Springer, 1990.

[141] Hartmut Haug and Antti-Pekka Jauho. *Quantum Kinetics in Transport and Optics of Semiconductors*. Springer, 2004.

[142] D. Eastlake 3rd, S. Crocker, and J. Schiller. Randomness Recommendations for Security. RFC 1750 (Informational), December 1994. Obsoleted by RFC 4086.

[143] D. Eastlake and 3rd, J. Schiller, and S. Crocker. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[144] Pierre L'Ecuyer and Richard Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[145] Michael Mascagni, David Ceperley, and Ashok Srinivasan. SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

Index

Philipp Schwaha

oleum et operam perdidi

Plautus

1977
Born in Wien, Austria

1996
High school graduation

1996/7
Compulsory civil service

1997
Enrolled in Electrical Engineering at the TU Wien, Austria

2004
Degree of Diplom-Ingenieur, TU Wien, Austria, *Summa Cum Laude*

2004
Research assistant at the Institute for Microelectronics, TU Wien, Austria

2006-2010
Assistant professor (Universitätsassistent) at the Institute for Microelectronics, TU Wien, Austria

[1] Josef Weinbub, Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Siegfried Selberherr. A dispatched covariant type system for numerical applications in C++. In *AIP Conference Proceedings*, pages 1663–1666, 2010. talk: International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Rhodos; 2010-09-19 – 2010-09-25.

[2] Josef Weinbub, Rene Heinzl, Philipp Schwaha, Franz Stimpfl, and Siegfried Selberherr. A lightweight material library for scientific computing in C++. In *Proceedings of the European Simulation and Modelling Conference (ESM)*, pages 454–458, 2010. talk: The European Simulation and Modelling Conference (ESM), Hasselt; 2010-10-25 – 2010-10-27.

[3] Philipp Schwaha and Rene Heinzl. Marching simplices. In *AIP Conference Proceedings*, pages 1651–1654, 1281, 2010. talk: International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Rhodos; 2010-09-19 – 2010-09-25.

[4] Georg Mach, Rene Heinzl, Philipp Schwaha, Franz Stimpfl, Josef Weinbub, and Siegfried Selberherr. A modular tool chain for high performance CFD simulations in intracranial aneurysms. In *AIP Conference Proceedings*, pages 1647–1650, 2010. talk: International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Rhodos; 2010-09-19 – 2010-09-25.

[5] Franz Stimpfl, Josef Weinbub, Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. A unified topological layer for finite element space discretization. In *AIP Conference Proceedings*, pages 1655–1658, 2010. talk: International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Rhodos; 2010-09-19 – 2010-09-25.

[6] Philipp Schwaha, Oskar Baumgartner, Rene Heinzl, Mihail Nedjalkov, Siegfried Selberherr, and Ivan Dimov. Classical approximation of the scattering induced Wigner correction equation. In *13th International Workshop on Computational Electronics*, pages 177–180, 2009. poster presentation: International Workshop on Computational Electronics (IWCE), Beijing; 2009-05-27 – 2009-05-29.

[7] Stanislav Tyaginov, Wolfgang Gös, Tibor Grasser, Viktor Sverdlov, Philipp Schwaha, Rene Heinzl, and Franz Stimpfl. Description of Si-O bond breakage using pair-wise interatomic potentials under consideration of the whole crystal. In *2009 IEEE International Reliability Physics Symposium Proceedings*, pages 514–522, 2009. talk: International Reliability Physics Symposium (IRPS), Montreal; 2009-04-26 – 2009-04-30.

[8] Philipp Schwaha, Rene Heinzl, and Mihail Nedjalkov. The forced evolution of implementations: Using a Monte Carlo algorithm as example. In *Proceedings of the 8th workshop on Parallel/High-Performance*, 2009. talk: European Conference on Object-Oriented Programming, Genova; 2009-07-07.

[9] Rene Heinzl, Philipp Schwaha, Franz Stimpfl, and Siegfried Selberherr. GUIDE: Parallel library-centric application design by a generic scientific simulation environment. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):505–520, 2009.

