Computer Methods for Design Automation

by

Christian Bliek

Burgerlijk Elektro-Werktuigbouwkundig Ingenieur
Katholieke Universiteit Leuven
Belgium, June 1986

Submitted to the Department of Ocean Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1992

Author ....
Department of Ocean Engineering
July 6, 1992

Certified by ....
Professor Chryssostomos Chryssostomidis
Thesis Supervisor

Accepted by ....
A. Douglas Carmichael
Chairman, Departmental Committee on Graduate Students
Department of Ocean Engineering

Computer Methods for Design Automation
by
Christian Bliek

Submitted to the Department of Ocean Engineering
on July 6, 1992, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Design automation requires reliable methods for solving the equations describing the performance of the engineering system. While progress has been made to provide good algorithms for polynomial systems, we propose new techniques for the solution of general nonlinear algebraic systems. Moreover, the interval arithmetic approach we have chosen also guarantees numerical reliability.

In this thesis we present a number of new algorithms that improve both the quality and the efficiency of existing interval arithmetic techniques for enclosing the solution of nonlinear algebraic equations. More specifically, we propose an exact algorithm for the solution of midpoint preconditioned linear interval equations. We extend existing techniques for automatic compilation of fast partial derivatives to include interval slopes and have conceived a number of graph algorithms to improve their efficiency for general computational graphs. Furthermore, we have devised variable precision techniques to automatically control the required precision based on interval width. Finally, we have unified a number of enclosure languages using denotational semantics.

Since design computations can be performed with conservatively bounded models instead of point models, this approach also allows us to develop a framework in which the hierarchical design process can take place and to address the consistency problem associated with incremental refinements. In addition, conservative enclosures are particularly useful when computations are made in a distributed fashion, at different speeds and when communication delays are unpredictable.

We believe that since numerical set computations for engineering design fills a void between traditional numerical analysis and discrete mathematics, it promises to be an exciting new area of research.

Thesis Supervisor: Chryssostomos Chryssostomidis
Title: Professor of Ocean Engineering

à Doudou

Acknowledgments

Many people contributed to the work presented in this thesis.

My advisor, Professor Chryssostomidis gave me the opportunity to come to MIT and supported me throughout the years. He allowed me to explore new areas of research and has helped to broaden my background considerably.

Members of thesis committee, Nicholas Patrikalakis and Warren Seering gave valuable feedback and shared their enthusiasm.

Mike Drooker, my office mates and fellow students have provided for years of stimulating discussions, encouragement and friendship.

I thank my parents, who provided me with a stimulating environment to grow and learn, and my brother for showing me the way. Thanks to my friends, near and far, for standing by me.

Most of all, I would like to thank my wife Françoise for her patience and endurance, she made it all worthwhile.

This work was supported, in part, by the Office of Naval Research under the contracts N00014-87-K-0462, N00014-89-J-1187 and N00014-91-J-1014.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Design Automation

The design of large-scale complex systems usually requires the participation of a large number of people. It typically proceeds as follows. A handful of people, e.g. the project manager, chief engineer and a few experts lay down initial concepts and dimensions. They tend to rely on back of the envelope type calculations or sometimes on personal computers and spread sheets for working through this very first design stage. This initial design solution is refined by the design engineers. Drafting people finally detail and enter the entire model into the computer. Once the necessary detail is available, some preliminary analysis is done. It is only very late in the design cycle that CAD software is used and extensive analysis, such as finite element analysis, is used to check the performance of the design.

1.1.2 Shortcomings

When designing complex systems, usually about 70% of the cost is committed during the early design stages. This means that the most important decisions are made at the very beginning of the design cycle, when there is very little precise information concerning the unfolding of the remaining part of the design process. The people which have to make these decisions essentially rely on experience and intuition. Once these important decisions are made, it becomes very costly to reevaluate them, since substantial changes in the overall system would have to be made. To avoid major redesign efforts, corrections and fixes are made all along the design process, resulting in workable but usually suboptimal solutions. Ironically most of the computer resources are used during the later design stages. They are essentially used to create CAD models of the designed structure and to perform complicated numerical analysis.

1.2 Thesis

Although some aspects of the design process have been automated in an incremental fashion, we believe that further progress in this area requires us to rethink some of the basic premises.

To take advantage of the flexibility and the speed of available computer power requires integration of the entire design process in the machine. On the one hand, this can provide more and better information to support decision making so that it can be more effective. On the other hand when unanticipated problems or costs surface at a later design stage substantial changes can be made without incurring the large extra costs.

We believe that the key issues in conceiving such a system are reliability and robustness. Indeed, while designers often rely on numerical computations, they will always check the results to ensure their validity. Furthermore, current implementations do not allow for late coming information and their performance degrades drastically in the face of inconsistencies. In this thesis we therefore study fully reliable computational methods that allow for late coming information and for the detection of inconsistencies. Finally, it should be noted that while the current technology allows for very fast computations, the algorithmic time complexity remains an important issue.

We are convinced that this type of system will not only reduce the number of people involved in the design cycle, but that it will speed up the design process and improve the overall quality of the designed product.

1.3 Overview

In chapter 2 we discuss the problem of design in more depth and propose a general framework for design automation. Some basic aspects of interval arithmetic, which allows for full reliability of numerical algorithms, are reviewed in chapter 3. In this chapter we also describe a novel approach for specification of the denotational semantics of enclosure programming languages. In chapter 4 we describe a new algorithm for the solution of linear interval systems. This algorithm is a significant improvement of traditional linear interval solvers. A number of new results relating to computational graphs are proposed in chapter 5. In particular, we discuss the problem of optimal computation of partial derivatives and interval slopes, and propose a method for computing upperbounds on the dimensionality of the solution of computational graphs. The following chapter, chapter 6, contains a number of new methods that allow for automatic control of the required precision of interval computations, including the use of variable precision. In chapter 7 we put these techniques in perspective and discuss their importance to the design problem. This includes the computation of solution sets, the hierarchical aspect of design and its relation to the consistency problem. We conclude this chapter with the presentation of an approach for reliable and robust concurrent computations. Finally, in chapter 8, we sum up and discuss a number of further research topics in this new area of research.

For the reader interested only in the contributions in the field of interval arithmetic, we recommend to quickly go over chapter 3 and read chapters 4, 5 and 6. If one wants to focus only on the design aspects, we would suggest to read chapters 2 and 7.

Chapter 2

Design

2.1 Introduction

We believe that the general problem of design is of a combinatorial nature. As a result, except for a few of special cases, one can only approach this problem in a heuristic fashion. It is probably for this reason that design is often not considered an exact science.

Combinatorial explosion is typically avoided by incrementally adding problem specifications as the design process proceeds. It is therefore important to carefully distinguish the givens from the unknowns. In our view, problem specification can only be done by the designer. In addition, we will assume that models associated with the design space are given. The task of capturing requirements and modeling information can however considerably be facilitated by high level tools. Furthermore, we believe that the selection and solution of the appropriate models can to a large extent be automated. Our ultimate goal can therefore only be to devise tools that allow us to represent the available information effectively and to automatically select the set of design models that satisfy the imposed requirements, not to replace the designer.

We believe that further automation of the design process requires a careful study of the interaction of the representation problem and the tractability of automatic model selection methods. In this thesis we study a number of key issues related to automatic solution techniques for design.

In section 2.2 we define our approach of the design problem more precisely. In section 2.3 we propose a framework for design and identify the issues that need to be addressed. We then contrast our approach with existing ones and review some of the previous work in the area of design automation in section 2.4. Finally, in section 2.5 we use the Design Society metaphor to put this framework into perspective.

2.2 Towards a Theory of Design

In this section we present our interpretation of design. It also provides an abstract setting for the framework proposed in the next section.

2.2.1 Functional Requirements and Hierarchy

Design can be seen as the process of mapping *functional requirements* into a description of a realization of those requirements. By definition, functional requirements are functions or properties that the design needs to provide. This mapping process generally involves search to find the particular realizations that provide the required functions. A realization is unambiguously described by *design variables*.

The design process is hierarchical. First, a general solution is found by imposing the most important functional requirements. Then, more refined solutions can be obtained by imposing additional requirements. This mapping process from functional requirements to design variables continues level by level until sufficient detail is reached. The resulting model relates the functional requirements to the realization of those requirements and is called a *design model*. Although one ofter refers to this hierarchical approach as a design principle, it really only is a heuristic to control the complexity of the search space. Indeed, the designer attempts to focus the search of the solution space by identifying the dominant characteristics of the design model first.

Both design variables and functional requirements contain information concerning the design model. The fundamental difference between them is in the way they are used. Functional requirements correspond to those pieces of information that describe the problem specification, while the design variables are chosen in order to satisfy them. In a different design situation some of the functional requirements can become design variables while some design variables can be qualified as functional requirements. Note also that by definition the designer cannot require that design variables behave in a certain way, otherwise he would need to express this as additional functional requirements.

Our definition of design doesn't necessarily exclude manufacturing. Indeed, some of the design variables could in fact be associated with manufacturing processes and the functional requirements can in principle be mapped directly into process variables. This is however not how it is done traditionally. Instead, the mapping between functional requirements and design variables, i.e. the design process, is augmented with a second mapping from design variables to process variables, i.e. the manufacturing process. Fortunately, this extension is of the same nature as the design process itself, and can be supported by the same framework.

2.2.2 Design and Modeling

As described above, design is a requirement driven top-down selection process. However, the incremental specification of the functional requirements can only be meaningful in an appropriate modeling context. The design process therefore requires access to modeling information.

Models are valid representations of possible realizations and are described by *model variables*. Typical engineering models are beams, engines, resistors, while height, width, power, resistance are model variables. Models cannot only be created by definition of variables, equations and computations, but also by composition or enumeration of more specific ones. Furthermore they can inherit information of more general models. Design is the selection of sets of realizations in the solution space and therefore requires model selection operators. This includes the specification of parameters, objective functions, model types or names of previous designs.

The representation of models and their organization should be conceived so as to facilitate the selection process. To this end models should contain as much information as possible concerning the feasible realizations while avoiding combinatorial enumeration. Furthermore, design models can be refined by incorporating new information about the design solutions in an incremental fashion.

Establishing the relations between model variables, involves modeling knowledge, i.e. understanding of how a realization provides certain functions. Models can be created by modelers, can come in the form of predefined packages and even as manufacturers catalogs. The formulation of new functional requirements, based on more general design models, on the other hand, requires design knowledge, i.e. an insight as to what functions the design needs to provide to serve its purpose. Both modeling and design, usually rely on problem decomposition to break up bigger problems into smaller ones. Indeed, design requires understanding of how important requirements can be decomposed, while modeling is concerned with defining the right pieces of the model and putting them together.

The design process can also be viewed as model instantiation. In the case of parametric design, this process is straightforward as only parametric computations of prespecified models are required. Furthermore, in this case design knowledge can often be encoded, so that new functional requirements are automatically defined based on information provided by the design model. This implies that for parametric design, the design process can almost entirely be automated. In most cases however, the design process can be computationally intensive, since due to combinatorial complexity it usually is not possible to precompute all candidate solutions at modeling time.

As further scientific progress is made, one can improve existing models and incorporate new models. The automatic generation of entirely new models from novel combinations of functional requirements, which is at the heart of the engineering discipline, does however remains an open challenge.

2.2.3 Design and Set Theory

In a set theoretical sense, all possible realizations represent a set. The size of this solution set is incrementally reduced as the design process proceeds. Sets of solutions are represented by models and functional requirements select desired models from the current solution set.

The set theoretical interpretation of design allows for a very simple definition of hierarchy. Indeed, a model is a *refinement* of another if it is a subset of that model. Note that by this definition the introduction of additional modeling variables automatically constitutes a refinement. It should be emphasised that, as dictated by the subset relation, model hierarchies are only partially ordered. Finally, as with any hierarchy, one can implicitly incorporate an inheritance mechanism and only represent additional information.

Any operator that cuts down the solution space can represent a functional requirement. The boolean `And` operator for example, represents an intersection with another model that has certain desired properties. Optimization can also be viewed as a selection process retaining only the solutions from the current design model that are optimal w.r.t. a design objective. Observe that an optimum solution can be a set, it does not necessarily have to be a singleton. Note also that different optimization requirements can be compatible as long as one optimum doesn't preclude the other. This trivially holds for uncoupled optimization problems for example, with disjunct optimization domains.

Algebraic equations can also be considered as selection operators for they require a certain relation to hold between different quantities. They can each be viewed as an intersection with all possible solutions satisfying the equation, i.e. the hyperspace defined by the equation. The same consideration holds for equations defining solution strips. These equations, such as for example algebraic equations with interval coefficients, are called interval equations.

When no solutions remain, one obtains the empty set. This means that the models currently available do not contain a realization that can satisfy all functional requirements. Either new models have to be created that provide the desired functionality or the requirements have to be relaxed.

Different methods can be used to represent solution sets. Non-manifold boundary modeling techniques for example, can explicitly represent general point sets, whether continuous or discrete. The complexity of the representation can however be reduced if one considers special cases, such as manifold boundary models or convex point sets. In this thesis we will focus on interval representations, which are in fact a special case of convex point sets and represent the set of values a variable can take.

Discrete point sets can simply be represented by enumeration. This can include types, corresponding to a classification of models based on specific attributes, as well as ordered or unordered lists, corresponding to a set of models.

2.3 A Framework for Design

In the sections below, we describe the key elements that need to be integrated in a framework for design and point to the issues that will be resolved in later chapters.

2.3.1 Bounded Models

As indicated in the previous section, we are concerned with sets of solutions. We therefore propose to develop appropriate representations and computational methods to work with *bounded models*. In addition, to provide for fully reliable computations for design automation, these techniques need to control both approximation errors, such as discretization errors, and rounding errors. This approach should be contrasted with the traditional estimation models, where computations are performed with a representative point solution. Not only do approximate estimation models provide no information concerning the size of the solution sets, the accuracy of the corresponding design computations cannot be automatically verified.

We will say that an approximate bounded model is *conservative* if it encloses the solution set (see figure 2.1). As a consequence, we can no longer simply classify the elements of the domain based on whether or not they are solutions. The overestimation introduced by the conservative nature of the computations requires us to consider three types of sets. The exterior set contains no solutions, whereas all elements of the interior set are solutions. The boundary set contains elements with unknown classification. This is illustrated in figure 2.2. In this work, we will focus on conservative models which approximate the solution set by including both the interior and the boundary set. Note that while approximate

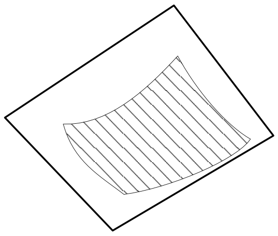

Figure 2.1: Approximation by Conservative Bounded Model

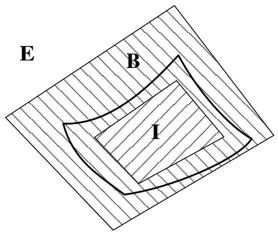

Figure 2.2: Definition of Interior (I), Boundary (B) and Exterior (E) Sets

bounded models enclose all possible realizations that satisfy the functional requirements, not all elements will be solutions. To efficiently prune the search space however, whether with numerical solution algorithms or branch and bound type techniques, it is desired that the overestimation remain small. Indeed, tighter bounds lead to faster elimination of unsatisfactory or suboptimal solutions. On the other hand, sometimes overestimation is introduced to reduce the complexity of the solution algorithms. Conservative enclosures of discrete point sets are a case in point.

As the size of the solution set is reduced, more general models of the hierarchy can be refined allowing for the computation of tighter bounds, so that more accurate models can be used in the selection process. It is important that no additional solutions are introduced in this process. Indeed, if the refined solution set is a subset of the original one, all statements concerning the enclosing solution set hold for the refined set as well. In that case, we say that a refinement is *consistent*. Since we are working with conservative models, we need to distinguish between semantic and procedural consistency. *Semantic consistency* refers to the consistency of the exact solution set, i.e. without overestimation. Consistency of approximate models on the other hand is called *procedural consistency*. Both concepts are illustrated in figure 2.3. Note that the latter definition can also be used for obtaining better approximations when the exact solution remains unchanged.

While semantic consistency preserves the meaning of models, procedural consistency ensures that the associated decision process remains valid. As illustrated in figure 2.3, the two concepts are not identical and cannot freely be interchanged. In particular, if a solution technique is conservative but not procedurally consistent, a refinement can be procedurally consistent only if it is enclosed by all approximations used in the solution process.

As indicated earlier, optimization problems can be associated with functional requirements. Similar techniques can therefore be used to conservatively bound the corresponding solution sets and the same definitions of consistency can be used. Observe that if the value

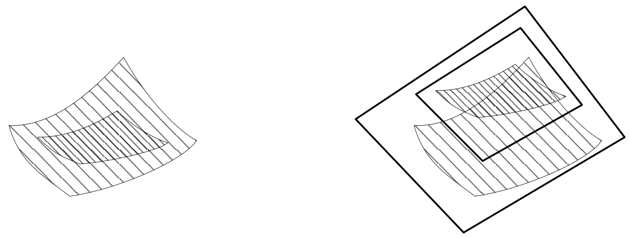

Figure 2.3: Semantic and Procedural Consistency

of the objective function is to be used in the computation, refined definitions should be enclosed by the original one. One of the main advantages of the model refinement technique is that the hierarchical aspects of the design process can be maintained. This is of crucial importance for controlling the complexity of the solution process.

It should be noted that model improvement is fundamentally different from model refinement. While the latter refers to incorporation of information during the design process, the former relates to the definition of new models that supersede older ones. In fact, improved models can be discovered and learned during the design process, so that future performance will be increased.

While bounded models can be represented in different ways, we have chosen to use interval arithmetic for a number of reasons. It provides a natural setting for computations with bounded sets and allows for automatic control of rounding errors. In addition, a large number of techniques have been developed for interval computations. The most important reason however is that the complexity of the algorithms remains polynomial.

We have decided to focus on nonlinear algebraic problems in particular, as the corresponding techniques can be extended to cover a large number of design situations. A brief review of interval arithmetic techniques related to the solution of nonlinear algebraic equations is presented in chapter 3. General nonlinear interval arithmetic solvers conservatively approximate nonlinear systems by linear ones and use linear interval solvers on the resulting system of equations. We have made contributions in both areas. In chapter 5 new algorithms are presented for more efficient computation of nonlinear approximations, while in chapter 4 we describe new linear interval solution algorithms that produce better enclosures then the existing ones.

We would like to indicate that for frequently used models or functions tight bounds can be precomputed. One can for example use techniques based on turning and border points (see figure 2.4) similar to the ones proposed by [36, 138]. In addition, efficient computation strategies can be precompiled. In our opinion this library-like approach, which explicitly incorporates modeling knowledge, will prove very valuable in practice.

Computations with sets of models that include ranges of operations can also be performed in the context of interval arithmetic. In this case, one imposes requirements on the size of the interior set. This type of computation therefore necessitates the use of conservative interior bounds. While very little work has been done on this subject, we believe that most interval techniques can simply extended to handle this case. A preliminary discussion of the inner solution of linear interval equations can already be found in [46], while the applicability of bounding interval endpoints is mentioned in [260].

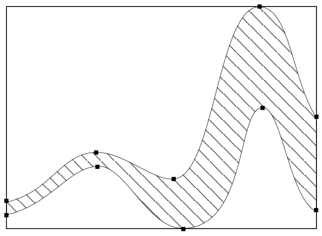

Figure 2.4: Precomputation of Interval Bounds

Appropriate computational models can automatically be selected based on the size of the solution set. Indeed, the width of an interval is usually a good indicator of the required accuracy. In chapter 6 we present a number of novel techniques for variable accuracy interval computations that are based on this idea. These techniques can also be extended to include interval range computations. In this case, the required accuracy should however be controlled by the size of the boundary set.

Finally, the reader is referred to chapter 7 for a discussion of consistency in interval arithmetic computations and practical examples.

2.3.2 Design Graphs

Graphs are a powerful, flexible and efficient way of representing structured information and are in our opinion particularly well suited for representing the dependencies prominent in engineering models. The two most important types of modeling dependencies are the relation between model variables and the hierarchical dependency of different design models. Models are typically associated with nodes, arcs with dependencies. For simplicity the models representing variables will be referred to as variable nodes.

The selection of modeling graphs based on the functional requirements can be done by activation algorithms. For computational graphs associated with equations or functions for example, these algorithms will automatically determine a partial order required for their solution [218]. In this thesis, we focus on reliable solution techniques for strongly coupled components of nonlinear computational graphs.

In general, design graphs can represent distributed computational systems. Moreover nodes can be associated with a variety of different models, as long as they satisfy a number of conditions associated with the nonlinear solution algorithms. For the basic nonlinear interval solvers, described in chapter 3, it is typically required that models provide a conservative linear approximation. More sophisticated methods, such as the ones introduced in chapter 6, can specify the required accuracy and request that the model provide nonlinear solution bounds. It should be emphasised that except for the above interface conditions, the actual computation of this information is internal to the model, in fact it can even be performed by a dedicated processor. In addition, this approach allows for the use of model libraries, which can include frequently used engineering models such as beams, resistors, etc., and manufacturers catalogs, providing product specific information.

While some models are valid throughout the entire domain, others are valid only over a limited domain. In our framework, this information can simply be represented by inheritance of the corresponding bounded model. Furthermore, activation algorithms can automatically select valid models based on the current solution domain.

The hierarchical structure of the design process is solely determined by the functional requirements. The corresponding partial order can therefore only be associated with design models and is not predetermined. Indeed, the inheritance mechanisms used in section 2.3.4 are semantic ones. A hierarchic level is a set of design models that can determine the corresponding value of the design variables, given the value of the functional requirements. The selection of more specific design models typically requires the introduction of new variable nodes. In this case a refinement of the more general models can for example be obtained by node condensation (see chapter 5), so that the complexity of the resulting model remains unchanged. For composed models it sometimes is necessary to introduce variable nodes describing more then one submodel, hereby leading to a partially overlapping decomposition [242]. This situation is illustrated in figure 2.5. Rectangles represent models (e.g. equations), circles variables nodes. The original decomposition, required for model

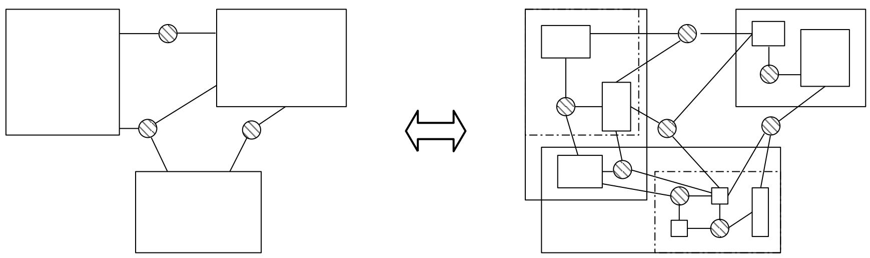

Figure 2.5: Partially Overlapping Decompositions

refinement, can however be reestablished by condensing these particular nodes.

Graph representations also encompass a large number of engineering discretizations. In fact, numerical solution packages use graph algorithms to exploit the sparsity of the associated matrices. We believe that it is therefore beneficial to directly represent the discretization information with graphs and use efficient node condensation techniques for their solution. In our framework, mesh refinement corresponds with automatic definition of functional requirements based on the governing equations, while finite element condensation techniques represent model refinements. Note also that the mesh refinement process can be controlled by the variable accuracy techniques.

Graphs that are not active and their associated semantic networks represent different views of a design model and provide the designer with information deduced from the current design solution (cfr. slices in [242]). Indeed, although the designer is by definition only interested in the functional requirements, he needs to be well informed about the design model so that he can impose appropriate additional functional requirements. While the information emerging from the different views is important, it is not directly controlling the design process, unless new functional requirements are added to do so. Since the nature of the solution usually is not know a priory, specific information requests can however only be made interactively.

2.3.3 Concurrency in Design

Problem subdivision, which is essential in engineering design forms the basis for specialization. In addition, it allows for the concurrent solution of the corresponding subproblems, where *concurrency* refers to parallelism with unbounded resources. We therefore believe that systems for design automation can be organized as a network of simple minded agents, encoding knowledge in a modular way and allowing for concurrency. Furthermore, parallelism is needed to overcome the fundamental speed limitations of current computer technology.

It should be emphasized that concurrent architectures and algorithms are conceptually very different as compared to their sequential counterparts. In fact, it is generally acknowledged that difficult problems have parallel decompositions that yield easier subtasks than serial decompositions. Techniques for automatic discovery of parallel subtasks on the other hand, have enjoyed only limited success due to the associated complexity issues. It is therefore important to incorporate concurrency information so that it is directly available for organizing parallel computation.

Unfortunately, distributed concurrent systems require mechanisms to coordinate local computations and ensure global consistency. Indeed, while local behavior can be assumed to be consistent and sequential (cfr. microtheories in [95]), there is no global knowledge nor time in a concurrent system [96]. In fact, organizations of concurrent decision making agents can show unstable or even chaotic behavior [105].

Computation with bounded models, as proposed in this framework and illustrated in chapter 7, significantly reduces the consistency problem in concurrent systems. Indeed, as decisions are based on solution sets, they also are valid for any subset. Consistency as defined here, therefore allows for asynchronous communications, as well as late arriving and changing information. As compared to the traditional point wise notion of consistency, the coordination mechanisms required by our approach do not constrain the performance of the system as much. Furthermore, global behavior can conveniently be controlled by the use of conservative enclosures.

However, as indicated earlier, inconsistencies can arise in model refinement. In addition, empty solution sets can prompt the designer for redefinitions or parameter changes. Fortunately, these consistency issues are of a topological nature and can be resolved algorithmically. We refer the reader to [144] for a general treatment of strict consistency contracts in concurrent languages. A practical implementation of these contracts requires that conventions about different communication mechanisms be made. The corresponding protocols can be viewed as a generalization of the continuation idea. Commands correspond to a single message, while evaluation protocols consist of both a request and a reply part. Protocols for general distributed control mechanisms, include a specification of the message set and their partial order.

Concurrent computations come to an end when local agents detect no further change and no additional decisions can be made. At this state of quiescence called *design equilibrium*, consistency contracts ensure that the design model reflects current functional requirements. A discussion of the termination problem for asynchronous distributed iterative algorithms can be found in [19]. Note that except for the partial order of the design models selected by the functional requirements, the sequence in which operations are performed is immaterial to the actual solution. External changes can therefore take place at any time and models can dynamically forward queries as soon as model refinements have been made.

Automated design systems require mechanisms to control the cost of computation, including communication and processor time, and both real and virtual memory usage. These more complicated contracts can make explicit tradeoffs between speed, optimality and modeling bounds hereby allowing the designer to impose additional functional requirements before equilibrium is reached.

2.3.4 A Language for Design

To be effective, a design system should be able to capture design as well as modeling information in a clear and unambiguous manner. Furthermore, communication should be done at an appropriate level. In our opinion both requirements entail the creation of a high level language for design that allows us to incorporate as much information as possible concerning the designers knowledge and intent. Indeed, the availability of this information can significantly reduce unnecessary complexity of solution algorithms and is therefore essential for further automation.

A language for design should support both design and modeling constructs. We have already indicated that modeling operators are sometimes used during the design process while modeling can involve selection. Although it is often artificial to distinguish both processes, modeling usually is associated with the expression of definitions, while the application of these definitions and the associate computational process corresponds with design.

One of the most powerful mechanism for expressing information in high level languages is inheritance. Indeed, hierarchical semantic networks and the associated inheritance mechanism can implicitly associate existing information with new definitions. Semantic inheritance and its compact notation can be used for the definition of models as well as relations. While for models this implies the implicit definition of a large number of modeling variables and relations associated with the parent models:

```
(Define Name
  (Model {ParentModel*}
     (VariableBinding*)
    Body))
```

application of `Lambda` expressions can automatically create a large number of arcs relating its model arguments. The following simple resistor model:

```
(Define Resistor
  (Model {}
     ((current Real)
      (resistance Real)
      (voltage Real))
    (= (* current resistance) voltage)))
```

can be used in series by application of:

```
(Define Connect-in-Series
  (Lambda (Resistor1 Resistor2) {Resistor}
    (= current (current of Resistor1) (current of Resistor2))
    (= resistance (+ (resistance of Resistor1) (resistance of Resistor2)))
    (= voltage (+ (voltage of Resistor1) (voltage of Resistor2)))))
```

Variables defined by semantic inheritance can simply be accessed by name. Model variables associated with components, on the other hand require the use of accessor constructs.

In the above example the left hand variables could be referenced by semantic inheritance of the `Resistor` model, while the variables of the component resistors had to be accessed. In addition, name spaces can be associated with top level inheritance to reduce the complexity of the selection process, hereby providing design specific contexts. Types, which reflect general information associated with models, are themselves first class models. This implies not only that types can be inherited but also that they can form a hierarchy. The `current` variable of `Resistor` for example, inherits the model `Real`, representing the set of all real numbers. `Real` itself could inherit from the type `Complex`.

We want to reiterate that this inheritance mechanism is of a semantic nature. It facilitates the creation of models and the expression of functional requirements, but does not necessarily reflect the hierarchy associated with design models. In the example above one could for example express functional requirements relating to the components before the ones associated with the series resistor.

Boolean operations unify the variables of their arguments. If, for example, we define:

```
(Define Heating-Element
  (Model {Resistor}
     ((power Real))
    (= (* current voltage) power)))
```

the statement:

```
(And Heating-Element Series-Resistor)
```

will automatically identify the variables `current`, `resistance` and `voltage` of both models. This type of identification mechanism is important, for it reduces the number of trivial identities and therefore the complexity of the solution process. The boolean operator `Or` also allows us to enumerate models:

```
(Define resistance (Or 1kΩ 10kΩ))
```

The selection of the solution set by optimization can be expressed by:

`(Optimize` *Objective Model*`)`

resulting in the subset of *Model* that minimizes *Objective*.

A concise specification of the meaning of language expressions, requires the definition of a semantic domain. Whereas denotational semantics (see chapter 3) uses a λ-calculus denotation, we propose to use graph denotations for the semantic specification of concurrent languages. The semantics of a design language will therefore be expressed as a mapping between syntax and design graphs. As with traditional denotational semantics, this representation is independent of specific parallel computer architectures or network implementations. The topological computational structure will however be directly available to the compiler. In this regard, we would like to indicate that as design graphs will be more structured then computational graphs associated with general purpose languages, the allocation of computational networks on specific architecture can be done more efficiently.

The practical value of graph denotations is dependent on the combinatorial complexity of the associated solution algorithms. As a consequence basic graph denotational components for design will reflect the state of the art in design automation. As this thesis contains an number of contributions in the area of automatic solution techniques, it will allow for more powerful language constructs.

Observe that the design language does not concern itself with message passing, but rather with the construction of appropriate graphs and automatic creation of protocols. Practical implementations will however need to control the interaction of concurrent computational systems with the sequential designer, requiring integration of language editors, command shells and system managers. This approach would also allow for the incremental definition of inconsistency handlers that are invoked upon the detection of inconsistencies. These user defined handlers incorporate design knowledge by automatic respecification of functional requirements.

2.4 Previous work on Design Automation

2.4.1 Design Theory

In *Synthesis of Form*, one of the early works in the area of design theory and methodology, Alexander [8] defines the design problem as a match between context, i.e. functional requirements, and form, i.e. the subject of design. In the *analytical* part of the design process, requirements are analyzed for possible interactions and a minimum information exchange principle is used to attempt to decouple subsystems and devise a hierarchy of requirements. Then, in the *synthesis* part, constructive diagrams are created to incorporate both requirements and form aspects. By composition and fusion of these diagrams, called *hypotheses*, form is synthesized in a bottom up fashion. Throughout this process, binary decision variables are used to indicate fit or misfit between context and form, i.e. if the hypothesis are satisfied.

In our opinion, one can in general not define functional requirements independently of the physical solution. Indeed, a different approach is proposed by Mandel and Chryssostomidis. In [151], they describe a hierarchical method for designing complex systems using the principle of subdivision. As opposed to [8], functional requirements are specified in a top down fashion based on a high level description of the design solution. New design information is then incorporated through the hierarchical use of iterative techniques. In this thesis can be viewed as a continuation of this work, with a specific emphasis on the robustness and consistency problem arizing in the automation of the hierarchical solution process.

Suh, in *Principles of Design* [240], has attempted to provide a scientific basis for design by developing fundamental design axioms. The *independence* axiom dictates that one should maintain the independence of the functional requirements, hereby avoiding coupled design situations. This allows the designer to satisfy the functional requirements without having to make compromises. Finally, by the *information* axiom, of all the design solutions that satisfy the independence axiom, the one with the least information content is defined as the best one.

Although the guidelines offered by this approach are very valuable, we believe that for many practical problems it is usually not possible to maintain the independence of the functional requirements. Nor is it, in our opinion, practical to eliminate some of the functional requirements, or to come up with a different physical solution in order to obtain independence, as would be required by the axiomatic approach. The techniques proposed in this thesis therefore focus on coupled design problems, and can be viewed as a rational approach for making engineering compromises in coupled design situations.

It is generally acknowledged, that optimization is not used to its full potential in the design process. In [244], Taguchi indicates that design techniques based on optimization lead to better and more integrated products. He therefore developed a quality loss minimization framework for design and in order to obtain robust products, defines sensitivity to noise factors as the objective to be minimized. As pointed out by Suh [240], both approaches do not necessarily contradict each other. Indeed, minimization of the information content associated with the functional requirements also promotes simple and robust solutions. However, this imposed criterion often is not compatible with the wishes of the designer and in fact generally precludes a minimum cost design. While we agree that optimization is an integral part of the design process, it is in our opinion not appropriate to prescribe objective functions and hereby to define what good design is. We believe that this should be left to the designer to decide. Note however, that in our framework, it is still possible to define these particular objectives explicitly.

A general theory of design is presented by Yoshikawa in [272, 250]. Design is defined as a mapping from the attribute space to the function space, both of which are defined over the entity concept set. Using three basic axioms, the axiom of recognition, the axiom of correspondence and the axiom of operation, it follows that with ideal knowledge the specification of function directly leads to a design solution. With real knowledge however, it is possible to make design a convergent process, but nor the uniqueness, nor the existence of the solution is guaranteed. This abstract set theoretical view of design should be contrasted to the practical approach presented in this work. Whereas the general design theory focuses on the difference between real and ideal knowledge and pays very little attention to complexity issues, our goal is to devise tractable set algorithms that operate in real knowledge in order to find conservative design solutions.

In recent years researchers have becoming increasingly aware of the need for a sound theoretical basis for design automation. Since a general overview of research in this area is given in the papers by Finger and Dixon [60, 61], we will only review a number of recent contributions.

Painter, in [178], identifies energy as the common denominator in mechanical systems and proposes a unifying framework for design based on bond graph representations, which are independent of geometry. This representation, and the corresponding transformations have in fact recently been used in the area of artificial intelligence for further automation of the synthesis process [254]. An interesting extension of the energy based approach to include geometry is discussed in [220].

Kott and May [125] propose to classify design problems, either as decomposable or as monolithic and suggest that, depending on the type of problem, a different model of the design process should be used. They do however acknowledge that no design problem will be exclusively decomposable, or exclusively monolithic. In our opinion both types of design problems should be allowed in the same framework, where the solution of monolithic or coupled problems typically dictates interface conditions for the subsequent decomposition into decoupled problems.

Other researchers have attempted to cast the design process into explicit iterations between synthesis and analysis [102]. In fact, most domain specific design programs are based on this methodology. In this generate and test approach, analysis has obtained the status of performance checking while other techniques are used to propose design changes.

It is our opinion that one should solve the design equations directly so that the above iterations are kept implicit. In fact, the *design for analysis* [241] approach suggests that designers should only be allowed to use models which can be analyzed in an automated fashion. A number of systematic approaches of exploring the solution space are reviewed by [245]. In this context it should be noted that while our approach allows for discrete enumeration by using enclosures, we will not address the general combinatorial problem associated with discrete point sets.

2.4.2 Modeling

In one of the early control-mathematical theories on hierarchical systems [158], Mesarovic studies multilevel goal-seeking and optimizing systems. Analysis of the conditions under which this type of systems can be decomposed and coordinated in a hierarchical structure lead to the definition of the *interaction decoupling*, *interaction balance*, and *interaction prediction* coordination principles. It is interesting to note that since in our framework coordination is determined by conservative approximation of possible interactions, it is in fact a combination of all three coordination techniques. The more restrictive case of linear systems will not be discussed here. We refer the reader to the literature review presented chapter 5 or to [48] for a general overview.

A number of techniques have been suggested to identify measures of importance. Saaty [211] in particular, proposes to use pairwise comparisons in conjunction with a eigenvalue formulation to obtain weights which reflect the priorities of different parts of the hierarchy. These weights can subsequently be used to analyze the consistency of the comparisons and to guide conflict resolution. Unfortunately, this technique assumes that the pairwise preferences are given and can therefore not be used in an automated fashion.

Hierarchies based on inclusion of operators and metrics were already defined by Manheim in [152] to solve the highway location problem. However, the actual solution is found by a best first search combined with a decision process based on Bayesian probability theory.

Inheritance mechanisms have been used extensively in the field of artificial intelligence [59] as a representation for problem solving systems, see [247] for a recent overview. More recently a formal approach was proposed to represent product model information [54]. Although, this approach is not directly based on set inclusion, it forms in our opinion an important contribution towards extending automation beyond parametric design.

Sussman points out that most engineering systems can only be partioned in almost hierarchical decompositions. Furthermore, these decompositions are not unique. He therefore proposes, not only to consider different levels of detail, but also different views or *slices* [242]. In fact, multiple views of the same model can be used to solve difficult problems. Indeed, exchange of information between different slices allows for the solution of problems which appear hard or impossible in one slice using information derived in another. The solution techniques proposed here do not require redundant information to solve Sussman's hard problems. In our framework views are therefore only used to deduce information concerning the design model and to allow the designer to impose additional functional requirements. Design graphs can in fact be viewed as *metamodels* [249] to integrate the background theories behind the different views.

Graphs are a natural choice to represent the interdependencies in complex engineering design. In [56], it is proposed to use a quality loss criterion to control loss of design freedom in associated decision process. As we use bounded models instead of estimation models, our framework allows for maximum design freedom compatible with previous decisions. An architecture for integrating engineering idealizations in a unified framework is proposed by [124]. However, the use of approximate point models frequently leads to significant errors resulting in wrong decisions [243]. Furthermore, unlike the conservative error control used in this thesis, traditional error estimation provides no guarantee concerning the accuracy of the results, which ultimately need to be verified manually. In this context, it should be noted that the robust numerical methods proposed here considerably extend the applicability of truth maintenance systems [50].

Modeling and database techniques for creating, manipulating and analyzing models of the designed structure are used extensively and have drawn considerable attention of the research community. For recent contributions on the topological aspects of mixed dimensional modeling systems, we refer the reader to [261, 208, 83, 28]. An overview of the current state of the art in geometry processing can be found in [14] while some of the outstanding geometric issues in Computer Aided Design are presented in [37]. Geometry has also extensively been used in the area of automatic analysis, the medial axis techniques [82] in particular, have proven to be very promising. More recently, researchers have attempted to integrate current geometric capabilities in so called *geometric reasoning* systems [268, 197, 269]. While this research furthers our understanding of geometric modeling and analysis, it does, in our opinion, not provide an appropriate organizational framework needed for design automation.

2.4.3 Computation in Design

The computational aspects of the design process encompass a large number of different areas, including optimization, numerical analysis, computer science, and artificial intelligence. Rather then being exhaustive, we will therefore only review the literature relevant to our framework for design.

Only very few general design methods are based on global optimization techniques [244], whereas they have been used extensively in the area of VLSI design [121]. In our opinion, one of the reasons for the limited success of optimization techniques is that general design problems, as opposed to VLSI design, are usually very coupled [200]. Other shortcomings of mechanical design automation are cited in [66]. The area of optimization for design has however enjoyed increased attention in recent years.

For an overview of well known general optimization algorithms we refer the reader to [74], while the specific aspects of translating design problems into constrained optimization problems can be found in [198]. In addition, a number of powerful methods have been developed for shape optimization based on discretization techniques, see for example [256] or [187] for the specific case of elliptic systems. Unfortunately, little work has been done to incorporate optimization techniques in practical computer languages as is proposed in [147].

Papalambros and Wilde suggest that a large number of engineering optimization problems can be solved using *monotonicity analysis* [181]. They derive rigorous analytic formulas for a large number of design optimization problems and are able to show how it can justify engineering design practice. For problems which to not exhibit monotonic behavior, this approach can be combined with numerical analysis techniques [91]. Approximate optimization algorithms need however often to be combined with multistart methods to increase their reliability [109]. A graphic interpretation of problem of attraction basins for optimization is given [35]. While optimization problems are not directly addressed in this thesis, it should be noted that one can simply use the nonlinear solvers of chapter 3 for their solution and that the efficient techniques of chapter 5 can be extended to obtain second order derivatives.

Many designers argue that one really wants to optimize different criteria at the same time. Multiobjective optimization methods, however, make implicit assumptions on the relative importance of the various objectives. We believe that this should be left for the designer to decide under the form of functional requirements.

Combinatorial explosion in large scale design optimization problems can be handled effectively using hierarchical decomposition by relating subproblem objectives to the original optimization problem [114]. A number of numerical techniques for automatic decomposition techniques are described in [22]. Design however is an incremental process and care should be taken to adapt the objective function as more information becomes available. While some methods do indeed allow for incremental changes in the optimization process [42, 75], they do in our opinion not provide the reliability required for design automation. Instead we propose to rely on procedural consistency for incremental selection of the design model.

In general one can view nonlinear algebraic equations as constraints. This approach is suggested by Sussman [242] who uses constraints as fundamental building blocks and constraint propagation as basic model of computation. In addition, his *constraints* language allows for propagation in almost hierarchical systems. To overcome the limitations of the basic propagation algorithm, Serrano extends this work by incorporating numerical solution algorithms to solve strongly connected components and by including techniques for handling inequality constraints [218, 219]. Other hierarchical approaches related to constraint satisfaction have been proposed, see for example [257, 199]. As indicated earlier, one of the contributions of this thesis is to address the robustness problems associated with these numerical solution algorithms.

Some researchers view design as exploring constraints [79] and use constraints to represent design knowledge [80]. Indeed, in many ways models can be viewed as constraints. In this context, we would like to emphasize the importance of automatic differentiation methods. The additional information they derive is not only required for nonlinear solution techniques, but can also be used for the automatic incorporation of changes and to provide an indication as to the relative importance of the variables involved. Constraints can also be used as a computational model for combinatorial problems. A good account of constraint satisfaction and optimization methods for this type of problems can be found in [255]. Finally, it should be noted that particular solution techniques can be selected based on the type of constraints [31].

Constraint propagation techniques were recently extended to include intervals which can represent sets of artifacts [259]. Unfortunately, as the associated iterations terminate quite rapidly [212], these techniques do not allow for the computation of very tight bounds. More importantly, interval propagation algorithms are limited to directed acyclic graphs and can therefore not handle strongly connected components. Since these methods can be viewed as interval evaluation, one can in fact use the more sophisticated enclosure methods described in chapter 3. Finally, the labeling of intervals, used to better describe the constraints associated with the set, can, as we have indicated earlier, significantly be simplified.

Fuzzy set calculus [267] extends interval propagation by associating preferences to individual solutions. Unfortunately, this technique cannot handle coupled systems, moreover the computation of the associated preferences is of combinatorial complexity. In our view, designer preferences concerning particular solutions should explicitly be expressed as functional requirements. While the probabilistic approach [93, 225] is much more powerful and probabilistic analysis is needed to model stochastic uncertainty, it is in our opinion inappropriate for modeling the imprecision uncertainty associated with the selection of alternatives in design.