[10] Stanislav Tyaginov, Viktor Sverdlov, Wolfgang Gös, Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Tibor Grasser. Impact of the surrounding network on the Si-O bond-breakage energetics. In *Proceedings of the 2009 MRS Spring Meeting*, 2009. talk: Materials Research Society Spring Meeting (MRS), San Francisco; 2009-04-13 – 2009-04-17.

[11] Mihail Nedjalkov, Philipp Schwaha, Oskar Baumgartner, and Siegfried Selberherr. Particle model of the scattering-induced Wigner function correction. In *Proceedings of the 7th International Conference on Large-Scale Scientific Computations*, pages 411–418, 2009. talk: International Conference on Large-Scale Scientific Computations (LSSC), Sozopol; 2009-06-04 – 2009-06-08.

[12] Stanislav Tyaginov, Viktor Sverdlov, Wolfgang Gös, Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Tibor Grasser. Si-O bond-breakage energetics under consideration of the whole crystal. In *Proceedings of the International Semiconductor Technology Conference & China Semiconductor Technology International Conference*, page 84, 2009. talk: International Semiconductor Technology Conference & China Semiconductor Technology International Conference, Shanghai; 2009-03-19 – 2009-03-20.

[13] Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Siegfried Selberherr. Synergies in scientific computing by combining multi-paradigmatic languages for high-performance applications. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):539–549, 2009.

[14] Oskar Baumgartner, Philipp Schwaha, Markus Karner, Mihail Nedjalkov, and Siegfried Selberherr. Coupling of non-equilibrium Green's function and Wigner function approaches. In *International Conference on Simulation of Semiconductor Processes and Devices 2008*, pages 345–348, 2008. talk: International Conference on Simulation of Semiconductor Processes and Devices (SISPAD), Hakone; 2008-09-09 – 2008-09-11.

[15] Franz Stimpfl, Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. High performance parallel mesh generation and adaptation. In *Proceedings Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Trondheim; 2008-05-13 – 2008-05-16.

[16] Rene Heinzl, Philipp Schwaha, Franz Stimpfl, and Siegfried Selberherr. A parallel generic scientific simulation environment. In *Proceedings Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Trondheim; 2008-05-13 – 2008-05-16.

[17] Rene Heinzl, Philipp Schwaha, Franz Stimpfl, and Siegfried Selberherr. Parallel library-centric application design by a generic scientific simulation environment. In *7th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'08)*, 2008. talk: Workshop on Parallel Object-Oriented Scientific Computing (POOSC), Paphos; 2008-07-08.

[18] Franz Stimpfl, Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. A robust parallel Delaunay mesh generation approach suitable for three-dimensional TCAD. In *International Conference on Simulation of Semiconductor Processes and Devices 2008*, pages 265–268, 2008. poster presentation: International Conference on Simulation of Semiconductor Processes and Devices (SISPAD), Hakone; 2008-09-09 – 2008-09-11.

[19] Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Siegfried Selberherr. Synergies in scientific computing by combining multi-paradigmatic languages. In *Proceedings Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Trondheim; 2008-05-13 – 2008-05-16.

[20] Philipp Schwaha, Rene Heinzl, Franz Stimpfl, and Siegfried Selberherr. Synergies in scientific computing by combining multi-paradigmatic languages for high-performance applications. In *7th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'08)*, 2008. talk: Workshop on Parallel Object-Oriented Scientific Computing (POOSC), Paphos; 2008-07-08.

[21] Tibor Grasser, Wolfgang Gös, Oliver Triebl, Philipp Paul Hehenberger, Paul-Jürgen Wagner, Philipp Schwaha, Rene Heinzl, Stefan Holzer, Robert Entner, Stephan Wagner, and Franz Schanovsky. 3 year report 2005-2007. Technical report, E360 - Institute for Microelectronics; Vienna University of Technology, 2007.

[22] Michael Spevak, Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. A computational framework for topological operations. In *Lecture Notes in Computer Science*, pages 781–790. Springer, Berlin-Heidelberg, 2007.

[23] Philipp Schwaha, Rene Heinzl, Georg Mach, Claudia Pogoreutz, Susanne Fister, and Siegfried Selberherr. Electro-biological simulation using a web front-end. In *Proceedings European Simulation and Modeling Conference*, pages 493–495, 2007. talk: The European Simulation and Modelling Conference (ESM), Malta; 2007-10-22 – 2007-10-24.

[24] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Siegfried Selberherr. A high performance generic scientific simulation environment. In *Lecture Notes in Computer Science*, pages 996–1005. Springer, Berlin-Heidelberg, 2007.

[25] Philipp Schwaha, Rene Heinzl, Georg Mach, Claudia Pogoreutz, Susanne Fister, and Siegfried Selberherr. A high performance webapplication for an electro-biological problem. In *Proceedings 21st European Conference on Modelling and Simulation*, pages 218–222, 2007. talk: 21st European Conference on Modelling and Simulation, Prag; 2007-06-04 – 2007-06-06.

[26] Rene Heinzl, Georg Mach, Philipp Schwaha, and Siegfried Selberherr. Labtool - a managing software for computer courses. In *Proceedings European Simulation and Modeling Conference*, pages 488–492, 2007. talk: The European Simulation and Modelling Conference (ESM), Malta; 2007-10-24 – 2007-10-27.

[27] Rene Heinzl, Philipp Schwaha, Carlos Giani, and Siegfried Selberherr. Modeling of non-trivial data-structures with a generic scientific simulation environment. In *Proceedings of the 4th High-End Visualization Workshop*, pages 5–13, 2007. talk: High-End Visualization Workshop, Obergurgl; 2007-06-18 – 2007-06-22.

[28] Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. Modern concepts for high-perfomance scientific computing - library centric application design. In *Proceedings of the 2nd ICSOFT 2007*, pages 100–107, 2007. talk: International Conference on Software and Data Technologies (ICSOFT), Barcelona; 2007-07-22 – 2007-07-25.

[29] Franz Stimpfl, Rene Heinzl, Philipp Schwaha, and Siegfried Selberherr. A multi-mode mesh generation approach for scientific computing. In *Proceedings European Simulation and Modeling Conference*, pages 506–513, 2007. talk: The European Simulation and Modelling Conference (ESM), Malta; 2007-10-22 – 2007-10-24.

[30] Rene Heinzl, Georg Mach, Philipp Schwaha, and Siegfried Selberherr. A performance test platform. In *Proceedings European Simulation and Modeling Conference*, pages 483–487, 2007. talk: The European Simulation and Modelling Conference (ESM), Malta; 2007-10-22 – 2007-10-24.

[31] Philipp Schwaha, Markus Schwaha, Rene Heinzl, Stephan Enzo Ungersböck, and Siegfried Selberherr. Simulation methodologies for scientific computing - modern application design. In *Proceedings of the 2nd ICSOFT 2007*, pages 270–276, 2007. talk: International Conference on Software and Data Technologies (ICSOFT), Barcelona; 2007-07-22 – 2007-07-25.

[32] Philipp Schwaha, Carlos Giani, Rene Heinzl, and Siegfried Selberherr. Visualization of polynomials used in series expansions. In *Proceedings of the 4th High-End Visualization Workshop*, pages 139–148, 2007. talk: High-End Visualization Workshop, Obergurgl; 2007-06-18 – 2007-06-22.

[33] Philipp Schwaha, Rene Heinzl, Michael Spevak, and Tibor Grasser. Advanced equation processing for TCAD. In *Proceedings of the PARA Conference*, page 64, 2006. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umea; 2006-06-18 – 2006-06-21.

[34] Michael Spevak, Rene Heinzl, Philipp Schwaha, and Tibor Grasser. Automatic linearization using functional programming for scientific computing. In *ICCAM 2006 Abstracts of Talks*, page 147, 2006. talk: International Congress on Computational and Applied Mathematics (ICCAM), Leuven; 2006-07-10 – 2006-07-14.

[35] Michael Spevak, Rene Heinzl, Philipp Schwaha, and Tibor Grasser. A computational framework for topological operations. In *Proceedings of the PARA Conference*, page 57, 2006. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umea; 2006-06-18 – 2006-06-21.