A very appealing idea is to use continuously decreasing Lyapunov [148] functions for stabilizing nonlinear design systems, including large organizations of intelligent agents [237]. Unfortunately, we were not able to find any literature on applications of this method in the area of design and therefore refer the reader to [17], who uses the dynamic systems metaphor to provide a general framework for design.

Freeman [64] was one of the first to address the problem of design automation from the artificial intelligence perspective. He uses automatic reasoning techniques to connect structures providing certain functions with the ones requiring them. Models therefore directly represent design knowledge, hereby allowing for automatic specification of functional requirements. The knowledge representation and reasoning techniques used in expert systems are in fact very similar, see for example [120]. As indicated earlier, explicit association of design knowledge with models and the corresponding inferences are however bound to be highly problem specific. A more powerful approach was presented by Brown [32], who extends the expert systems technology by providing specific contexts for design knowledge and uses a hierarchical control structure to coordinate a number of simple design experts. Observe that the techniques of chapter 5 can be used to automatically provide these experts with required derivative information. In fact, some of the design expert reasoning techniques are a simplification of approximate nonlinear solution operators.

In addition, researchers in the field of qualitative reasoning are attempting to apply some of their findings to the field of design [62]. In our framework qualitative knowledge is represented by heuristic models, which typically do not have very tight bounds. It was indicated earlier that the use of these models is not compromising the validity of the results as long as more refined models are procedurally consistent.

Frameworks for expert system cooperation [76, 73] and environments to support team based design [262, 235] often use so called *blackboard architectures* [57]. This very popular centralized representation mechanism has in fact been proposed as a fundamental tool for design automation [49]. Any form of centralized control or representation does however form a computational bottleneck in a distributed concurrent problem solving architecture [145]. Indeed, the growing area of distributed problem solving really focuses on coordination of participating agents to achieve global coherence. Early distributed problem solving systems [123] are based on concurrent *Planner*-like architectures with fine grain parallelism. More elaborate meta-level control schemes to achieve global coherence have been suggested by [39] and [145] who introduces *functionally accurate* cooperative distributed systems to deal error-recovery and robustness. A general paradigm for task-sharing based on contract nets and negotiation is proposed by [231, 45]. For more recent work we refer to [52] who introduces a special issue on this subject.

This type distributed problem solving architectures is however relatively coarse grained and typically organized around nodes of a physical network [146]. Moreover, the functionality of the nodes can be compared to human expertise. Since, we are only moderately successful in mimicking human intelligence, let alone in a social context [33], we have proposed to organize systems for design automation as networks of simple computer agents. Furthermore, we believe that to be practical, problem solving architectures should leave implementation aspects to hardware dependent compilers.

A different approach for distributed intelligent systems is based on Petri Nets [185]. While experimental results comparing hierarchical and other decision making architectures using this computational model can be found in [113], further results concerning organization and coordination of distributed decision makers are presented in [182]. We like to point to [209] in particular who discusses the design of simulation languages with multiple modularities based on this approach. In fact, some protocols required for distributed design systems based have already been established based on Petri Nets [58].

2.5 The Design Society

Design is a social process [33] and from CEO to people in the workshop, a large number of people participate in the design process [21]. We therefore propose to use the *Design Society* metaphor to view systems for design automation as large organizations of simple agents that cooperate coherently to solve design problems. The word society is borrowed from [159] who suggests that human intelligence can best be explained in terms of a society of simple agents. As opposed to human agents these simple agents do not understand how other agents work, they only know how to exploit them. The focus is therefore on integration of different types of knowledge to create a system that appears to be smart.

A design society is made up of large number of different computer agents that solve simple routine subproblems which were previously solved by individual engineers. These agents are organized in a hierarchical manner and are working concurrently to achieve the same goal. Indeed, in *The Architecture of Complexity* [228, 229] Simon points out that among possible systems of a given size and complexity, hierarchical systems are the most likely to appear through evolutionary processes and require much less information transmission among their parts than do other types of systems. The short run behavior of subproblem agencies is therefore approximately independent of other agencies, whereas their long run behavior depends on them only in an aggregate way.

Design agents are influenced by their superiors, colleagues and by their subordinates. The respective influences are determined by the designers requirements and depending on the particular design philosophy, a design society will show more resemblance to a centralized bureaucracy then to a democracy. Furthermore, by charging for processor time, communication cost and memory requirements, explicit compromises can be made between performance and limited resources.

The model selection process based on designers requirements corresponds to activation of Minsky's K-lines and can for example be implemented using the bucket brigade algorithm [75] or activation spreading mechanism [150]. Model refinements and the corresponding level shifts operate very much in the same fashion as the *Subsumption Architecture* [29].

Throughout the design process design agents cooperate with each other by sharing results concerning bounded models. We therefore propose that the design model will be organized around agents of the design society and will be represented in a distributed fashion. Indeed, since they are the originators of information they also are the best keepers of it. The meaning of a design model is therefore defined through the agencies making up the model, for only they can answer queries and analyze dependencies. Furthermore depending on which agency is queried, different views of the same design can be obtained. Indeed, in our opinion, only a small fraction of knowledge is static, and intelligence really has to be associated with processes [30]. We therefore postulate that intelligent design behavior can emerge from interaction simple minded agents and agencies in a natural fashion [100].

We have decided to use the term behavior because of the basic adaptation capabilities of agents in the design society, for automatic solution techniques and incremental specification make design a dynamic rather then a static process. Furthermore, we believe that only a hybrid symbolical–numerical approach [251] can overcome the robustness problems associated with rule based systems [43, 44]. We therefore expect that inconsistency and indeterminacy will cause the performance of the design society to deteriorate gracefully, instead of causing total failure.

The design society also can be used as a metaphor for devising collaborative work platforms. Indeed, a large number of people can interact with local computer agents who are in communication with other agencies over a large network [180]. This approach therefore allows for the use of different languages and compilers as long as the interface to other agencies remains the same. Furthermore, the authority to make design changes can be granted only to a specific agency that can reject individual requests for changes based on consistency considerations. Specialists can be provided different views on the design model by drawing information from appropriate agencies. We could also make a distinction between designers and modelers. Designers decide the requirements of the product, while modelers input new technology and know–how. In this context, partly automated modeling techniques can be allowed, where part of the communication is handled by computer agents, while other messages are forwarded to engineers.

Although it is highly desirable to pick the *better* alternative, it is usually not possible to find the *best* one. Indeed, the hierarchical aspects of our framework are of a heuristic nature and bounded models therefore enclose only *satisfycing* solutions [230]. Finally, in *Administrative Behavior* [227], Simon studies rational decision making in organizations and suggests that two rational agents provided with the same information, can rationally only reach the same decision. He concludes that behavior is therefore not determined by rationality but rather by the limits of rationality. In this thesis we therefore set out to study computational methods for reasoning with information bounds.

Chapter 3

Interval Arithmetic

3.1 Motivation

We are concerned with the automation of the design of complex structures. In this context interval arithmetic is an excellent computational tool for two reasons. First, full automation requires the 100% reliability provided by interval arithmetic, and second the inclusion monotonicity properties are extremely useful for solving the consistency problems associated with the hierarchical design methodology. In addition, it allows for concurrency in a natural fashion, as performance degrades gracefully with excessive interval widths.

We believe that robust solution methods for nonlinear algebraic equations are essential in design automation, and in the present chapter we are therefore investigating this problem.

Although some very reliable techniques have recently been developed for the solution of polynomial equations [138, 184], these methods are currently limited to 2–dimensional problems and do not apply in the case of general nonlinear equations. Classical numerical algorithms are applicable [41, 252], but they do not guarantee that all the solutions will be found, nor is there any guarantee of their accuracy. Moreover, user action is sometimes required in case of singularities or inappropriate starting points.

Instead of using point approximations to the solution, feasible domains can be described by intervals. In the literature, computations involving intervals have been have been referred to as *Interval Arithmetic*. Interval computations, when performed conservatively, are 100% reliable. Numerical solution methods based on these techniques can therefore lead to further automation.

3.2 Basics

Interval arithmetic can be traced back to Young's arithmetic for calculation with sets of numbers, developed in 1931 [273], and later to Dwyer, who considered the special case of closed intervals [53]. The first practical applications, however, date back to Moore in 1959 [160]. And since the mid 1960 interval arithmetic has been an area of active research [86, 162]. There exists a large literature on interval techniques, it covers subjects such as inclusions of the ranges of functions [193], linear interval equations [6], inclusion of zeros complex rational polynomials [186], and optimization [67, 194]. More recently various interval arithmetic packages became available for general use [140, 165].

In this section we give definitions and explain the basic principles of interval arithmetic. We have attempted to use simple notation and terminology consistent with the literature [7, 164, 171, 193, 194].

3.2.1 Definitions

An *interval* $X \in \mathbf{I}$ is a set of numbers $x \in \mathbf{R}$ defined by:

$$X = \{x \mid X_l \leq x \leq X_u\} \tag{3.1}$$

$\mathbf{I}$ is the set of intervals and $\mathbf{R}$ is the set of real numbers. Capital letters denote interval quantities, whereas lowercase letters correspond to numbers. Subscripts are used to indicate entities related to an interval. X_l stands for the *lower bound* of X and X_u for the *upper bound*. When we explicitly refer to the bounds of an interval we will use the following notation:

$$X = [X_l\,,\, X_u]$$

The traditional definition of the *midpoint* of an interval is:

$$X_c = \frac{X_l + X_u}{2}$$

although, for implementation purposes it sometimes is defined as the median of the floating point enumerations between X_l and X_u. When an interval is centered around its midpoint, we have:

$$\Delta X = X - X_c$$

which is a symmetric interval. The *interior* of an interval corresponds with:

$$i(X) = \Box\{x \mid X_l < x < X_u\}$$

$\Box$ indicates the convex hull operation to ensure that the result can be represented with intervals. Finally, *width* is defined as interval norm:

$$w(X) = X_u - X_l$$

and *distance* as interval metric [162, 169]:

$$d(X\,,\, Y) = \max\{|Y_l - X_l|, |Y_u - X_u|\}$$

Note that we use the following convention when referring to absolute values of intervals [168]:

$$\begin{aligned} \langle X \rangle &= \min\{|x| \mid x \in X\} \\ |X| &= \max\{|x| \mid x \in X\} \end{aligned}$$

3.2.2 Operators

Interval arithmetic operations are defined by applying the corresponding algebraic operators to the elements of the interval operands, hereby generating a new interval:

$$X \circ Y = \Box\{x \circ y \mid x \in X, y \in Y\} \tag{3.2}$$

For the usual algebraic operators $\circ \in \{+, -, \cdot, /\}$. Based on this definition we can derive the following explicit expressions:

$$\begin{array}{lcl} [X_l\,,\,X_u]+[Y_l\,,\,Y_u] & = & [X_l+Y_l\,,\,X_u+Y_u] \\ [X_l\,,\,X_u]-[Y_l\,,\,Y_u] & = & [X_l-Y_u\,,\,X_u-Y_l] \\ [X_l\,,\,X_u]\cdot[Y_l\,,\,Y_u] & = & [\min(X_lY_l, X_lY_u, X_uY_l, X_uY_u)\,,\,\max(X_lY_l, X_lY_u, X_uY_l, X_uY_u)] \\ [X_l\,,\,X_u]/[Y_l\,,\,Y_u] & = & [X_l\,,\,X_u]\cdot[1/Y_u\,,\,1/Y_l] \end{array} \tag{3.3}$$

As is the case with the multiplication of numbers, we will usually omit the $\cdot$ notation.

The definition of set operations on intervals is straightforward, interval intersection and union, for example are defined as follows:

$$\begin{array}{rcl} X \cap Y & = & \{z \mid z \in X \; and \; z \in Y\} \\ X \cup Y & = & \{z \mid z \in X \; or \; z \in Y\} \end{array}$$

The result of the set operations are not necessarily intervals, an intersection can be the empty set ϕ.

The transition to machine interval arithmetic is made by conservative rounding of the bounds, so that $X \subseteq X_M$ and by ensuring that all machine operations satisfy the inclusion principle:

$$\left.\begin{array}{c} x \in X \\ y \in Y \end{array}\right\} \Rightarrow x \circ y \in (X \circ Y)_M$$

Practical aspects of implementing interval arithmetic on a computer are discussed in [141].

3.2.3 Properties

Most properties of algebraic operators carry over to their interval definition, such as commutativity:

$$\begin{array}{rcl} X + Y & = & Y + X \\ XY & = & YX \end{array}$$

and associativity:

$$\begin{array}{rcl} X + (Y + Z) & = & (X + Y) + Z \\ X(YZ) & = & (XY)Z \end{array}$$

Distributivity however, is not satisfied in general, but we do have the following *subdistributivity* property:

$$X(Y + Z) \subseteq XY + XZ$$

An interval function $F(X)$ is said to be *inclusion monotone* if:

$$X \subseteq Y \;\Rightarrow\; F(X) \subseteq F(Y) \tag{3.4}$$

For an interval operator $\circ$ it corresponds to:

$$\left.\begin{array}{l} X' \subseteq X \\ Y' \subseteq Y \end{array}\right\} \Rightarrow X' \circ Y' \subseteq X \circ Y$$

One easily verifies that the explicit expressions (3.3) for $\circ \in \{+, -, \cdot, /\}$ satisfy the above relation.

3.3 Enclosures for the Range of a Function

The challenge of interval arithmetic is to find bounds that are as tight as possible. This problem really is fundamental in that any pointwise dependency of variables is lost when interval representations are used.

Methods to compute an inclusion of the range of functions and their derivatives, can be found in the literature [162, 193]. The simplest method is to use interval evaluation, replacing all variables by their intervals and arithmetic operations by the equivalent interval operations. This can be used for functions as well as for their derivatives after explicit differentiation. Better inclusions are obtained with centered forms. They are often used in conjunction with the more efficient recursive differentiation or Taylor expansion techniques. Recently, Neumaier defined the concept of inclusion algebra [171] to unify different recursive interval techniques. We have chosen to formally define these recursive systems using *Denotational Semantics* [214, 239]. This approach allows us to define a more complete language, including procedures and higher level constructs.

3.3.1 Definitions

The *range* of a function is defined as:

$$f(X) = \Box\{f(x) \mid x \in X\}$$

Note that we use a different notation for the range as for an interval enclosure function of that range:

$$f(X) \subseteq F(X)$$

Example 3.1

As illustrated in figure 3.1, $f_1(x) = 1 - 5x + x^3/3$ has range $f_1(X) = [1 - 10\sqrt{5}/3\,,\, -5]$ over $X = [2\,,\,3]$

An interval function $F : \mathbf{I} \to \mathbf{I}$ is *Lipschitz continuous* over X if:

$$\forall X', X'' \subseteq X\,,\ \exists L \in \mathbf{R}\ :\ d(F(X')\,,\, F(X'')) \leq L d(X'\,,\, X'') \tag{3.5}$$

the real number L is called the *Lipschitz constant*. An interval enclosure function over X is of order $\alpha > 0$ when:

$$\forall X' \subseteq X\,,\ \exists A \in \mathbf{R}\ :\ d(f(X')\,,\, F(X')) \leq A w(X')^{\alpha}$$

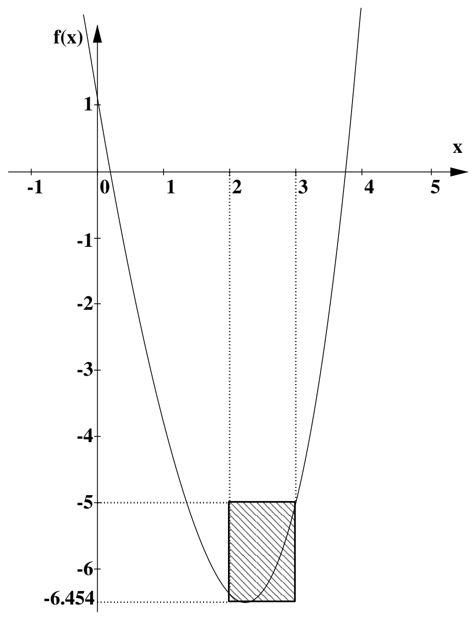

Figure 3.1: Range of $1 - 5x + x^3/3$ over $X = [2\,,\,3]$

The definitions for multivariable functions are very similar. The scalar quantities simply need to be replaced by vector interval quantities, see section 3.4 below.

Example 3.2

The range of $f_2(x, y) = e^{1/(x^2+y^2)}$ over $X = [1\,,\,2]$ and $Y = [2\,,\,4]$ is $f_2(X, Y) = [1.051\,,\,1.222]$

3.3.2 Interval Evaluation

Enclosures of arithmetic expressions can be simply be computed by interval evaluation [162], i.e. by replacing all quantities by their interval equivalents.

Example 3.3

The interval evaluation of $f_1(x)$ of example 3.1 is computed by:

$$\begin{aligned} E(f_1(X), X) &= 1 - 5X + 1/3XXX \\ &= [-11.334\,,\,0] \end{aligned} \tag{3.6}$$

over $X = [2\,,\,3]$. Note the conservative rounding of the lower bound -34/3 to -11.334.

To define this interval evaluator with denotational semantics, we need the following domains:

$v \in$ *Expressible-Value* $= \{$ *Interval* $+$ *Procedure* $\}$

$u \in$ *Value-Environment* $=$ *Identifier* $\mapsto$ *Expressible-Value*

the valuation functions are:

$$\mathcal{C} \ : \mathit{Constant} \mapsto \mathit{Expressible\text{-}Value}$$

$$\mathcal{E} \ : \mathit{Expression} \mapsto \mathit{Value\text{-}Environment} \mapsto \mathit{Expressible\text{-}Value}$$

$\mathcal{C}$ maps constants into expressible values in the obvious way and $\mathcal{E}$ is defined as follows:

$$\mathcal{E}[\![C]\!] \mapsto \lambda u.\mathcal{C}[\![C]\!]$$

$$\mathcal{E}[\![I]\!] \mapsto \lambda u.u[\![I]\!]$$

$$\mathcal{E}[\![\texttt{(+ } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.(\mathcal{E}[\![E_1]\!]u + \mathcal{E}[\![E_2]\!]u)$$

$$\mathcal{E}[\![\texttt{(- } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.(\mathcal{E}[\![E_1]\!]u - \mathcal{E}[\![E_2]\!]u)$$

$$\mathcal{E}[\![\texttt{(* } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.(\mathcal{E}[\![E_1]\!]u \cdot \mathcal{E}[\![E_2]\!]u)$$

$$\mathcal{E}[\![\texttt{(/ } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.(\mathcal{E}[\![E_1]\!]u / \mathcal{E}[\![E_2]\!]u)$$

$$\mathcal{E}[\![\texttt{(lambda (}I\texttt{) } E\texttt{)}]\!] \mapsto \lambda u.\lambda v.\mathcal{E}[\![E]\!](\mathit{bind}\, v[\![I]\!]u)$$

$$\mathcal{E}[\![\texttt{(}E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\mathcal{E}[\![E_1]\!]u(\mathcal{E}[\![E_2]\!]u)$$

The auxiliary function used to bind an identifier in the environment is:

$$\mathit{bind} = \lambda v.\lambda I.\lambda u.\lambda I'.(I = I') \mapsto v \, [\!] \, uI'$$

Example 3.4

For expression (3.6) of example 3.3 the interval evaluator would be used as follows:

$$\mathcal{E}[\![\texttt{(+ (- 1 (* 5 x)) (* 1/3 (* x (* x x))))}]\!](\mathit{bind}\, [\![\mathrm{x}]\!][2\,,\,3]u_0) = [-11.334\,,\,0]$$

where u_0 stands for the initial environment.

Note that this type of recursive evaluation can be done in many different orders and the question arises as to the optimality of one evaluation strategy w.r.t. another, for example:

$$X(1-X) \subseteq X - XX$$

because of subdistributivity. Theoretically, the best computable enclosure is the intersection of all possible equivalent computations. This issue is similar to ordering floating point operations. Various strategies for obtaining optimal accuracy are discussed in [122, 193].

Based on these definition we can also introduce curried arguments:

$$\mathcal{E}[\![\texttt{(lambda (}I\ I*\texttt{) } E\texttt{)}]\!] \mapsto \mathcal{E}[\![\texttt{(lambda (}I\texttt{)(lambda (}I*\texttt{) } E\texttt{))}]\!]$$

$$\mathcal{E}[\![\texttt{(}E_0 \ E_1 \ E*\texttt{)}]\!] \mapsto \mathcal{E}[\![\texttt{((}E_0 \ E_1\texttt{) } E*\texttt{)}]\!]$$

and define the `let`-form as a syntactic sugar:

$$\mathcal{E}[\![\texttt{(let ((I } E\texttt{) } B*\texttt{) } E_0\texttt{)}]\!] \mapsto \mathcal{E}[\![\texttt{((lambda (}I\texttt{) (let (}B*\texttt{) } E_0\texttt{)) } E\texttt{)}]\!]$$

This evaluator can be complemented with natural interval extensions of elementary functions:

$\mathcal{E}_{\mathcal{F}}$: *Function* $\mapsto$ *Procedure*

The exponential function for example, could be included by the following definitions:

$\mathcal{E}[\![\texttt{(exp } E\texttt{)}]\!] \mapsto \lambda u.\mathcal{E}_{\mathcal{F}}[\![\texttt{exp}]\!](\mathcal{E}[\![E]\!]u)$

$\mathcal{E}_{\mathcal{F}}[\![\texttt{exp}]\!] \mapsto \lambda v.[\mathrm{e}^{v_l}\, ,\, \mathrm{e}^{v_u}]$

Let us illustrate these extensions with an example.

Example 3.5

The computation of $f_2(x, y)$ of example 3.2 is expressed as follows:

$\mathcal{E}[\![$
```
(let ((square (lambda (v) (* v v))))
  (exp (/ 1 (+ (square x) (square y)))))
```
$]\!]$

$$(\mathit{bind}\,[\![\mathtt{x}]\!][1\, ,\, 2](\mathit{bind}\,[\![\mathtt{y}]\!][2\, ,\, 4]u_0)) = [1.051\, ,\, 1.222]$$

which is the exact value of the range.

If the extensions are Lipschitz continuous, it can be shown that extended interval evaluation is also Lipschitz continuous [160, 164]. The enclosures produced by this interval evaluation are of order 1 and therefore of lesser quality. Nevertheless, they are essential for more sophisticated methods. Also, when w(X) is large, interval evaluation often produces more effective bounds than centered forms. In fact, their quadratic convergence implies that the width of the enclosure can grow quadratically.

3.3.3 Centered Forms

The centered form of a function $f(x)$ with center c is an expression of the type:

$$C(f, X, c) = f(c) + G(X - c)(X - c)$$

and was originally introduced by Moore in [162]. This form is of interest because it is of order 2 when G is Lipschitz continuous [137]. Centered forms of rational polynomials were studied by Ratschek and Rokne in [193]. The general *Taylor form* [162, 191, 192] is defined by:

$$T_k(f, X, c) = f(c) + \sum_{j=1}^{k-1} \frac{f^{(j)}(c)}{j!}(X - c)^j + \frac{F^{(k)}(X)}{k!}(X - c)^k$$

The special case $k = 1$ is called the *Mean Value form*:

$$T_1(f, X, c) = f(c) + F'(X)(X - c)$$

If $F'(X)$ is Lipschitz continuous this form has quadratic convergence.

Example 3.6

When applied to $f_1(x)$ as defined in example 3.1:

$$T_1(f_1, [2\, ,\, 3], 5/2) = [-8.292\, ,\, -4.291]$$

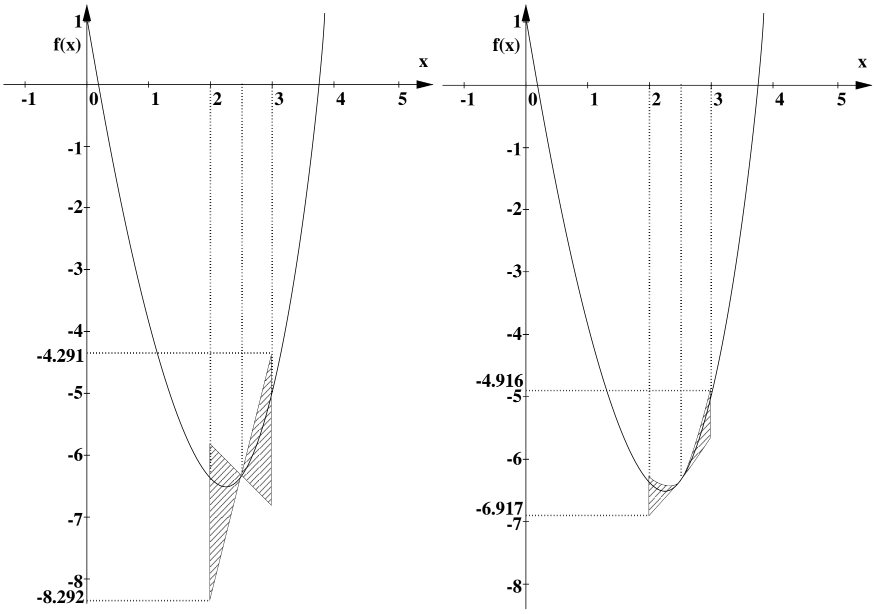

Figure 3.2: Geometric interpretation of the centered forms for the function f_1 of example 3.1

The value of the interval derivative $F'(X)$ was obtained by interval evaluation of $f'(x)$ over X. A geometric interpretation of the enclosure produced by the mean value form is given in figure 3.2. The shaded area represents the feasible domain for the given $F'(X)$.

Another very useful Taylor form is the second order one:

$$T_2(f, X, c) = f(c) + f'(c)(X - c) + \frac{F''(X)}{2}(X - c)^2$$

for it can be shown that the boundedness of $F''(X)$ is sufficient to ensure second order convergence. Note that the computation of $(X - c)^n$ can be done exactly by using an interval extension of the power function.

Example 3.7

The second order Taylor form improves on the bound of example 3.6, indeed:

$$\begin{aligned} T_2(f_1, [2\,,\,3], 5/2) &= [-6.917\,,\,-4.916] \\ &\subseteq T_1(f_1, [2\,,\,3], 5/2) \end{aligned}$$

This is illustrated in figure 3.2, again $F''(X)$ is computed by interval evaluation of $f''(x)$ over X.

The convergence of the higher order Taylor forms remains of order 2. The inclusion they provide however, is usually increasingly better. If, for example, the higher order derivatives are enclosed by the mean value form, we have:

$$T_k(f, X, c) \subseteq T_{k-1}(f, X, c)$$

Example 3.8

If we use:

$$\begin{aligned} E(f_1'', [2\,,\,3]) &= [4\,,\,6] \\ T_1(f_1', [2\,,\,3], 5/2) &= [-1.75\,,\,4.25] \end{aligned}$$

in the computation of the following enclosures:

$$\begin{aligned} T_1(f_1, [2\,,\,3], 5/2) &= [-8.417\,,\,-4.166] \\ T_2(f_1, [2\,,\,3], 5/2) &= [-6.917\,,\,-4.916] \end{aligned}$$

we have:

$$T_2(f_1, [2\,,\,3], 5/2) \subseteq T_1(f_1, [2\,,\,3], 5/2)$$

Again, these definitions can simply by extended to multivariate functions by using interval vectors defined in section 3.4.

3.3.4 Interval Differentiation

Very often, as is the case in the mean value form, it is required to evaluate the range of derivatives. The process of computing the derivative and at the same time evaluating the range of this derivative time is called interval differentiation. A good account of recursive interval differentiation is given by Rall [190]. To formally define a recursive interval differentiator, we introduce some additional semantic domains:

$d \in$ *Expressible-Differential*

$g \in$ *Differential-Environment* $=$ *Identifier* $\mapsto$ *Expressible-Differential*

the valuation function is:

$\mathcal{D}$: *Expression* $\mapsto$ *Value-Environment* $\mapsto$ *Differential-Environment* $\mapsto$ *Expressible-Differential*

$$\mathcal{D}[\![C]\!] \mapsto \lambda u.\lambda g.zero$$

$$\mathcal{D}[\![I]\!] \mapsto \lambda u.\lambda g.g[\![I]\!]$$

$$\mathcal{D}[\![\texttt{(+}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda g.(\mathcal{D}[\![E_1]\!]ug + \mathcal{D}[\![E_2]\!]ug)$$

$$\mathcal{D}[\![\texttt{(-}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda g.(\mathcal{D}[\![E_1]\!]ug - \mathcal{D}[\![E_2]\!]ug)$$

$$\mathcal{D}[\![\texttt{(*}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda g.(\mathcal{D}[\![E_1]\!]ug \cdot \mathcal{E}[\![E_2]\!]u + \mathcal{E}[\![E_1]\!]u \cdot \mathcal{D}[\![E_2]\!]ug)$$

$$\mathcal{D}[\![\texttt{(/}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda g.(\mathcal{D}[\![E_1]\!]ug - \mathcal{E}[\![\texttt{(/}\ E_1\ E_2\texttt{)}]\!]u \cdot \mathcal{D}[\![E_2]\!]ug)/\mathcal{E}[\![E_2]\!]u$$

$$\mathcal{D}[\![\texttt{(lambda (}I\texttt{)}\ E\texttt{)}]\!] \mapsto \lambda u.\lambda g.\lambda v.\lambda d.\mathcal{D}[\![E]\!](bind\, v[\![I]\!]u)(bind\, d[\![I]\!]g)$$

$$\mathcal{D}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda g.\mathcal{D}[\![E_1]\!]ug\,(\mathcal{E}[\![E_2]\!]u)(\mathcal{D}[\![E_2]\!]ug)$$

Example 3.9

As in example 3.4, the first interval derivative is simply obtained by:

$$\mathcal{D}[\![\texttt{(+ (- 1 (* 5 x)) (* 1/3 (* x (* x x))))}]\!](\mathit{bind}\,[\![\mathrm{x}]\!][2\,,\,3]u_0)(\mathit{bind}\,[\![\mathrm{x}]\!][1]d_0) = [-1\,,\,4]$$

u_0 is again the initial value environment and d_0 is the initial derivative environment. As we noted earlier, interval evaluation sometimes produces better enclosures than more sophisticated methods. Indeed, in this example recursive differentiation is better than the mean value form computed in example 3.8:

$$\begin{array}{rcl} D(f_1, [2\,,\,3]) & = & E(f_1', [2\,,\,3]) \\ & \subseteq & T_1(f_1', [2\,,\,3], 5/2) \end{array}$$

Introduction of elementary functions proceeds as with the interval evaluator:

$$\mathcal{D}[\![\texttt{(exp } E\texttt{)}]\!] \mapsto \lambda u.\lambda g.\mathcal{D}_{\mathcal{F}}[\![\texttt{exp}]\!](\mathcal{E}[\![E]\!]u)(\mathcal{D}[\![E]\!]ug)$$

$$\mathcal{D}_{\mathcal{F}}[\![\texttt{exp}]\!] \mapsto \lambda v.\lambda d.[\mathrm{e}^{v_l}\,,\,\mathrm{e}^{v_u}]\cdot d$$

Example 3.10

If we go back to example 3.5 we now have:

$\mathcal{D}[\![$
```
(let ((square (lambda (v) (* v v))))
  (exp (/ 1 (+ (square x) (square y)))))
```
$]\!]$

$$(\mathit{bind}\,[\![\mathrm{x}]\!][1\,,\,2](\mathit{bind}\,[\![\mathrm{y}]\!][2\,,\,4]u_0))(\mathit{bind}\,[\![\mathrm{x}]\!][1](\mathit{bind}\,[\![\mathrm{y}]\!][0]d_0)) = [-0.196\,,\,-0.005]$$

$\mathcal{D}[\![$
```
(let ((square (lambda (v) (* v v))))
  (exp (/ 1 (+ (square x) (square y)))))
```
$]\!]$

$$(\mathit{bind}\,[\![\mathrm{x}]\!][1\,,\,2](\mathit{bind}\,[\![\mathrm{y}]\!][2\,,\,4]u_0))(\mathit{bind}\,[\![\mathrm{x}]\!][0](\mathit{bind}\,[\![\mathrm{y}]\!][1]d_0)) = [-0.391\,,\,-0.011]$$

which can be used in the multivariate mean value form:

$$T_1(f_2, ([1\,,\,2]\,[2\,,\,4])^T, ([3/2]\,[3])^T) = [0.604\,,\,1.582]$$

3.3.5 Implementation

We illustrate the above semantic concepts with *Scheme* code of a similar evaluator. For efficiency both interval evaluation and differentiation are performed at the same time:

```
(define general-eval (expression environment)
  (cond ((number? expression)
         (number-eval expression))
        ((variable? expression)
         (variable-eval expression environment))
        ((primitive? expression)
         (primitive-eval expression environment))
        ((lambda? expression)
         (lambda-eval expression environment))
        (t
         (application-eval expression environment))))
```

```
(define number-eval (expression)
  (let ((value-interval (point-interval expression))
        (derivative-interval (point-interval 0)))
    (make-tuple value-interval derivative-interval)))

(define variable-eval (expression environment)
  (lookup expression environment))

(define primitive-eval (expression environment)
   (let* ((operator (primitive-operator expression))
          (operand1 (primitive-operand1 expression))
          (operand2 (primitive-operand2 expression))
          (tuple1 (general-eval operand1 environment))
          (tuple2 (general-eval operand2 environment)))
          (value1 (tuple-value tuple1))
          (value2 (tuple-value tuple2))
          (derivative1 (tuple-derivative tuple1))
          (derivative2 (tuple-derivative tuple2)))
      (cond ((+? operator) (+tuple value1 value2 derivative1 derivative2)
            ((-? operator) (-tuple value1 value2 derivative1 derivative2))
            ((*? operator) (*tuple value1 value2 derivative1 derivative2))
            ((/? operator) (/tuple value1 value2 derivative1 derivative2)))))

(define lambda-eval (expression environment)
  (let ((identifier (lambda-identifier expression))
        (body (lambda-body expression)))
    (lambda (tuple)
      (general-eval body (bind identifier tuple environment)))))

(define application-eval (expression environment)
  (let ((operator (application-operator expression))
        (operand (application-operand expression)))
    ((general-eval operator environment)
     (general-eval operand environment))))
```

The primitive operations are defined as follows:

```
(define +tuple (value1 value2 derivative1 derivative2)
  (make-tuple (i+ value1 value2) (i+ derivative1 derivative2)))
```

```
(define -tuple (value1 value2 derivative1 derivative2)
  (make-tuple (i- value1 value2) (i- derivative1 derivative2)))

(define *tuple (value1 value2 derivative1 derivative2)
  (make-tuple (i* value1 value2)
              (i+ (i* derivative1 value2) (i* value1 derivative2))))
```

```
(define /tuple (value1 value2 derivative1 derivative2)
  (let ((value1/value2 (i/ value1 value2)))
    (make-tuple value1/value2
                (i/ (i- derivative1 (i* value1/value2 derivative2)) value2))))
```

Where `i+`, `i-`, `i*` and `i/` are the arithmetic interval operators. To be complete we include a simple implementation of the auxiliary functions:

```
(define bind (identifier tuple environment)
  (lambda (other-identifier)
    (if (equal other-identifier identifier) tuple (environment identifier))))

(define lookup (identifier environment)
  (environment identifier))
```

3.3.6 Higher Order Taylor Forms

In the same spirit as the interval differentiator, higher order Taylor coefficients, needed for the computation of enclosures with higher order, can be computed recursively [162, 164, 191]. Using denotational semantics we can formally define this computation as follows:

$$x, y \in \textit{Variable-Set}$$

$$t \in \textit{Taylor-Environment} = \textit{Identifier} \mapsto \textit{Variable-Set} \mapsto \textit{Expressible-Taylor-Coefficient}$$

$$f \in \textit{Coefficient-Environment} = \textit{Variable-Set} \mapsto \textit{Expressible-Taylor-Coefficient}$$

with valuation function:

$$\mathcal{T} : \textit{Expression} \mapsto \textit{Taylor-Environment} \mapsto \textit{Variable-Set} \mapsto \textit{Expressible-Taylor-Coefficient}$$

$$\mathcal{T}[\![C]\!] \mapsto \lambda t.\lambda x.(x = \phi) \mapsto \mathcal{C}[\![C]\!][\,]\,\textit{zero}$$

$$\mathcal{T}[\![I]\!] \mapsto \lambda t.\lambda x.t[\![I]\!]x$$

$$\mathcal{T}[\![\texttt{(+}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda t.\lambda x.(\mathcal{T}[\![E_1]\!]tx + \mathcal{T}[\![E_2]\!]tx)$$

$$\mathcal{T}[\![\texttt{(-}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda t.\lambda x.(\mathcal{T}[\![E_1]\!]tx - \mathcal{T}[\![E_2]\!]tx)$$

$$\mathcal{T}[\![\texttt{(*}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda t.\lambda x.\textstyle\sum_{i=0}^{n}\mathcal{T}[\![E_1]\!]t(x)_i \cdot \mathcal{T}[\![E_2]\!]t(x\backslash(x)_i)$$

$$\mathcal{T}[\![\texttt{(/}\ E_1\ E_2\texttt{)}]\!] \mapsto \lambda t.\lambda x.(\mathcal{T}[\![E_1]\!]tx - \textstyle\sum_{i=1}^{n}\mathcal{T}[\![E_2]\!]t(x)_i \cdot \mathcal{T}[\![\texttt{(/}\ E_1\ E_2\texttt{)}]\!]t(x\backslash(x)_i))/\mathcal{T}[\![E_2]\!]t\phi$$

$$\mathcal{T}[\![\texttt{(lambda (}I\texttt{)}\ E\texttt{)}]\!] \mapsto \lambda t.\lambda x.\lambda f.\lambda y.\mathcal{T}[\![E]\!](\textit{bind}\, f[\![I]\!]t)y$$

$$\mathcal{T}[\![\texttt{(}E_1\ E_2\texttt{)}]\!] \mapsto \lambda t.\lambda x.(\mathcal{T}[\![E_1]\!]tx)(\mathcal{T}[\![E_2]\!]t)x$$

In the above summations, all subsets $(x)_i$ of x are enumerated starting with the empty set for $\sum_{i=0}^{n}$ and with singletons in the case of $\sum_{i=1}^{n}$.

Example 3.11

To obtain the interval Taylor coefficient used in example 3.7 we simply evaluate:

$$\mathcal{T}[\![\texttt{(+ (- 1 (* 5 x)) (* 1/3 (* x (* x x))))}]\!](\textit{bind}\,[\![\mathtt{x}]\!]_\phi[2\,,3](\textit{bind}\,[\![\mathtt{x}]\!]_{[\![\mathtt{x}]\!]}[1]t_0))\{[\![\mathtt{x}]\!][\![\mathtt{x}]\!]\} = [2\,,3]$$

with initial environment t_0.

An enclosure for the Taylor coefficients of the exponential function, for example, would be:

$$\mathcal{T}[\![\texttt{(exp } E\texttt{)}]\!] \mapsto \lambda t.\lambda x.\mathcal{T}_{\mathcal{F}}[\![\texttt{exp}]\!](\mathcal{T}[\![E]\!]t)x$$

$$\mathcal{T}_{\mathcal{F}}[\![\texttt{exp}]\!] \mapsto \lambda f.\lambda y.(y = \phi) \mapsto [\mathrm{e}^{(f\phi)_l}, \mathrm{e}^{(f\phi)_u}] [\!] \textstyle\sum_{i_1=1}^{n_1} i_1/n_1\, f(y)_{i_1} \cdot \mathcal{T}_{\mathcal{F}}[\![\texttt{exp}]\!] f(y \backslash (y)_{i_1})$$

The index i_1 refers to the first element of y. The following example illustrates the use of the above extension.

Example 3.12

Referring again to the expression of example 3.5, the 2nd order Taylor coefficients are:

$$\begin{aligned}&\mathcal{T}[\![\ \texttt{(let ((square (lambda (v) (* v v))))}\\ &\qquad \texttt{(exp (/ 1 (+ (square x) (square y)))))}\ ]\!]\\ &\qquad\quad (\mathit{bind}\,[\![\mathtt{x}]\!]_\phi[1\,,\,2](\mathit{bind}\,[\![\mathtt{x}]\!]_{[\![\mathtt{x}]\!]}[1]\\ &\qquad\qquad (\mathit{bind}\,[\![\mathtt{y}]\!]_\phi[2\,,\,4](\mathit{bind}\,[\![\mathtt{y}]\!]_{[\![\mathtt{y}]\!]}[1]t_0)))) \{[\![\mathtt{x}]\!][\![\mathtt{x}]\!]\} = [-0.047\,,\,0.160]\end{aligned}$$

$$\begin{aligned}&\mathcal{T}[\![\ \texttt{(let ((square (lambda (v) (* v v))))}\\ &\qquad \texttt{(exp (/ 1 (+ (square x) (square y)))))}\ ]\!]\\ &\qquad\quad (\mathit{bind}\,[\![\mathtt{x}]\!]_\phi[1\,,\,2](\mathit{bind}\,[\![\mathtt{x}]\!]_{[\![\mathtt{x}]\!]}[1]\\ &\qquad\qquad (\mathit{bind}\,[\![\mathtt{y}]\!]_\phi[2\,,\,4](\mathit{bind}\,[\![\mathtt{y}]\!]_{[\![\mathtt{y}]\!]}[1]t_0)))) \{[\![\mathtt{x}]\!][\![\mathtt{y}]\!]\} = [0.002\,,\,0.688]\end{aligned}$$

$$\begin{aligned}&\mathcal{T}[\![\ \texttt{(let ((square (lambda (v) (* v v))))}\\ &\qquad \texttt{(exp (/ 1 (+ (square x) (square y)))))}\ ]\!]\\ &\qquad\quad (\mathit{bind}\,[\![\mathtt{x}]\!]_\phi[1\,,\,2](\mathit{bind}\,[\![\mathtt{x}]\!]_{[\![\mathtt{x}]\!]}[1]\\ &\qquad\qquad (\mathit{bind}\,[\![\mathtt{y}]\!]_\phi[2\,,\,4](\mathit{bind}\,[\![\mathtt{y}]\!]_{[\![\mathtt{y}]\!]}[1]t_0)))) \{[\![\mathtt{y}]\!][\![\mathtt{y}]\!]\} = [-0.040\,,\,0.676]\end{aligned}$$

which gives us the following second order enclosure:

$$T_2(f_2, ([1\,,\,2]\,[2\,,\,4])^T, ([3/2]\,[3])^T) = [0.633\,,\,2.218]$$

which in this case is of lesser quality then the mean value bound.

3.3.7 Interval Slopes

Krawczyk [127] found that if *interval slopes* $S(f, X, c)$, defined by:

$$f(x) - f(c) \in S(f, X, c)(x - c)$$

instead of derivatives are used to develop the centered form:

$$R(f, X, c) = f(c) + S(f, X, c)(X - c)$$

better enclosures are produced than with the mean value form. As is the case for derivatives, interval slopes can be computed recursively. It can be shown [171] that for the slopes computed by the slope evaluator described below:

$$w(S(f, X, c)) = \frac{w(D(f, X))}{2} + \mathcal{O}(w(X)^2) \tag{3.7}$$

In addition, the slopes $S(f, X, c)$ are Lipschitz continuous so that *Krawczyk's centered form* is also of order 2.