[36] Rene Heinzl, Philipp Schwaha, Michael Spevak, and Tibor Grasser. Concepts for high performance generic scientific computing. In *5th Mathmod Vienna Proceedings*, pages 4–1–4–9, 2006. talk: International Symposium on Mathematical Modeling (MATHMOD), Wien; 2006-02-08 – 2006-02-10.

[37] Philipp Schwaha, Rene Heinzl, Michael Spevak, and Tibor Grasser. A generic approach to scientific computing. In *ICCAM 2006 Abstracts of Talks*, page 137, 2006. talk: International Congress on Computational and Applied Mathematics (ICCAM), Leuven; 2006-07-10 – 2006-07-14.

[38] Michael Spevak, Rene Heinzl, Philipp Schwaha, Tibor Grasser, and Siegfried Selberherr. A generic discretization library. In *OOPSLA Proceedings*, pages 95–100, 2006. talk: Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Portland; 2006-10-22 – 2006-10-26.

[39] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Tibor Grasser. A generic scientific simulation environment for multidimensional simulation in the area of TCAD. In *NSTI Nanotech Proceedings*, pages 526–529, 2006. poster presentation: The Nanotechnology Conference and Trade Show, Boston; 2006-05-07 – 2006-05-11.

[40] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Siegfried Selberherr. A generic topology library. In *OOPSLA Proceedings*, pages 85–93, 2006. talk: Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Portland; 2006-10-22 – 2006-10-26.

[41] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Tibor Grasser. A high performance generic scientific simulation environment. In *Proceedings of the PARA Conference*, page 61, 2006. talk: Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umea; 2006-06-18 – 2006-06-21.

[42] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Tibor Grasser. High performance process and device simulation with a generic environment. In *Proceedings of the 14th Iranian Conference on Electrical Engineering ICEE 2006*, 2006. talk: Iranian Conference on Electrical Engineering (ICEE), Tehran; 2006-05-16 – 2006-05-18.

[43] Philipp Schwaha, Rene Heinzl, Wolfgang Brezna, Jürgen Smoliner, H. Enichlmair, R. Minixhofer, and Tibor Grasser. Leakage current analysis of a real world silicon-silicon dioxide capacitance. In *Proceedings International Caribbean Conference on Devices, Circuits and Systems*, pages 365–370, 2006. talk: International Caracas Conference on Devices, Circuits and Systems (ICCDCS), Playa del Carmen; 2006-04-26 – 2006-04-28.

[44] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Tibor Grasser. Multidimensional and multitopological TCAD with a generic scientific simulation environment. In *Proceedings International Caribbean Conference on Devices, Circuits and Systems*, pages 173–176, 2006. talk: International Caracas Conference on Devices, Circuits and Systems (ICCDCS), Playa del Carmen; 2006-04-26 – 2006-04-28.

[45] Rene Heinzl, Michael Spevak, and Philipp Schwaha. A novel high performance approach for technology computer aided design. In *Proceedings of the 12th Conference Student EEICT 2006*, pages 446–450, Vol.4, 2006. talk: Student Electrical Engineering, Information and Communication Technologies (STUDENT EEICT), Brünn; 2006-04-27.

[46] Rene Heinzl, Michael Spevak, Philipp Schwaha, Tibor Grasser, and Siegfried Selberherr. Performance analysis for high-precision interconnect simulation. In *The 2006 European Simulation and Modelling Conference*, pages 113–116, 2006. talk: European Simulation and Modeling Conference (ESMC), Toulouse; 2006-10-23 – 2006-10-25.

[47] Rene Heinzl, Philipp Schwaha, Michael Spevak, and Tibor Grasser. Performance aspects of a DSEL for scientific computing with C++. In *Proceedings of the POOSC Conference*, pages 37–41, 2006. talk: Workshop on Parallel Object-Oriented Scientific Computing (POOSC), Nantes; 2006-07-03 – 2006-07-07.

[48] Michael Spevak, Rene Heinzl, Philipp Schwaha, and Tibor Grasser. Process and device simulation with a generic scientific simulation environment. In *Proceedings International Conference on Microelectronics (MIEL)*, pages 475–478, 2006. talk: International Conference on Microelectronics (MIEL), Beograd; 2006-04-14 – 2006-04-17.