For Krawczyk's interval slopes evaluator, we define the following semantic domains:

$p \in \textit{Expressible-Value}$

$t \in \textit{Expressible-Slope}$

$c \in \textit{Center-Environment} = \textit{Identifier} \mapsto \textit{Expressible-Value}$

$s \in \textit{Slope-Environment} = \textit{Identifier} \mapsto \textit{Expressible-Slope}$

The valuation functions is:

$\mathcal{S}$: *Expression* $\mapsto$ *Value-Environment* $\mapsto$ *Center-Environment* $\mapsto$
Slope-Environment $\mapsto$ *Expressible-Slope*

$\mathcal{S}[\![C]\!] \mapsto \lambda u.\lambda c.\lambda s.\mathit{zero}$

$\mathcal{S}[\![I]\!] \mapsto \lambda u.\lambda c.\lambda s.s[\![I]\!]$

$\mathcal{S}[\![\texttt{(+ } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.(\mathcal{S}[\![E_1]\!]ucs + \mathcal{S}[\![E_2]\!]ucs)$

$\mathcal{S}[\![\texttt{(- } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.(\mathcal{S}[\![E_1]\!]ucs - \mathcal{S}[\![E_2]\!]ucs)$

$\mathcal{S}[\![\texttt{(* } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.(\mathcal{E}[\![E_1]\!]u \cdot \mathcal{S}[\![E_2]\!]ucs + \mathcal{S}[\![E_1]\!]ucs \cdot \mathcal{E}[\![E_2]\!]c)$

$\mathcal{S}[\![\texttt{(/ } E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.(\mathcal{S}[\![E_1]\!]ucs - \mathcal{E}[\![\texttt{(/ } E_1 \ E_2\texttt{)}]\!]c \cdot \mathcal{S}[\![E_2]\!]ucs)/\mathcal{E}[\![E_2]\!]u$

$\mathcal{S}[\![\texttt{(lambda (}I\texttt{) }E\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.\lambda v.\lambda p.\lambda t.\mathcal{S}[\![E]\!](\mathit{bind}\, v[\![I]\!]u)(\mathit{bind}\, p[\![I]\!]c)(\mathit{bind}\, t[\![I]\!]s)$

$\mathcal{S}[\![\texttt{(}E_1 \ E_2\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.\mathcal{S}[\![E_1]\!]ucs\,(\mathcal{E}[\![E_2]\!]u)(\mathcal{E}[\![E_2]\!]c)(\mathcal{S}[\![E_2]\!]ucs)$

Example 3.13

We apply this recursive evaluator to expression (3.6) of example 3.3:

$$\mathcal{S}[\![\texttt{(+ (- 1 (* 5 x)) (* 1/3 (* x (* x x))))}]\!]$$
$$(\mathit{bind}\,[\![\mathrm{x}]\!][2\,,\,3]u_0)(\mathit{bind}\,[\![\mathrm{x}]\!][5/2]c_0)(\mathit{bind}\,[\![\mathrm{x}]\!][1]s_0) = [0.083\,,\,2.584]$$

u_0, c_0 and s_0 are the initial value, center and slope environments. With this interval slope Krawczyk's centered form gives us the following enclosure:

$$R(f_1, [2\,,\,3], 5/2) = [-7.584\,,\,-5]$$

which is better than $T_1(f_1, [2\,,\,3], 5/2)$ of example 3.6.

In this case the exponentiation function would be defined as follows:

$\mathcal{S}[\![\texttt{(exp }E\texttt{)}]\!] \mapsto \lambda u.\lambda c.\lambda s.\mathcal{S}_{\mathcal{F}}[\![\texttt{exp}]\!](\mathcal{E}[\![E]\!]u)(\mathcal{E}[\![E]\!]c)(\mathcal{S}[\![E]\!]ucs)$

$\mathcal{S}_{\mathcal{F}}[\![\texttt{exp}]\!] \mapsto \lambda v.\lambda p.\lambda t.[\mathrm{e}^{v_l}\,,\,\mathrm{e}^{v_u}] \cdot t$

Example 3.14

The slope for the expression of example 3.10 would be:

$$\mathcal{S}[\![\ \texttt{(let ((square (lambda (v) (* v v))))}\ \texttt{(exp (/ 1 (+ (square x) (square y)))))}\ ]\!]$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][1\,,\,2](\mathit{bind}\,[\![\mathtt{y}]\!][2\,,\,4]u_0))$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][3/2](\mathit{bind}\,[\![\mathtt{y}]\!][3]c_0))$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][1](\mathit{bind}\,[\![\mathtt{y}]\!][0]s_0)) = [-0.076\,,\,-0.011]$$

$$\mathcal{S}[\![\ \texttt{(let ((square (lambda (v) (* v v))))}\ \texttt{(exp (/ 1 (+ (square x) (square y)))))}\ ]\!]$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][1\,,\,2](\mathit{bind}\,[\![\mathtt{y}]\!][2\,,\,4]u_0))$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][3/2](\mathit{bind}\,[\![\mathtt{y}]\!][3]c_0))$$
$$(\mathit{bind}\,[\![\mathtt{x}]\!][0](\mathit{bind}\,[\![\mathtt{y}]\!][1]s_0)) = [-0.152\,,\,-0.023]$$

which is better then the enclosure produced by the differentiator. The Krawczyk centered form accordingly improves the bound of the mean-value form given in example 3.10:

$$R(f_2, ([1\,,\,2]\,[2\,,\,4])^T, ([3/2]\,[3])^T) = [0.902\,,\,1.283]$$

3.4 Interval Matrices

3.4.1 Definition

A multidimensional interval array simply is a multidimensional array with interval components. For example, an interval vector $X \in \mathbf{I}^n$ is a set of vectors $x \in \mathbf{R}^n$ with:

$$X = \{x \mid x_i \in X_i, i = 1, \ldots, n\}$$

It represents an N-dimensional box aligned with the coordinate axes. Similarly, an interval matrix $A \in \mathbf{I}^{m \times n}$ is a set of real matrices $a \in \mathbf{R}^{m \times n}$ defined as:

$$A = \{a \mid a_{ij} \in A_{ij}, i = 1, \ldots, m\,,\ j = 1, \ldots, n\}$$

Many quantities related to intervals can simply be extended to the multidimensional case by componentwise application of the corresponding interval definition. For example, if we use the componentwise notation $A = (A_{ij})$ for $A \in \mathbf{I}^{m \times n}$, we have:

$$\begin{aligned} (A_{ij})_l &= (A_{ij\,l}) \\ (A_{ij})_u &= (A_{ij\,u}) \\ (A_{ij})_c &= (A_{ij\,c}) \end{aligned}$$

The definitions of width and distance do however need to be adapted to deal with multiple dimensions:

$$\begin{aligned} w((A_{ij})) &= \|(w(A_{ij}))\|_\infty \\ d((A_{1ij})\,,\,(A_{2ij})) &= \|(d(A_{1ij}\,,\,A_{2ij}))\|_\infty \end{aligned}$$

We use the row sum norm of a matrix defined as:

$$\|A\|_\infty = \max_{i=1,\ldots,m}\{\sum_{j=1}^{n}|A_{ij}|\}$$

3.4.2 Operators

Operations on multidimensional arrays are defined the same way as in definition (3.2). For example if $A \in \mathbf{I}^{m\times n}$ and $B \in \mathbf{I}^{n\times p}$ we define:

$$A \circ B = \square\{a \circ b \mid a \in A, b \in B\} \tag{3.8}$$

with $\circ \in \{+, -, \cdot\}$. Explicit expressions for these operators are:

$$\begin{aligned}(A_{ij}) + (B_{ij}) &= (A_{ij} + B_{ij})\\ (A_{ij}) - (B_{ij}) &= (A_{ij} - B_{ij})\\ (A_{ij}) \cdot (B_{ij}) &= (\textstyle\sum_{k=1}^{n} A_{ik}B_{kj})\end{aligned}$$

Note that for the interval representation of a matrix multiplication, as defined in equation (3.8), in general we do not have strict equality, but rather:

$$\{a \cdot b \mid a \in A, b \in B\} \subseteq A \cdot B \tag{3.9}$$

3.4.3 Properties

The commutativity of $+$ and $\cdot$ is maintained when applied to higher dimensional vectors, as is subdistributivity. But although $+$ is associative, $\cdot$ in general is not. However, in case $w(P) = 0$, i.e. P is a point matrix, we have:

$$\begin{aligned}(PA)B &\subseteq P(AB)\\ A(BP) &\subseteq (AB)P\end{aligned}$$

Finally, one can show that the higher dimensional operators as defined in equation (3.8) also are inclusion monotone.

3.4.4 Inverse

The inverse of a regular square interval matrix is defined as:

$$A^{-1} = \square\{a^{-1} \mid a \in A\}$$

An interval matrix is *regular* if it contains no singular point matrices. Verifying the regularity of interval matrices is, in general, very difficult. Some regularity tests can be found in [171]. For certain classes of matrices the problem becomes more tractable. In particular, the verification of regularity of $A_c^{-1}A$, i.e. strong regularity, can be done by checking for strict diagonal dominance. The latter holds if:

$$\|A_c^{-1}A - I\|_\infty < 1 \tag{3.10}$$

Note that $(A_c^{-1}A)_c = I$.

3.5 Linear Interval Equations

3.5.1 Definition

The solution $X_\sigma \in \mathbf{I}^n$ of $AX = B$ with $A \in \mathbf{I}^{n\times n}$ and $B \in \mathbf{I}^n$ is defined as:

$$X_\sigma = \square\{x \ : \ ax = b \mid a \in A, b \in B\} \tag{3.11}$$

When A is regular this corresponds to:

$$X_\sigma = \square\{a^{-1}b \mid a \in A, b \in B\} \subseteq A^{-1}B$$

It is a remarkable fact that no strict equality is obtained in the above expression. If in addition the solution is restricted to a given interval X, we use the following notation:

$$\Sigma(A, B, X) = X_\sigma \cap X$$

The solution of interval equations is inclusion monotone, in the sense that:

$$\left.\begin{array}{l} A' \subseteq A \\ B' \subseteq B \end{array}\right\} \Rightarrow X'_\sigma \subseteq X_\sigma$$

For a square system of N interval equations, there can be up to $\mathcal{O}(2^{2N})$ linear point systems bounding the exact solution. So, the difficulty is to find tight bounds for the solution of this equation while avoiding combinatorial explosion. This problem is relatively well understood, and a thorough treatment of possible solution methods can be found in [171].

Example 3.15

To illustrate the complexity of interval equations we borrow the following simple example from [86]:

$$\begin{array}{rcl} [2\,,\,3]X_1 + [0\,,\,1]X_2 & = & [0\,,\,120] \\ [1\,,\,2]X_1 + [2\,,\,3]X_2 & = & [60\,,\,240] \end{array} \tag{3.12}$$

The exact solution is represented by the shaded area in figure 3.3, the corresponding interval solution is:

$$X_\sigma = [-120\,,\,90] \times [-60\,,\,240]$$

Solution methods can be simplified by preconditioning the system with the midpoint inverse [90, 130, 157, 168] and use the solution X_p of $(A_c^{-1}A)X = A_c^{-1}B$ instead, with $X_\sigma \subseteq X_p$. The overestimation occurring by this simplification is quantified by:

$$w(X_\sigma) \leq w(X_p) \leq \frac{1+\beta}{1-\beta} w(X_\sigma)$$

for $\|A_c^{-1}A - I\|_\infty \leq \beta < 1$ [171]. Unfortunately this implies that considerable overestimation is possible when $\beta \approx 1$.

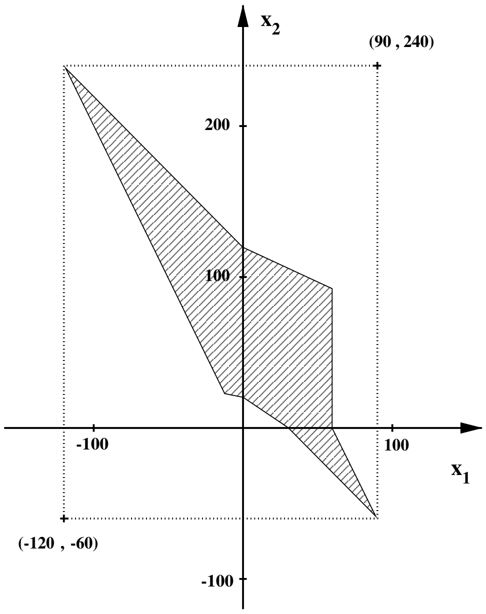

Figure 3.3: Exact solution of interval equations (3.12)

3.5.2 Krawczyk's method

Krawczyk's operator [126] is defined by:

$$\mathcal{K}(A, B, X) = (B - (A - I)X) \cap X \tag{3.13}$$

It has the property:

$$X_\sigma \subseteq X \Rightarrow X_\sigma \subseteq \mathcal{K}(A, B, X)$$

For simplicity we will assume here that $X_\sigma \subseteq X$. The enclosures produced by Krawczyk's operator on simple initial enclosures have the quadratic approximation property if A is regular, i.e. if both $w(A)$ and $w(B)$ are of $\mathcal{O}(\epsilon)$ then $d(\mathcal{K}(A, B, X)\,,\, X_\sigma)$ is of order $\mathcal{O}(\epsilon^2)$.

When A is strongly regular we can apply Krawczyk's operator on the following crude bound:

$$X_\sigma \subseteq X_0 = \langle A\rangle^{-1}|B|[-1\,,\,1] \tag{3.14}$$

to obtain Krawczyk's direct inverse solution:

$$X_{\mathcal{K}d} = B + \langle A\rangle^{-1}|A - I||B|[-1\,,\,1] \tag{3.15}$$

$\langle A\rangle$ is the comparison matrix [177], with:

$$\begin{array}{rcl} \langle A\rangle_{ii} & = & \langle A_{ii}\rangle \\ \langle A\rangle_{ij} & = & -|A_{ij}| \quad \textit{for } i \neq j \end{array}$$

So that Krawczyk's direct inverse solution requires only the inversion of one point matrix. Krawczyk's iteration is obtained by repeated application of the operator:

$$X_{\mathcal{K}}^{(l+1)} = \mathcal{K}(A, B, X_{\mathcal{K}}^{(l)})$$

and in the limit one obtains:

$$X_\kappa = \lim_{l\to\infty} X_\kappa^{(l)} \tag{3.16}$$

When:

$$\|A - I\|_\infty \leq \beta < 1$$

the quality of the above bounds can be quantified by:

$$w(X_\sigma) \leq w(X_\kappa) \leq w(X_{\kappa d}) \leq \frac{1+\beta}{1-\beta} w(X_\sigma)$$

for every initial enclosure $X_\kappa^{(0)}$ of iteration (3.16) [171]. This indicates that both solutions are very good for small β. Furthermore if $X_{\kappa d} \subseteq X_\kappa^{(0)}$ it can be shown that $X_\kappa = X_{\kappa d}$. Although Krawczyk's iteration can yield better results than the direct inverse solution, the latter is usually preferred since it is more efficient.

Example 3.16

For the linear interval equations (3.12) of example 3.15, after preconditioning with the midpoint inverse, we find:

$$X_{\kappa d} = \begin{pmatrix} [-140.455\,,\,167.728] \\ [-163.637\,,\,267.273] \end{pmatrix}$$

and indeed:

$$X_\kappa = \begin{pmatrix} [-140.455\,,\,167.728] \\ [-163.637\,,\,267.273] \end{pmatrix}$$

This solution corresponds to the following system of equations:

$$\begin{aligned} 2.5X_1 + 0.5X_2 &= [-217.5\,,\,337.5] \\ 1.5X_1 + 2.5X_2 &= [-157.5\,,\,457.5] \end{aligned}$$

and is illustrated in figure 3.4. For this example $\beta = \|A_c^{-1}A - I\|_\infty = 0.364$ so that the overestimation bound $(1+\beta)/(1-\beta) = 6.334$ is rather conservative as compared to $w(X_\kappa)/w(X_\sigma) = 1.468$.

3.5.3 Gauss-Seidel Iteration

The Gauss-Seidel method can be extended to interval arithmetic [201]. For A and $B \in \mathbf{I}$ we first define the one dimensional solution operator over domain $X \in \mathbf{I}$:

$$\gamma(A, B, X) = \Box\{x \in X \;:\; ax = b \mid a \in A, b \in B\}$$

The corresponding explicit formula is [89]:

$$\gamma(A,B,X) = \begin{cases} B/A \cap X & \textit{if } 0 \notin A \\ \Box\{x \mid x \in X \textit{ and } (x \leq B_l/A_l \textit{ or } x \geq B_l/A_u)\} & \textit{if } B > 0 \in A \\ \Box\{x \mid x \in X \textit{ and } (x \leq B_u/A_u \textit{ or } x \geq B_u/A_l)\} & \textit{if } B < 0 \in A \\ X & \textit{if } 0 \in A, B \end{cases}$$

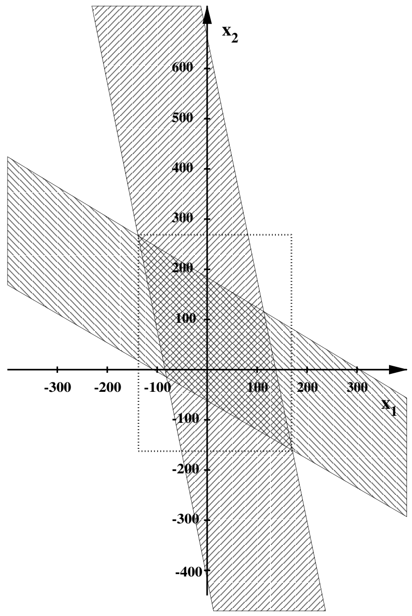

Figure 3.4: Krawczyk's solution X_κ of equations (3.12)

The multidimensional solution operator for $A \in \mathbf{I}^{n\times n}$ and $B \in \mathbf{I}^n$ can now be defined as:

$$\gamma_i(A,B,X) = \gamma(A_{ii}, B_i - \sum_{k<i} A_{ik}Y_k - \sum_{k>i} A_{ik}X_k, X_i) \tag{3.17}$$

Note that even when A is singular this operator can possibly improve the estimate. In addition, it can be proven that the Gauss-Seidel iterates *always* are better than Krawczyk's iterates [171]:

$$\gamma(A,B,X) \subseteq \kappa(A,B,X) \tag{3.18}$$

The Gauss-Seidel iteration consequently also has the quadratic approximation property. More importantly, this statement is true for a larger class of Krawczyk like iterations [171]. In particular, it shows that, as opposed to non interval equations, no improvement can be obtained by the use of so called overrelaxation techniques.

The limit of the Gauss-Seidel iteration:

$$X_\gamma^{(l+1)} = \gamma(A,B,X_\gamma^{(l)})$$

is denoted as:

$$X_\gamma = \lim_{l\to\infty} X_\gamma^{(l)}$$

When A is strongly regular X_γ is of similar quality as X_κ [171] but in view of (3.18) the Gauss-Seidel method is always preferred.

Example 3.17

The Gauss-Seidel solution X_γ of the same preconditioned system of example 3.16 is:

$$X_\gamma = \begin{pmatrix} [-130.228\,,\, 167.728] \\ [-104.416\,,\, 267.273] \end{pmatrix}$$

Clearly we have $X_\gamma \subseteq X_\kappa$.

The speed of convergence can be improved if the forward sweep (3.17) is followed by a backward sweep [2]. This is a fairly inexpensive improvement as some quantities do not need to be recomputed.

Finally, when $X_\gamma^{(l)} \subset i(X_\gamma^{(0)})$ and assuming that midpoint inverse preconditioning is used, it can be shown that A is strongly regular. Furthermore, Neumaier in [171] found that for these matrices, X_γ can be obtained in one step. As was the case for Krawczyk's direct solution (3.15), application of the Gauss-Seidel operator on initial bound (3.14) yields following explicit result:

$$X_{\gamma d_i} = \frac{b_i + (\langle A_{ii} \rangle X_{0i} - |B_i|)[-1\,,\,1]}{A_{ii}}$$

The iterative solution process can therefore be replaced by this direct solution as soon as strong regularity is detected. On the other hand, the strong regularity result also implies that if the matrix is not strongly regular and $\kappa(A, B, X) \neq \phi$, at least one of the bounds will not be improved.

3.5.4 Gauss Elimination

Systems of linear interval equations can also be solved by Gauss elimination. Indeed, the Gauss elimination solution $e(a, b)$ can be applied on definition (3.11):

$$X_\sigma = \Box\{e(a, b) \mid a \in A, b \in B\}$$

If we simply carry out the elimination step:

$$\begin{array}{rcl} L_{ij} & = & A_{ij}^{(j-1)} / A_{jj}^{(j-1)} \\ A_{ik}^{(j)} & = & A_{ik}^{(j-1)} - L_{ij} A_{jk}^{(j-1)} \\ B_i^{(j)} & = & B_i^{(j-1)} - L_{ij} B_j^{(j-1)} \end{array} \tag{3.19}$$

for i *and* $k > j$ and the backsubstitution step:

$$X_{\varepsilon j} = \frac{B_j^{(j-1)} - \sum_{k>j} A_{jk}^{(j-1)} X_{\varepsilon k}}{A_{jj}^{(j-1)}} \cap X_j \tag{3.20}$$

in interval arithmetic, by inclusion monotonicity we have:

$$X_\sigma \subseteq X_\varepsilon$$

As shown above, if the solution is restricted to a given domain X, component wise intersection can be performed in the backsubstitution step. The corresponding solution operator is denoted by:

$$X_\varepsilon = \mathcal{E}(A, B, X)$$

The Gaussian elimination algorithm can only be used when $0 \notin A_{jj}^{(j-1)}$, furthermore it can sometimes give unacceptably bad results [201]. However, when midpoint inverse preconditioning is used and A is strongly regular, it can be shown that Gauss elimination always can be carried out [3]. In addition, we have the following important result [171]:

$$\begin{aligned} X_\varepsilon &\subseteq X_\gamma \\ |X_\varepsilon| &= |X_\gamma| \end{aligned}$$

This means that under the above conditions, one endpoint of each of the components of X_ε agrees with X_γ, whereas the other endpoint may be sharper.

Example 3.18

If we apply the Gauss elimination algorithm to equations of example 3.16, we obtain:

$$X_\varepsilon = \begin{pmatrix} [-130.228\,,\,167.728] \\ [-60.000\,,\,267.273] \end{pmatrix}$$

which is an improvement over X_γ.

It is therefore always recommended to use midpoint inverse preconditioning in conjunction with Gauss elimination, in which case Gauss elimination is the preferred solution. Unfortunately it can only be guaranteed that the algorithm can be carried out when the matrix is strongly regular.

3.5.5 Other Methods

In [6] Alefeld focuses on the 'Single Step Method with Intersection after Every Component' (SIC). Gauss-Seidel iterates are very similar to SIC iterates, but in addition $X_i^{(l+1)}$ is solved for in every step. They will therefore always produce sharper results. Note that the symmetric version of SIC finds its counterpart in the application of Gauss-Seidel in forward and in backward sweeps.

An interval version of the Gauss-Jordan method was proposed by Hansen in [85, 90]. This method is however less efficient then Gauss elimination and the generalization of (3.18) by [171] proves that it never produces bounds that are sharper than Gauss-Seidel.

A different approach was adopted by Hansen in [86]. The proposed method uses the signs of $\partial x_r/\partial a_{ij}$ and $\partial x_r/\partial b_i$ to determine which interval bounds of A_{ij} and B_i correspond with the bounds of X_σ. This method is however computationally expensive and relies on other solution methods when the signs of the partial derivatives can not be determined.

More recently, methods have been proposed to find the exact solution X_σ. In [203, 204], Rohn shows that if $\beta = \|A_c^{-1}A - I\|_\infty \ll 1$ and A is inverse stable, i.e. $0 \notin (A^{-1})_{jj}$, this can be done in $\mathcal{O}(N^3)$ operations. When $\beta < 1$ becomes larger, the complexity increases to $\mathcal{O}(N^4)$. Finally, when A is not strongly regular and $\beta \geq 1$ the worst case complexity is often exponential in N.

3.6 Solution of Nonlinear Equations

Most of the solution techniques for solving nonlinear algebraic equations are obtained by extending classical numerical algorithms. A good overview of these techniques can be found in [266].

3.6.1 Newton Interval Method

One of the most effective techniques is to extend Newton's method with intervals [5, 7]. Figure 3.5 illustrates the 1–dimensional case. The classical Newton method is locally ap-

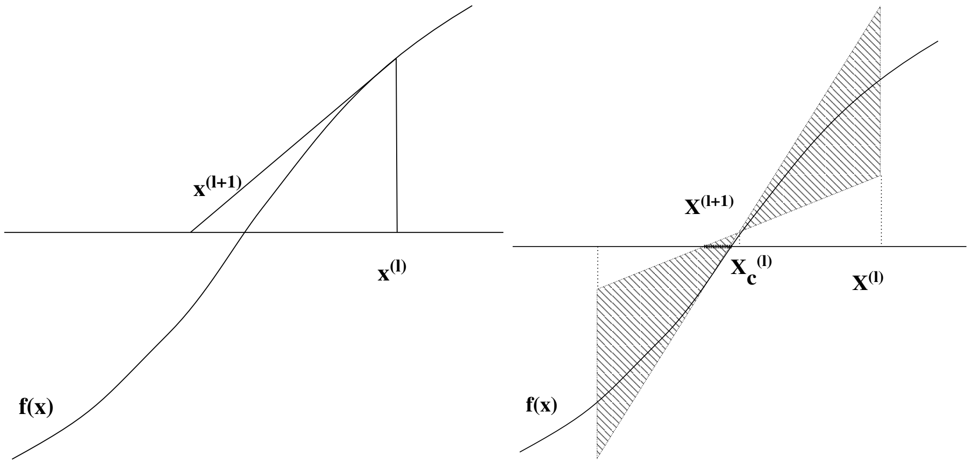

Figure 3.5: Interval extension of Newton's method

proximates the function near the current estimate $x^{(l)}$:

$$f(x) \approx f(x^{(l)}) + f'(x^{(l)})(x - x^{(l)}) \tag{3.21}$$

here superscripts indicate iteration numbers. The solution of $f(x) = 0$ is then approximated with the solution of this linear equation, which is used as next iterate:

$$x^{(l+1)} = x^{(l)} - \frac{f(x^{(l)})}{f'(x^{(l)})} \tag{3.22}$$

The interval extension of this method uses the mean value form instead of approximation (3.21):

$$f(X) \subseteq f(X_c^{(l)}) + f'(X^{(l)})\,(X - X_c^{(l)}) \tag{3.23}$$

to obtain an enclosure of the function range, and uses the corresponding interval solution as next iterate to bound the solution domain:

$$X^{(l+1)} = X_c^{(l)} - \frac{f(X_c^{(l)})}{f'(X^{(l)})} \tag{3.24}$$

Starting from the function value at the midpoint of the interval, the bounds on the derivatives limit the function to lie within the shaded area. This means that any solution has to belong to the intersection of this area with the zero-axis, which becomes the solution interval for the next iteration. As mentioned earlier, in the case of finite precision computations, the last significant digit is always rounded conservatively, i.e. downwards for the lower bound and upwards for the upper bound.

For a system of nonlinear equations $X \in \mathbf{I}^n$ and the interval derivative in equation (3.23) is replaced by an interval Jacobian $J(X) \in \mathbf{I}^{n\times n}$:

$$f(X_c^{(l)}) + J(X^{(l)})\,(X - X_c^{(l)}) = 0 \tag{3.25}$$

As was originally discussed in [161], a solution to this interval equation can be used as next iterate. The Newton operator uses the exact solution of (3.25) and is defined as:

$$N(f, X) = X_c - \Sigma(J(X), f(X_c), \Delta X) \tag{3.26}$$

the corresponding Newton iterates are therefore obtained by:

$$X^{(l+1)} = N(f, X^{(l)})$$

The quadratic convergence of these iterates:

$$\exists Q \in \mathbf{R} \;:\; w(X^{(l+1)}) \leq Q w(X^{(l)})^2$$

is well known, and already dates back to Bartle [15].

We illustrate the approximation introduced by the Newton operator with a simple example.

Example 3.19

The system of equations:

$$\begin{array}{rcl} x_1^2 + x_2^2 - 1 & = & 0 \\ (x_1 - 1)^2 + x_2^2 - 1 & = & 0 \end{array} \tag{3.27}$$

has two solutions $(1/2 \;\; \sqrt{3}/2)^T$ and $(1/2 \;\; -\sqrt{3}/2)^T$. With initial-domain:

$$X^{(0)} = \begin{pmatrix} [0.2\,,\,0.8] \\ [0.6\,,\,0.9] \end{pmatrix}$$

equation (3.25) gives the following linear interval approximation:

$$\begin{array}{rcl} [0.4\,,\,1.6]\Delta X_1 + [1.2\,,\,1.8]\Delta X_2 & = & [0.1875] \\ [-1.6\,,\,-0.4]\Delta X_1 + [1.2\,,\,1.8]\Delta X_2 & = & [0.1875] \end{array} \tag{3.28}$$

This is illustrated in figure 3.6. The shaded areas represent the feasible domains for each of the functions, and their intersection corresponds with the exact solution of (3.28). The first Newton iterate is:

$$X^{(1)} = \begin{pmatrix} [0.406\,,\,0.594] \\ [0.854\,,\,0.900] \end{pmatrix}$$

Note that for this first iteration the exact solution of (3.28) is not contained in $X^{(0)}$.

To achieve a 10 digit accuracy, the remaining Newton iterates are:

iteration	X	Y
0	[+0.2v0 +0.8v0]	[+0.6v0 +0.9v0]
1	[+0.4v0 +0.6v0]	[+0.85v0 +0.90v0]
2	[+0.499v0 +0.501v0]	[+0.8658v0 +0.8664v0]
3	[+0.4999999v0 +0.5000001v0]	[+0.86602538v0 +0.86602544v0]
4	[+0.4999999999v0 +0.5000000001v0]	[+0.8660254037v0 +0.8660254038v0]

Linear interval equation (3.25) can be solved using the solution methods discussed in section 3.5. In particular, if for each iterate the solution of (3.25) is approximated by a single application of a linear operator, we obtain the following solution operators.

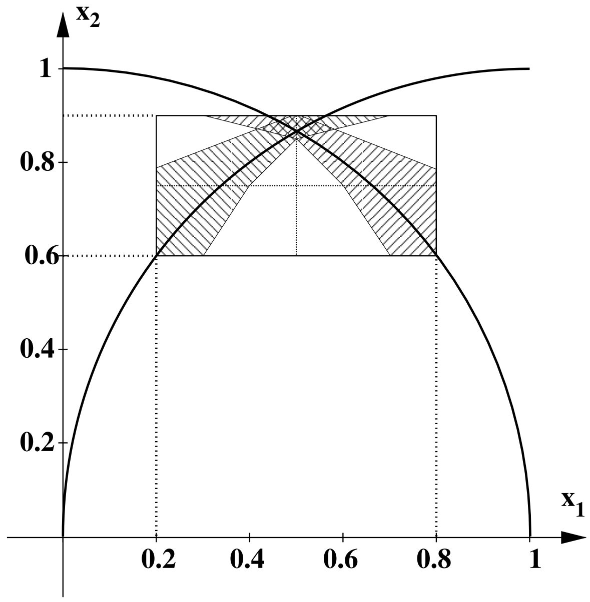

Figure 3.6: Interval Newton approximation for example 3.19

3.6.2 Krawczyk's Operator

Application of Krawczyk's linear operator (3.13) to solve (3.25) leads to the corresponding nonlinear krawczyk operator:

$$K(f, X) = X_c - \kappa(J(X), f(X_c), \Delta X) \tag{3.29}$$

Krawczyk's operator was originally proposed in [126].

Figure 3.7 shows the geometric interpretation of the corresponding 1–dimensional Newton interval method. Instead of finding the exact solution, Krawczyk's operator intersects

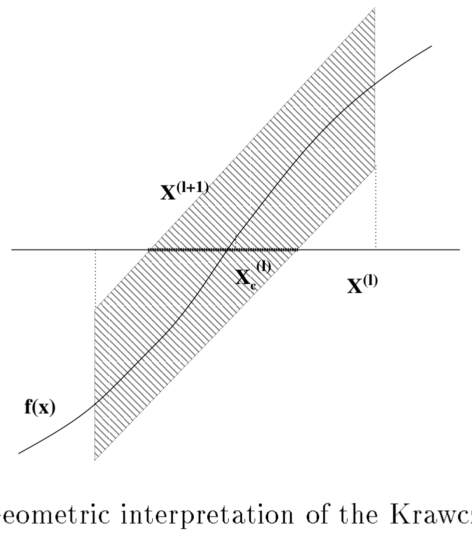

Figure 3.7: Geometric interpretation of the Krawczyk's operator

more conservative, but less complicated boundaries. Indeed, for a system of equations, the complexity of the preconditioned Krawczyk operator is only $\mathcal{O}(N^3)$, as the matrix to be inverted is not an interval matrix.

Repeated application of Krawczyk's operator leads to the iterates:

$$X^{(l+1)} = K(f, X^{(l)})$$

Example 3.20

If we apply Krawczyk's operator on the same initial box of example 3.19, we obtain the following equations:

$$\begin{array}{rcl} \Delta X_1 + 1.5\Delta X_2 & = & [-0.038\,,\,0.413] \\ -\Delta X_1 + 1.5\Delta X_2 & = & [-0.038\,,\,0.413] \end{array} \tag{3.30}$$

with solution:

$$X^{(1)} = \begin{pmatrix} [0.275\,,\,0.725] \\ [0.725\,,\,0.900] \end{pmatrix}$$

This is illustrated in Figure 3.8.

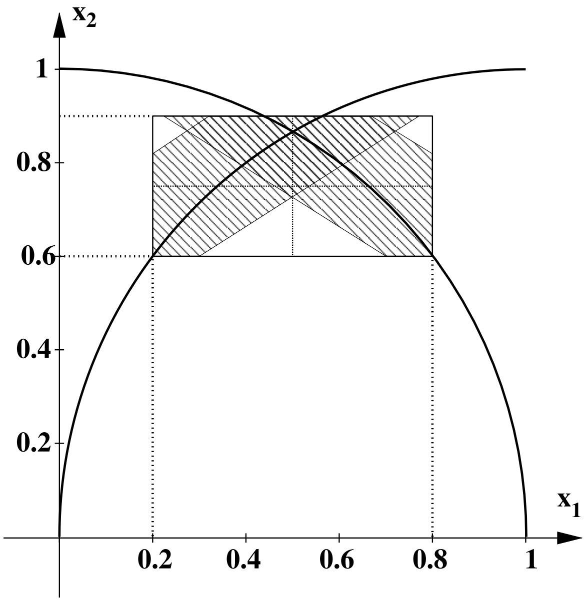

Figure 3.8: Krawczyk's approximation for example 3.19

The following iterates are:

iteration	X	Y
0	[+0.2v0 +0.8v0]	[+0.6v0 +0.9v0]
1	[+0.2v0 +0.8v0]	[+0.7v0 +0.9v0]
2	[+0.3v0 +0.7v0]	[+0.7v0 +0.9v0]
3	[+0.4v0 +0.6v0]	[+0.8v0 +0.9v0]
4	[+0.47v0 +0.53v0]	[+0.85v0 +0.89v0]
5	[+0.497v0 +0.503v0]	[+0.864v0 +0.868v0]
6	[+0.49997v0 +0.50003v0]	[+0.86601v0 +0.86605v0]
7	[+0.499999997v0 +0.500000003v0]	[+0.866025402v0 +0.866025406v0]
8	[+0.4999999999v0 +0.5000000001v0]	[+0.8660254037v0 +0.8660254038v0]

Moore [163] proposed a small modification to Krawczyk's iterative solution and proved superlinear convergence with fixed interval Jacobian. For variable Jacobians we have quadratic convergence, as shown in [166]. Alefeld [7] discovered that Gauss elimination could replace the inversion of the preconditioning matrix by modifying (3.29):

$$A(f, X) = X_c - e(J(X)_c, f(X_c) + \Delta J(X)\Delta X, \Delta X) \tag{3.31}$$

Although this *Alefeld Operator* is more efficient, Gauss elimination is performed with an interval right hand side. Therefore, by subdistributivity, this modification can produce wider intervals than (3.29):

$$K(f, X) \subseteq A(f, X)$$

Note also that in (3.31) intersection with the original domain ΔX can be performed during the backsubstitution step.

3.6.3 Hansen-Sengupta Operator

The Hansen-Sengupta operator is the non-linear version of the Gauss-Seidel operator for linear systems [89]:

$$H(f, X) = X_c - \gamma(J(X), f(X_c), \Delta X) \tag{3.32}$$

Since linear Gauss-Seidel iterates are better than Krawczyk iterates (3.18), we have:

$$H(f, X) \subseteq K(f, X)$$

The Hansen-Sengupta operator is therefore always preferred to Krawczyk's operator, as the following example shows.

Example 3.21

In the same conditions as example 3.20 the first Hansen-Sengupta iterate:

$$X^{(1)} = \begin{pmatrix} [0.387\,,\, 0.613] \\ [0.816\,,\, 0.900] \end{pmatrix}$$

is better as the first Krawczyk iterate and the convergence is faster:

iteration	X	Y
0	[+0.2v0 +0.8v0]	[+0.6v0 +0.9v0]
1	[+0.3v0 +0.7v0]	[+0.81v0 +0.90v0]
2	[+0.49v0 +0.51v0]	[+0.864v0 +0.869v0]
3	[+0.49998v0 +0.50002v0]	[+0.866024v0 +0.866028v0]
4	[+0.4999999999v0 +0.5000000001v0]	[+0.8660254037v0 +0.8660254038v0]

A symmetric version of the Hansen-Sengupta operator is proposed in [222].

3.6.4 Gauss-Newton Operator

In [217], Schwandt first introduced an operator that uses Gauss elimination to solve (3.25), the *Gauss-Newton operator*:

$$G(f, X) = X_c - \varepsilon(J(X), f(X_c), \Delta X) \tag{3.33}$$

Alefeld proves its quadratic convergence and discusses the conditions under which the Gauss-Newton iterates converge [5].

Example 3.22

If we apply the Gauss-Newton operator on equations (3.27), we obtain:

$$X^{(1)} = \begin{pmatrix} [0.387\,,\,0.613] \\ [0.833\,,\,0.900] \end{pmatrix}$$

and

iteration	X	Y
0	[+0.2v0 +0.8v0]	[+0.6v0 +0.9v0]
1	[+0.3v0 +0.7v0]	[+0.83v0 +0.90v0]
2	[+0.4998v0 +0.5002v0]	[+0.8659v0 +0.8662v0]
3	[+0.49999999v0 +0.50000001v0]	[+0.86602539v0 +0.86602541v0]
4	[+0.4999999999v0 +0.5000000001v0]	[+0.8660254037v0 +0.8660254038v0]

which is an improvement over the Hansen-Sengupta operator.

3.6.5 Other Newton Operators

Various hybrid approaches are possible, see for example [88] for a good combination of Gauss-Seidel iterations with Gauss elimination.

Shearer and Wolfe improve the convergence of Krawczyk's method by performing a number of inner iterations with the same interval Jacobian at each step [265]. In [224] they describe a method where the number of inner iterations are dermined automatically and apply it to the Alefeld operator. Inner iterations for the Hansen-Sengupta operator are discussed [221].

Neumaier in [169] shows how quadratic convergence can be maintained with a fixed interval Jacobian, when point Newton iterates are performed on the center.

Another class of Newton operators relies on inclusion of the inverse Jacobian. These operators are discussed in [6]. A very elegant operator is found in [6][Chapter 20]. Here iterates for inclusion of the inverse Jacobian are performed at the same time as the corresponding Newton iterates, while maintaining quadratic convergence.

3.6.6 Interval Slopes Operator

Krawczyk's interval slopes can be used to bound the range of functions. Similarly to Newton's method this set of equations can be inverted to obtain a bound on the solution [87, 134]:

$$f(X_c^{(l)}) + S(f, X^{(l)}, X_c^{(l)})\,(X - X_c^{(l)}) = 0 \tag{3.34}$$

As interval slopes produce sharper enclosures then derivatives (3.7), better solution bounds are obtained. The iterates are however not necessarily enclosed by the corresponding Newton iterates, as the following example shows:

Example 3.23

Using Krawczyk's slope evaluator described in section 3.3.7 and the Gauss-Newton operator we obtained the following results for (3.27):

$$X^{(1)} = \begin{pmatrix} [0.468\,,\,0.532] \\ [0.859\,,\,0.896] \end{pmatrix}$$

iteration	X	Y
0	[+0.2v0 +0.8v0]	[+0.6v0 +0.9v0]
1	[+0.46v0 +0.54v0]	[+0.85v0 +0.90v0]
2	[+0.4997v0 +0.5003v0]	[+0.8659v0 +0.8663v0]
3	[+0.49999998v0 +0.50000002v0]	[+0.86602539v0 +0.86602542v0]
4	[+0.4999999999v0 +0.5000000001v0]	[+0.8660254037v0 +0.8660254038v0]

These iterates are clearly better then the corresponding Newton iterates.

A drawback of this approach is that the slopes inclusions depend on the center of the interval equation (3.34). It is therefore not possible to improve the convergence by corresponding point iterations for the center. Furthermore the uniqueness result associated with Newton's method does not apply for interval slopes, as is discussed below.

3.7 Existence and Uniqueness Tests

If f is Lipschitz continuous, the interval operators $I(f, X)$ discussed above are conservative:

$$x \in X, f(x) = 0 \;\Rightarrow\; x \in I(f, X) \tag{3.35}$$

and as a direct consequence, we have:

$$I(f, X) \cap X = \phi \;\Rightarrow\; \forall x \in X \;:\; f(x) \neq 0 \tag{3.36}$$

which allows us to check for the existence of solution in a given domain. Furthermore it can be shown that if there are no solutions in X either (3.36) will hold after a finite number of iterations or $I(f, X) = X \;\Rightarrow\; f(X) = 0$. This property is called *Strong Convergence*. Uniqueness of a solution can be determined by:

$$I(f, X) \subseteq i(X) \;\Rightarrow\; \exists!\, x \in X \;:\; f(x) = 0 \tag{3.37}$$

For Newton's operator the existence test is due to Moore [162], whereas the uniqueness test was discovered by Nickel [172]. The strong convergence result is due to Neumaier [169]. Similar results hold for Krawczyk's operator [116, 126, 128, 163]. In case of the Hansen-Sengupta operator existence was proven by Hansen [89], uniqueness by Moore [167] and Qi [189], and strong convergence by Alefeld [4]. Existence and uniqueness tests for the symmetric Hansen-Sengupta operator are given by Shearer and Wolfe [223]. As these operators produce increasingly better enclosures these tests are also increasinlgy sharper. Similar results hold for the slope solution operator [170]. Unfortunately the uniqueness result is not valid, instead we have:

$$I(f, X) \subseteq i(X) \;\Rightarrow\; \exists x \in X \;:\; f(x) = 0 \tag{3.38}$$

3.8 Initial Domains

The convergence properties of the interval Newton method described above are only valid asymptotically. Indeed, in general there is no guarantee that the sequence of intervals generated by the Krawczyk's operator (3.29) will converge at all.

This implies that any practical implementation requires methods to find appropriate initial domains. This problem is very similar to the one of finding good starting points for iterative solution techniques. In order to ensure 100% reliability, it is however required that that *all* possible solution domains are found. The classic approach is to recursively bisect the original domain in the direction corresponding with the largest interval [118, 166], while attempting to eliminate these newly created subdomains. Testing if a domain could contain a solution usually is done using an interval function evaluation, i.e. $0 \in F(X^{(l)})$, or by using one of the above existence tests. Unfortunately, the subdivision technique can very quickly generate an exponential number of candidate domains when the starting domain is large.

This problem has received only very little attention in the literature. It is only recently that researchers actively started investigating optimal subdivision strategies. Their approach is to bisect in the direction that maximizes the chance of eliminating new candidate domains [195]. Although the fundamental complexity of the global solution of nonlinear algebraic equations can not be reduced, results to date indicate that significant improvements can be made over existing subdivision techniques.