[49] Michael Spevak, Rene Heinzl, Philipp Schwaha, and Tibor Grasser. Simulation of microelectronic structures using a posteriori error estimation and mesh optimization. In *5th Mathmod Vienna Proceedings*, pages 5–1–5–8, 2006. talk: International Symposium on Mathematical Modeling (MATHMOD), Wien; 2006-02-08 – 2006-02-10.

[50] Rene Heinzl, Philipp Schwaha, Michael Spevak, and Tibor Grasser. Adaptive mesh generation for TCAD with guaranteed error bounds. In *The 2005 European Simulation and Modelling Conference Proceedings*, pages 425–429, 2005. talk: European Simulation and Modeling Conference (ESMC), Porto; 2005-10-26 – 2005-10-28.

[51] Philipp Schwaha, Rene Heinzl, Michael Spevak, and Tibor Grasser. Coupling three-dimensional mesh adaptation with an a posteriori error estimator. In *International Conference on Simulation of Semiconductor Processes and Devices 2005*, pages 235–238, 2005. poster presentation: International Conference on Simulation of Semiconductor Processes and Devices (SISPAD), Tokyo; 2005-09-01 – 2005-09-03.

[52] Philipp Schwaha, Rene Heinzl, Wolfgang Brezna, Jürgen Smoliner, H. Enichlmair, R. Minixhofer, and Tibor Grasser. Fully three-dimensional analysis of leakage current in non-planar oxides. In *The 2005 European Simulation and Modelling Conference Proceedings*, pages 469–473, 2005. talk: European Simulation and Modeling Conference (ESMC), Porto; 2005-10-24 – 2005-10-26.

[53] Rene Heinzl, Michael Spevak, Philipp Schwaha, and Tibor Grasser. A novel technique for coupling three dimensional mesh adaption with an a posteriori error estimator. In *2005 PhD Research in Microelectronics and Electronics*, pages 175–178, Vol. 1, 2005. talk: PhD Research in Microelectronics and Electronics (PRIME), Lausanne; 2005-07-25 – 2005-07-28.

[54] Elaf Al-Ani, Rene Heinzl, Philipp Schwaha, Tibor Grasser, and Siegfried Selberherr. Three-dimensional state-of-the-art topography simulation. In *The 2005 European Simulation and Modelling Conference Proceedings*, pages 430–432, 2005. talk: European Simulation and Modeling Conference (ESMC), Porto; 2005-10-24 – 2005-10-26.

[55] Hajdin Ceric, Markus Karner, Alexandre Nentchev, Philipp Schwaha, Stephan Enzo Ungersböck, and Siegfried Selberherr. VISTA status report II. Technical report, E360 - Institute for Microelectronics; Vienna University of Technology, 2005.

[56] Phillip Schwaha, Solveig Anders, V Tamosiunas, Werner Schrenk, and Gottfried Strasser. Light field in quantum cascade ring lasers. In *GMe Annual Report 2003*, page 39. Gesellschaft für Mikro- und Nanoelektronik, 2004.

[57] Quantum cascade ring lasers. Diplomarbeit, Institut für Festkörperelektronik, Technische Universität Wien, 2004.

[58] Christian Pflügl, Solveig Anders, Werner Schrenk, Phillip Schwaha, and Gottfried Strasser. Electrically pumped GaAs-based quantum cascade microcavities. In *Program and Abstracs*, page 1150, 2003. talk: International Conference on Nonequilibrium Carrier Dynamics in Semiconductors (HCIS), Modena, Italien; 2003-07-28 – 2003-08-01.

[59] Solveig Anders, Phillip Schwaha, Werner Schrenk, and Gottfried Strasser. Electrically pumped quantum cascade ring lasers. poster presentation: 12th Euro-MBE Workshop, Bad Hofgastein, Österreich; 2003-02-16 – 2003-02-19, 2003.

[60] Phillip Schwaha, Solveig Anders, Tomas Roch, Werner Schrenk, and Gottfried Strasser. Electrically pumped quantum cascade ring lasers. In *Proceeding GMe Forum 2003*, pages 77–80, 2003. poster presentation: GMe Forum 2003, Wien, Austria; 2003-04-10 – 2003-04-11.