Example 3.24

To illustrate the amount of work involved we apply this approach to equation (3.27) with:

$$X^{(0)} = \begin{pmatrix} [-2\,,\,3] \\ [-2\,,\,2] \end{pmatrix}$$

and exact values of $f(X)$ and $J(X)$. The bisections are illustrated in figure 3.9 We used the Gauss-Seidel operator and resorted to bisection when width reduction was smaller than 90%. The shaded areas indicate the converging Gauss-Seidel iterations.

The initial domains for typical design problems however, are usually not very large. We therefore have decided on the one hand to improve accuracy and speed of interval enclosure methods, and on the other to address the consistency problem.

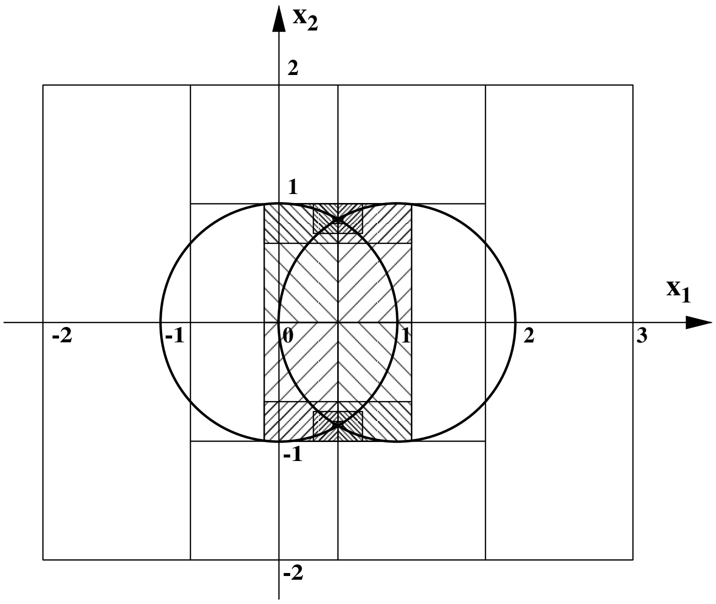

Figure 3.9: Bisections for example 3.24

Chapter 4

On Solving Midpoint Preconditioned Linear Interval Equations

4.1 Introduction

In this chapter we present a number of new results on the solution of midpoint preconditioned linear interval equations. A geometric interpretation of the midpoint preconditioning operation and the different solution algorithms is presented in section 4.2. In section 4.3 we propose a modification of the Gauss elimination solution technique. This method never fails, uses current domain information and *always* produces better enclosures then the Gauss-Seidel method. In section 4.4, we describe an $\mathcal{O}(N^3)$ algorithm to find exact solutions in case of regular matrices. This algorithm is modified in section 4.5 to find good enclosures when an initial domain is considered, including the case of singular matrices.

4.2 Geometric Interpretation of the Solution of Linear Interval Equations

4.2.1 Introduction

To give a geometric interpretation of both the Gauss-Seidel and Gauss elimination methods, we will again use example 3.15 borrowed from [86]. After midpoint preconditioning equations (3.12) become:

$$\begin{aligned}
[0.7272\,,\,1.2728]X_1 + [-0.2728\,,\,0.2728]X_2 &= [-21.819\,,\,49.091] \\
[-0.3637\,,\,0.3637]X_1 + [0.6363\,,\,1.3637]X_2 &= [-5.4546\,,\,109.091]
\end{aligned} \tag{4.1}$$

The effect of this operation is shown in figure 4.1. As is the case here, when $B_c \neq 0$ some overestimation can occur. However, when $B_c = 0$ no accuracy will be lost by this operation. So, as far as solving linear interval equations is concerned, it therefore pays to find a good center.

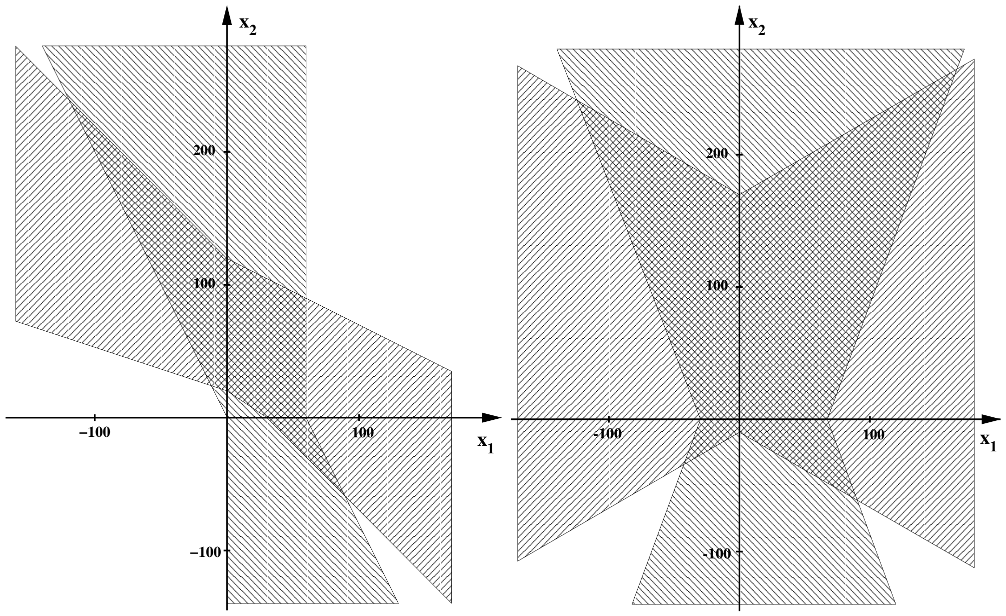

Figure 4.1: Midpoint Preconditioning of Linear Interval Equations

4.2.2 Gauss-Seidel Method

For equation (3.12), with initial domain:

$$X^{(0)} = \begin{pmatrix} [-200\,,\,200] \\ [-200\,,\,300] \end{pmatrix}$$

the Gauss-Seidel method is illustrated in figure 4.2. Using $X_2 = [-200\,,\,300]$ the first equation is solved for X_1, resulting in $X_1 = [-142.5\,,\,180.0]$. With these bounds one then solves the second equation for X_2. The Gauss-Seidel solution therefore is:

$$X_\gamma^{(1)} = \begin{pmatrix} [-142.5\,,\,180.0] \\ [-111.429\,,\,274.286] \end{pmatrix}$$

Since we have applied the Gauss-Seidel operator only once, these results are not as good as the Gauss-Seidel fixed point solution X_γ.

Note that the geometric interpretation presented in this section is also valid for linear interval systems, which are not midpoint preconditioned.

4.2.3 Gauss Elimination Method

The geometric interpretation of the Gauss elimination method is depicted in figure 4.3. Elimination of the first variable corresponds to a projection of the solution set onto the corresponding subspace. This —for now unknown— projection is then used in the second equation to solve for the remaining variable X_2. This variable is then backsubstituted in the first equation to find X_1. The overestimation occurred by elimination is associated with the projection operation.

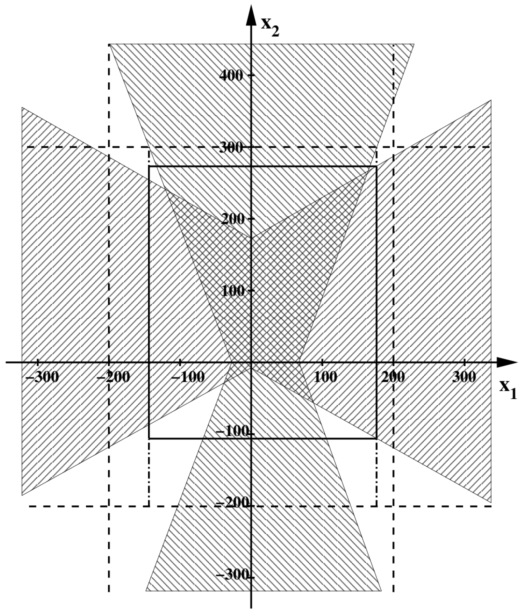

Figure 4.2: Geometric Interpretation of Gauss Seidel Solution of equations (3.12)

For equation (3.12) the Gauss elimination method produces:

$$X_\varepsilon = \begin{pmatrix} [-130.228\,,\,167.728] \\ [-60.000\,,\,267.273] \end{pmatrix}$$

It is now clear why Gauss elimination produces better results then Gauss-Seidel on midpoint preconditioned systems, i.e. $X_\varepsilon \subseteq X_\gamma$.

The geometric interpretation presented in this section does not apply for linear interval systems which are not midpoint preconditioned. The special structure of midpoint preconditioned systems indeed ensures that the correspondence between the worst case interval bounds is not lost, which is not the case for general linear interval systems.

4.3 Modified Gauss Elimination Solution

Gauss elimination is superior to Gauss-Seidel in case of midpoint preconditioned matrices [171][Thm 4.5.11]. However, it can not be carried out on a singular matrix, in which case a zero diagonal interval element is encountered. Furthermore, it does not make good use of the initial domain. For these reasons the midpoint preconditioned Gauss-Seidel operator is usually preferred.

To overcome these shortcomings we propose the following modification to the Gauss elimination scheme. Before a variable is eliminated, the Gauss-Seidel interval of that variable is computed. If this interval is within the corresponding domain interval, the current equation is used to eliminate the variable in the remaining equations. Otherwise, the Gauss-Seidel interval is used to eliminate (i.e. substitute for) that variable. This modified elimination scheme is recursively applied to the remaining equations.

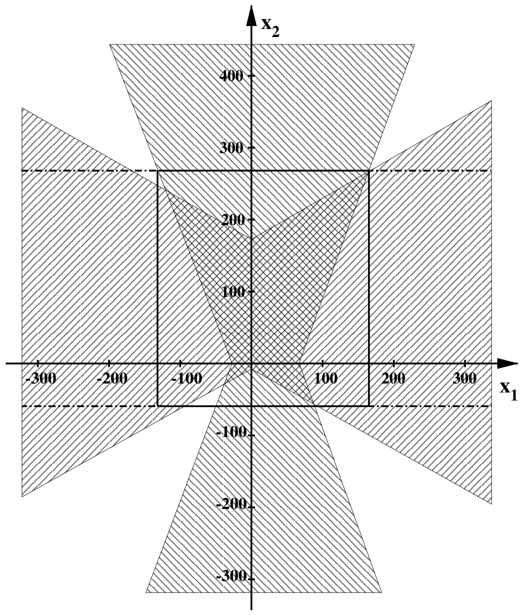

Figure 4.3: Geometric Interpretation of Gauss Elimination Solution of equations (3.12)

This modification allows for zero interval pivots, since in that case one will automatically resort to the Gauss-Seidel solution. In fact, when the initial domain is small enough, this algorithm will use Gauss-Seidel for each variable and therefore be equivalent the Gauss-Seidel operator. On the other hand when the initial domain is large, no Gauss-Seidel interval will ever be used and the algorithm simply performs Gauss elimination.

We now show that application of the Gauss-Seidel operator on the resulting system *always* produces better results than on the original system. To do this we first show that this holds after elimination of only one variable.

Let us first introduce some notation. Consider the following linear system of interval equations:

$$\sum_{j=1}^{n} A_{ij} X_j = B_i \quad i = 1, \ldots, n \tag{4.2}$$

The Gauss-Seidel solution of this system is:

$$Y_i = \gamma(A_{ii}, B_i - \sum_{j<i} A_{ij} Y_j - \sum_{j>i} A_{ij} X_j, X_i) \tag{4.3}$$

After elimination of X_1 the Gauss-Seidel solution is the same as in (4.3) for $i = 1$ and for $i, j > 1$ we have:

$$Y_i^{(1)} = \gamma(A_{ii}^{(1)}, B_i^{(1)} - \sum_{j<i} A_{ij}^{(1)} Y_j^{(1)} - \sum_{j>i} A_{ij}^{(1)} X_j, X_i) \tag{4.4}$$

where:

$$A_{ij}^{(1)} = A_{ij} - \frac{A_{i1} A_{1j}}{A_{11}} \tag{4.5}$$

$$B_i^{(1)} = B_i - \frac{A_{i1}B_1}{A_{11}} \tag{4.6}$$

Without loss of generality, we can assume that $Y_1 \subseteq X_1$. Otherwise the proposed algorithm would simply perform one Gauss-Seidel step, and the solution of both systems would be identical.

First, we need the following two lemmas. They are not new and similar results can be found in the literature [171].

Lemma 1 *Let* $A', A'', B, X \in \mathbf{I}$ *then* $\gamma(A' + A'', B, X) \subseteq \gamma(A', B - A''X, X)$

Proof: If $x \in X, (a' + a'')x = b$ then $a'x = b - a''x \in (B - A''X)$ □

Lemma 2 *Let* $A', A'', X \in \mathbf{I}$ *and* $A'_c = A''_c = 0$ *then* $(A' + A'')X = A'X + A''X$

Proof: $(A' + A'')X = |(A' + A'')||X|[-1\,, 1] = |A'||X|[-1\,, 1] + |A''||X|[-1\,, 1] = A'X + A''X$ □

We now have the following theorem:

Theorem 1 *For a midpoint preconditioned system, using definitions (4.3) and (4.4) if* $Y_1 \subseteq X_1$ *we have* $Y_i^{(1)} \subseteq Y_i$.

Proof: For $i = 1$, $Y_1^{(1)}$ and Y_1 are the same and so:

$$Y_1^{(1)} \subseteq Y_1 \tag{4.7}$$

For $i, j > 1$ by applying lemma 1 to definition (4.3) and using (4.5) we have:

$$Y_i^{(1)} \subseteq \gamma(A_{ii}, B_i^{(1)} - \sum_{j<i} A_{ij}^{(1)} Y_j^{(1)} - \sum_{j>i} A_{ij}^{(1)} X_j + \frac{A_{i1}A_{1i}}{A_{11}} Y_i^{(1)}, X_i) \tag{4.8}$$

By subdistributivity and using (4.5) and (4.6):

$$Y_i^{(1)} \subseteq \gamma(A_{ii}, B_i - \sum_{j<i} A_{ij} Y_j^{(1)} - \sum_{j>i} A_{ij} X_j - (\frac{A_{i1}B_1}{A_{11}} - \sum_{j\leq i} \frac{A_{i1}A_{1j}}{A_{11}} Y_j^{(1)} - \sum_{j>i} \frac{A_{i1}A_{1j}}{A_{11}} X_j), X_i) \tag{4.9}$$

By construction we know that $Y_j^{(1)} \subseteq X_j$, so using lemma 2, we can write:

$$Y_i^{(1)} \subseteq \gamma(A_{ii}, B_i - \sum_{j<i} A_{ij} Y_j^{(1)} - \sum_{j>i} A_{ij} X_j - A_{i1} \frac{B_1 - \sum A_{1j}X_j}{A_{11}}, X_i) \tag{4.10}$$

But by assumption:

$$Y_1 = \frac{B_1 - \sum A_{1j}X_j}{A_{11}}$$

furthermore for $j < i$ we can induce that:

$$Y_j^{(1)} \subseteq Y_j$$

so that:

$$Y_i^{(1)} \subseteq \gamma(A_{ii}, B_i - \sum_{j<i} A_{ij} Y_j - \sum_{j>i} A_{ij} X_j - A_{i1}Y_1, X_i) = Y_i \tag{4.11}$$

□

Recursive application of the theorem on $A^{(1)}$ and $B^{(1)}$ shows that application of the Gauss-Seidel operator on the eliminated system is better as compared to the original one:

$$Y_i^{(n)} \subseteq Y_i$$

By inclusion monotonicity this inequality still holds when both a forward and a backward sweep of the Gauss-Seidel operator is performed. As a consequence the proposed modified Gauss operator, is better then the symmetric Gauss-Seidel operator.

Due to the proposed modification, it is not needed to perform the intersection with the original domain at backsubstitution time. The additional cost incurred by this modification is at most $\frac{N^2}{2}$, which is less than N^2 of a simple Gauss-Seidel sweep.

The algorithm is not always better then Gauss elimination as the selection of a Gauss-Seidel solution is not guaranteed, but rather likely, to be better. So there are cases where the sequential application of the Gauss-Seidel operator and the Gauss elimination operator produces a better result. The converse is also true as the elimination projections can be better when information of the initial domain is used. This gain can help reduce the exponential complexity associated with the bisection process for the solution of nonlinear equations.

4.4 An $\mathcal{O}(N^3)$ Algorithm for the Exact Solution

As illustrated in figure 4.4 the exact solution for a midpoint preconditioned system of linear interval equations (4.1) can be seen as the intersection of the point sets corresponding to n interval equations. In this section, we ignore any initial domain information and assume that the interval matrix is regular. Both assumptions will be relaxed in the next section. Without loss of generality we also assume that $B_{i_c} \geq 0$. Indeed if it is not the case we simply redefine $X_i = -X_i$ and multiply the i-th equation by -1.

Furthermore, as the matrix A is midpoint preconditioned, the solution set of the i-th equation is symmetric w.r.t. the origin in the j-th coordinate direction $j \neq i$. The bounding surfaces of the i-th equation corresponding to B_l and B_u bound the lower, respectively upper values of x_i. Since by assumption $B_{i_c} \geq 0$ the values of x_i on the upper surface are positive and in absolute values they are larger then the corresponding x_i on the lower bounding surface. Furthermore, the x_i increase with increasing $|x_j|$. X_u therefore corresponds with the vertex v and can be found as the intersection of the n upper surfaces:

$$\langle A \rangle v = B_u$$

The lower values of x_i are bounded by the lower bounding surface. In addition they decrease with increasing $|x_j|$. X_{il} can therefore be obtained at $c[i]$, the intersection of the lower bounding surface corresponding with B_{il} and the upper bounding surfaces corresponding to B_{j_u} for $j \neq i$.

Going back to figure 4.4 we now know that the vertex v produces X_u, and so:

$$v = \langle A \rangle^{-1} B_u \tag{4.12}$$

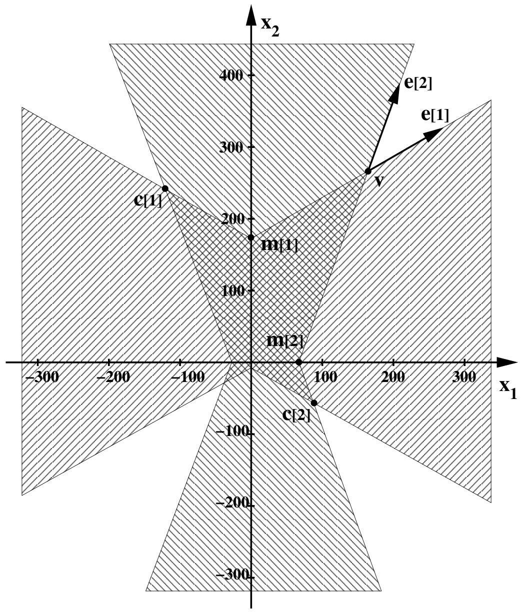

Figure 4.4: Exact Solution Algorithm for Equations (4.1)

To compute the location of each of the remaining corners $c[i]$ of the hypercube-like solution, we will follow the edges of this hypercube until we reach the opposite surface. The initial directions correspond to the vectors $e[i]$ emanating from v, which are the columns of $\langle A\rangle^{-1}$. We then reach the mirror points $m[i]$ where $m[i]_i = 0$:

$$m[i] = v - e[i]\frac{v_i}{e[i]_i}$$

Let us assume for now that the lower bounding surface is entirely below the origin, so that we can follow the image of the ray past the mirror point until we reach $c[i]$. Using the mirrored direction:

$$e'[i]_j = \begin{cases} -e[i]_j & \textit{for } i = j \\ e[i]_j & \textit{for } i \neq j \end{cases}$$

$c[i]$ can be computed as the intersection of the edge with the surface:

$$\begin{array}{rcl} \langle A\rangle^*_i c[i] - B_{il} & = & 0 \\ c[i] - (m[i] + e'[i]s') & = & 0 \end{array} \tag{4.13}$$

where $\langle A\rangle^*$ is the adjoint comparison matrix, with:

$$\begin{array}{rcl} \langle A\rangle^*_{ii} & = & \langle A_{ii}\rangle \\ \langle A\rangle^*_{ij} & = & |A_{ij}| \quad \textit{for } i \neq j \end{array}$$

Since $c[i]$ produces the lower bound in the of i-th coordinate, we are interested only in $c[i]_i$:

$$X_{il} = c[i]_i$$

Using equations (4.13) we obtain:

$$X_{il} = \alpha_i \tag{4.14}$$

where:

$$\alpha_i = 2B_{ic}e[i]_i - v_i \tag{4.15}$$

The derivation of this result is presented in appendix A.1.

For equations (4.1) we have:

$$\langle A\rangle^{-1} = \begin{pmatrix} 1.75 & 0.75 \\ 1 & 2 \end{pmatrix} \tag{4.16}$$

so that:

$$X_u = \langle A\rangle^{-1}B_u = \begin{pmatrix} 167.728 \\ 267.273 \end{pmatrix}$$

and:

$$X_l = \begin{pmatrix} (-21.818 + 49.091)1.75 - 167.728 \\ (-5.455 + 109.091)2 - 267.273 \end{pmatrix} = \begin{pmatrix} -120 \\ -60 \end{pmatrix}$$

As the following example shows, the lower bounding surfaces are not necessarily below the origin. A change of B in equation (3.12):

$$\begin{array}{rcl} [2\,,\,3]X_1 + [0\,,\,1]X_2 & = & [0\,,\,120] \\ [1\,,\,2]X_1 + [2\,,\,3]X_2 & = & [200\,,\,240] \end{array} \tag{4.17}$$

produces the following midpoint preconditioned system:

$$\begin{array}{rcl} [0.7272\,,\,1.2728]X_1 + [-0.2728\,,\,0.2728]X_2 & = & [-21.819\,,\,36.364] \\ [-0.3637\,,\,0.3637]X_1 + [0.6363\,,\,1.3637]X_2 & = & [58.182\,,\,109.091] \end{array} \tag{4.18}$$

which is shown in figure 4.5. In this case we encounter $c[2]$ before $m[2]$. This situation happens when:

$$\langle A\rangle_i^* m[i] - B_{il} < 0 \tag{4.19}$$

which, using definition (4.15) corresponds to the condition:

$$\alpha_i > 0$$

as is shown in the appendix A.1. Not only do we need to use the original direction $e[i]$ but the equation of the bounding surface also is different. In this case the corner $c[i]$ described by:

$$\begin{array}{rcl} |A|_i c[i] - B_{il} & = & 0 \\ c[i] - (m[i] + e[i]s) & = & 0 \end{array} \tag{4.20}$$

We obtain:

$$X_{il} = c[i]_i = \frac{\alpha_i}{2e[i]_i - 1}$$

as shown in appendix A.1.

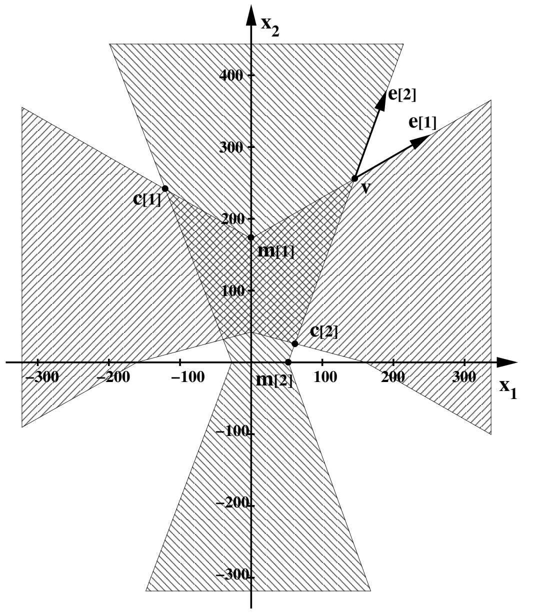

Figure 4.5: Exact Solution Algorithm for Equations (4.18)

We illustrate these computations for equations (4.18).

$$X_u = \langle A \rangle^{-1} B_u = \begin{pmatrix} 145.4545 \\ 254.5454 \end{pmatrix}$$

and:

$$\alpha = \begin{pmatrix} (-21.818 + 36.364)1.75 - 145.45 \\ (58.1818 + 109.091)2 - 254.545 \end{pmatrix} = \begin{pmatrix} -120 \\ 80 \end{pmatrix}$$

while $\alpha_1 < 0$, $\alpha_2 > 0$ so that:

$$X_l = \begin{pmatrix} -120 \\ \dfrac{80}{4-1} \end{pmatrix} = \begin{pmatrix} -120 \\ 26.666 \end{pmatrix}$$

The complexity of this algorithm is $\mathcal{O}(N^3)$ corresponding to the inversion of $\langle A \rangle$. Indeed the computation of all the lower bounds requires $\mathcal{O}(N)$, and is very fast. Figure 4.6 illustrates this method on a three dimensional problem.

Note that up to now we have assumed $B_{i_c} > 0$. For the general case we would have to make a simple change to the above results. We now have:

$$X'_{iu} = v'_i = \langle A \rangle_i^{-1} |B_u|$$

Defining:

$$\alpha'_i = 2|B_{ic}| e[i]_i - v'_i$$

we have:

$$X'_{il} = \begin{cases} \alpha'_i & \textit{for } \alpha'_i \leq 0 \\ \dfrac{\alpha'_i}{2e[i]_i - 1} & \textit{for } \alpha'_i > 0 \end{cases}$$

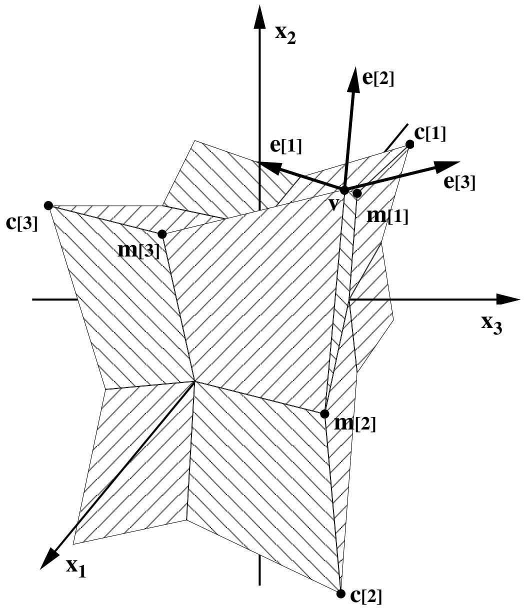

Figure 4.6: Three dimensional solution of Midpoint Preconditioned System

and finally:

$$X_i = sgn(B_{i\,c})X_i'$$

4.5 General Solution of Midpoint Preconditioned Interval Equations

The results of the preceding section can be modified to take advantage of initial domains and to deal with singular interval matrices. Unfortunately, at the time of writing it remains an open question whether the exact intersection of the initial domain with the solution can be computed in polynomial time. So in the algorithm described below, some overestimation can occur if the interval solution domain intersects the initial domain. Indeed, as was the case in section 4.3, the complexity of this problem doesn't allow us to select the best solution method at each step. Again, we have opted to improve on the Gauss-Seidel operator, while still obtaining the exact solution for large initial domains.

We start out by taking advantage of information concerning the initial domain only, i.e. we will assume for now that A is not singular. This implies that we can perform LU-decomposition on $\langle A\rangle$ so that the linear equation for v becomes:

$$\langle A\rangle v = LUv = B_v \tag{4.21}$$

In this equation we have adopted a more general notation B_v for the bounds of B which determine v. Using forward substitution equation (4.21) can be written as:

$$Uv = RB_v + C \tag{4.22}$$

where $R = L^{-1}$ also a lower triangular matrix. For reasons given below, we have included the constant vector $C \in \mathbf{R}^n$, which is zero initially. As before we can, without loss of generality, make the assumption that $B_{ic} > 0$. We now are in a position to determine the exact bounds on the n-th variable of the solution of the linear system. Indeed, we see from equation (4.22) that:

$$e[n]_n = \langle A \rangle_{nn}^{-1} = \frac{R_{nn}}{U_{nn}}$$

furthermore:

$$v_n = \frac{R_n B_v + C_n}{U_{nn}} \tag{4.23}$$

This allows us to compute α_n using equation (4.15):

$$\begin{aligned} \alpha_n &= 2B_{nc}e[n]_n - v_n \\ &= \frac{1}{U_{nn}}(2B_{nc}R_{nn} - R_n B_v - C_n) \end{aligned}$$

and so:

$$X_{nl} = \begin{cases} \alpha_n & \textit{for } \alpha_n \leq 0 \\ \dfrac{\alpha_n}{2e[n]_n - 1} & \textit{for } \alpha_n > 0 \end{cases} \tag{4.24}$$

We now have to decide whether to use information provided by the initial domain or by the n-th equation for the bounds on X_n in order to obtain the tightest solution. Unfortunately, this cannot always be decided perfectly, so that some overestimation can occur. We will return to this selection criterion later.

When the bounds on the exact solution are determined by the n-th equation, we simply project the solution on the corresponding subspace using backward substitution, where B_{nv} is either equal to $|B_{nl}|$ or to B_{nu}:

$$\begin{aligned} U_{ij}^{(1)} &= U_{ij} \\ R_{ij}^{(1)} &= R_{ij} - \frac{U_{in}}{U_{nn}} R_n \\ C_i^{(1)} &= C_i - \frac{U_{in}}{U_{nn}} C_n + (R_{in} - \frac{U_{in}}{U_{nn}}) B_{nv} \end{aligned}$$

for $i, j < n$.

If on the other hand the bounds of the initial domain constrain the exact solution, we proceed to use a fixed domain bound X_{nf} instead of the current equation to eliminate X_n:

$$\begin{aligned} U_{ij}^{(1)} &= U_{ij} \\ R_{ij}^{(1)} &= R_{ij} \\ C_i^{(1)} &= C_i - U_{in} X_{nf} \end{aligned}$$

for $i, j < n$.

In both cases the original system is reduced to the following system of dimension $n - 1$:

$$U^{(1)} v^{(1)} = R^{(1)} B_v^{(1)} + C^{(1)} \tag{4.25}$$

where the n-th variable has been eliminated, i.e. :

$$\begin{aligned} v_i^{(1)} &= v_i \\ {B_{iv}}^{(1)} &= B_{iv} \end{aligned}$$

where $i < n$.

We now apply this algorithm recursively to equation (4.25) until the solution has been bounded in all dimensions. Geometrically this can be interpreted as follows. When the n-th variable is eliminated by backsubstitution, one simply projects the exact solution of the linear interval system on the corresponding subspace. This is equivalent to the algorithm described in section 4.4 except for the occasional use of $|B_{nl}|$. The case where the variable is directly substituted for, is pictured in figure 4.7. Here the hypercube like solution is

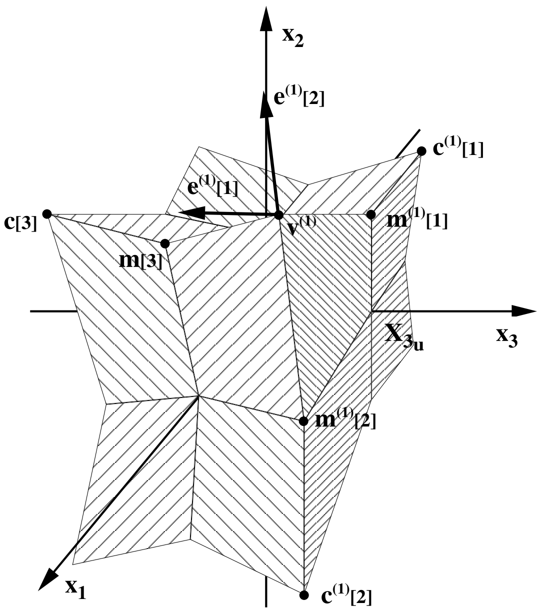

Figure 4.7: General Solution of 3 Dimensional Interval Equations

intersected with a hyperplane perpendicular to the n-th coordinate axis, located at X_{nf}. The solution in the corresponding subspace is therefore of a different shape then the one obtained by projection (see also figure 4.8).

Let us now deal with de selection criterion and the determination of the bounds of the n-th variable. We denote the bounds of X_n determined by equations (4.23) and (4.24) by $X_{n\sigma}$ and the bounds of the initial domain X_{nb}. Clearly, when:

$$X_{n\sigma u} \in X_{nb} \tag{4.26}$$

it is better to use the exact solution, i.e. to use:

$$X_n = X_{n\sigma} \cap X_{nb}$$

and perform backsubstitution using:

$$B_{nv} = B_{nu}$$

like before. If relation (4.26) does not hold, our decision will depend on the position of the intersecting hyperplane. As X_{nf} moves from $X_{n\sigma l}$ to $X_{n\sigma u}$ different solution shapes arise. This is illustrated in figure 4.8. The picture on the left shows a projection of the

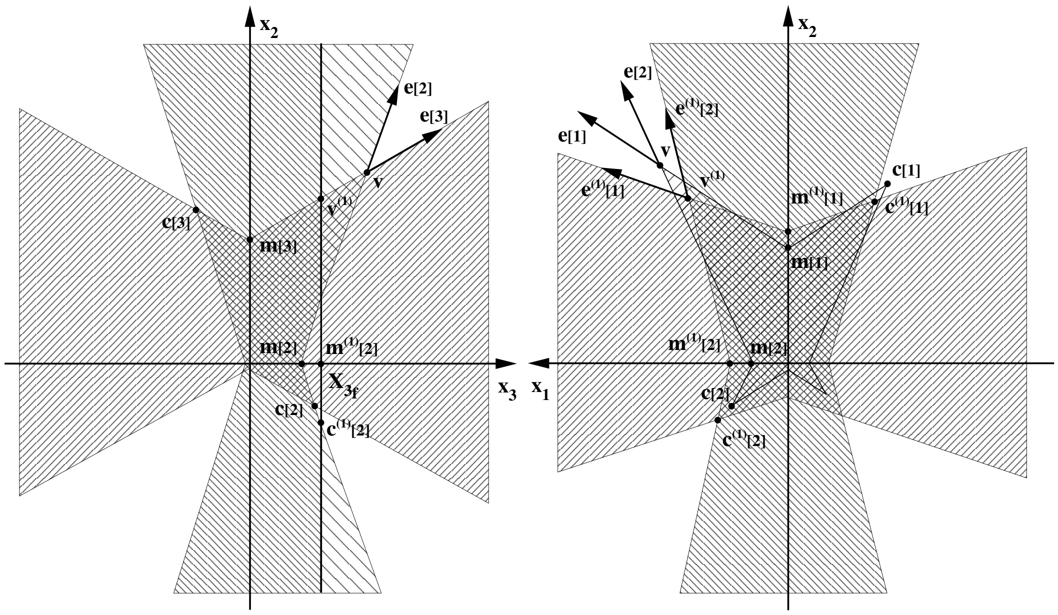

Figure 4.8: Intersection of the Exact Solution with a Hyperplane

hypercube like solution along coordinate X_1, the projection of this solution along coordinate X_3 and the shape of the intersection itself are shown on the right. Because the shape of the projection is not the same as the shape of intersection, partial overlap can occur. To obtain the exact solution we would need to use information of both shapes in further calculations. Unfortunately such a combined shape is not exactly representable within the framework of interval arithmetic, and would lead to exponential complexity.

As we indicated earlier, we set out to obtain a solution that is consistently better than the Gauss-Seidel solution. So in cases of partial overlap, we need to use the *inner* solution, corresponding with the shape of the intersection obtained by direct substitution rather than the *outer* solution corresponding with the projection obtained by backsubstitution.

With this in mind, one can verify that if:

$$X_{nb} \subseteq X_{n\sigma}$$

we should use:

$$X_n = X_{nb}$$

and perform direct substitution with:

$$X_{nf} = |X_{nb}|$$

This leaves with the situation where:

$$X_{n\sigma l} \in X_{nb}$$

Here we will use:

$$X_n = [X_{n\sigma l}\,,\, X_{nbu}]$$

but we need to consider four different cases as far as the subspace solution is concerned. First, when:

$$X_{n\sigma l} > 0$$

any intersection will contain the exact solution and we need to perform a direct substitution with:

$$X_{nf} = X_{nbu}$$

When:

$$X_{nbu} > |X_{n\sigma l}|$$

an overlap situation is possible, this is the case illustrated in figure 4.8. We have to use:

$$X_{nf} = X_{nbu}$$

and perform direct substitution. If it is the case that:

$$|X_{nbu}| \in [\beta_n\,,\, |X_{n\sigma l}|]$$

where we have defined β_n as:

$$\beta_n = \begin{cases} \dfrac{|B_{nl}|}{\langle A\rangle_{nn}} & \textit{for } B_{nl} \leq 0 \\ \dfrac{B_{nl}}{|A|_{nn}} & \textit{for } B_{nl} > 0 \end{cases}$$

we have no choice but to perform direct substitution with:

$$X_{nf} = |X_{n\sigma l}|$$

due to possible overlap with the projection of the opposite face of our hypercube like solution. Finally we are left with:

$$|X_{nbu}| < \beta_n$$

in which case we can safely perform backsubstitution using:

$$B_{nv} = |B_{nl}|$$

hereby projecting the face of the hypercube corresponding to B_{nl}.

The overestimation incurred by this method is due partly to the possible overestimation of the solution shape at each step, indeed depending on the bounds in the other directions it is still possible that the elimination operation produces better bounds then the substitution operation and vice versa. But also to the nature of the projection operation since the vertices of the exact solution might not belong to the projected solution.

After this analysis it is clear that the performance of Gauss-Seidel and modified Gauss elimination are not comparable, i.e. there are cases where one will be better then the other and vice versa. Due to the complexity of the exact solution, we tend to believe that there is no $\mathcal{O}(N^3)$ time algorithm that is consistently better than either of them. We could however conceive an $\mathcal{O}(N^3)$ algorithm that can be expected to yield best results *on average.* Instead of choosing direct or backward substitution, we could indeed generate an interval enclosure which minimizes the area of the solution in the current subspace.

Until now we have assumed that A is not singular. Indeed, it does not make sense to perform an LU-decomposition on $\langle A \rangle$ when A is singular; some L_{ii} would become negative. When during the LU-decomposition process negative pivots are encountered, we can simply use direct substitution for that variable, and reduce the dimension of the system by one. In this case the vector C will not be necessarily be zero at the beginning of our solution algorithm.

Geometrically, a negative pivot means that the corresponding equation does not bound the solution in that direction, so that the bounds of the initial domain need to be used.

Chapter 5

Fast Evaluation of Partial Derivatives

5.1 Introduction

Solving nonlinear systems requires evaluation of partial derivatives at every iteration and for every domain generated by the bisection strategy. We therefore believe that incorporation of efficient evaluation strategies in the compilation of partial derivatives can contribute significantly towards the reduction of the overall solution time.

We would like to note that the material proposed in this chapter is not restricted to arithmetic expressions. All algorithms apply without modification when more general operators replace the arithmetic ones, as long as local derivative information is available. In fact, they can be particularly useful for computational graphs involving vector and matrix operations.

The automatic computation of partial derivatives is usually done in a recursive fashion. As discussed in section 5.2 the backward approach can be more efficient. The problem of finding optimal evaluation strategies for partial derivatives on computational graphs is addressed in section 5.3.

5.2 Backward Evaluation Algorithm

In section 5.2.1 we describe the backward evaluation algorithm and how it can be used for interval differentiation. In section 5.2.2 we present the chain rule for interval slopes and use it to for backward evaluation in section 5.2.3.

5.2.1 Evaluation of Interval Partial Derivatives

As proposed in [190] automatic computation of partial derivatives is usually done recursively. This bottom up approach unfortunately has a complexity of $\mathcal{O}(nF)$ where n is the number of arguments and F is the complexity of one function evaluation. The forward computation of partial derivatives is not as efficient as it could be, indeed it involves incremental operations on vectors, and many of these operations would need to be performed only once per step.

This consideration led researchers to the more efficient backward, top down, computation of partial derivatives [232], a strategy very similar to the backpropagation algorithm in neural networks [210]. The complexity of evaluating both the function value and the partial derivatives reduces to CF, where the constant factor $C = 3$ as shown by Baur and Strassen [16]. Note that this result assumes that all operators are at most binary. Iri [108] analyzes this algorithm from the perspective of graph theory, and shows how the value of C depends on the different measures of complexity. A comparison between the different approaches together with some practical applications of this technique can be found in [77, 78].

Let us first introduce some terminology and notation. A directed graph G is a pair (N, A), where N is a finite set of nodes and A a finite set of arcs. An arc from x to y is an ordered pair (x, y). The notation $\alpha_G^-(x)$ is used to indicate arcs incident to node x, $\alpha_G^+(x)$ for the arcs leaving node x. Accordingly, the in-degree of x is $|\alpha_G^-(x)|$, the out-degree $|\alpha_G^+(x)|$. The maximum in-degree of G is denoted by $|\alpha_G^-|$, maximum out-degree by $|\alpha_G^-|$. Neighbors of x in the negative direction are denoted by $N_G^-(x)$, by $N_G^+(x)$ in the positive direction. Similarly, $R_G^-(x)$ is the set of nodes reachable from x in the negative direction, $R_G^+(x)$ in the positive direction. Finally, the sign will be dropped when we want to indicate the union of both positive and negative contributions, for example $R_G(x) = R_G^-(x) \cup R_G^+(x)$.

We will limit ourselves here to non-recursive computations, and therefore define computational graphs, such as arithmetic expressions, as directed acyclic graphs where nodes correspond to operators $\circ$ and arcs to data dependencies. Variables are associated with the result of each operator as well as with the inputs. Although strictly speaking there is a difference between the two, it is not needed to make that distinction here.

Evaluation of an computational graph is done through a bottom up traversal, starting from the independent variables the arcs are followed in the positive direction and arithmetic operations are performed as their arguments become available. This is illustrated in figure 5.1. For interval evaluation, all operators can simply be replaced by their interval arithmetic

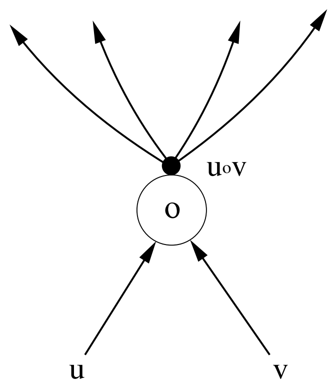

Figure 5.1: Evaluation of a Computational Graph

counterparts:

$$(u \circ v) \;\mapsto\; (U \circ V)$$

Using the chain rule of partial derivatives, backward evaluation can be expressed as follows:

$$\partial_x^f = \sum_{w \in N_G^+(x)} \partial_w^f \partial_x^w \tag{5.1}$$

The summation is performed over the nodes which directly depend on x, i.e. $N_G^+(x)$. This is illustrated in figure 5.2. First the partial derivatives of the function value f with respect

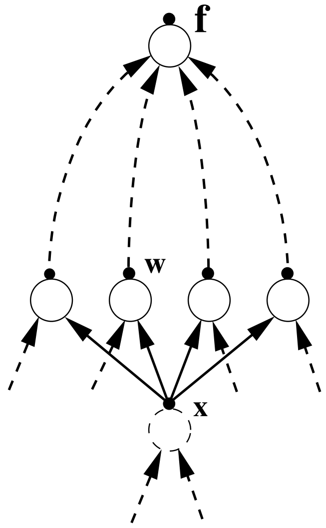

Figure 5.2: The Chain Rule for Partial Derivatives

to each node w are multiplied by the partial derivative of that node w.r.t. to its input x. The resulting contributions are then accumulated to yield the partial derivative of f w.r.t. to node x.

Equation (5.1) forms the basis of the backpropagation algorithm, which is illustrated in figure 5.3. Partial derivative contributions $(\partial_{u \circ v}^f)_w$ of all $w \in N_G^+(u \circ v)$ are added when node $u \circ v$ is reached. Backpropagation over that node corresponds with multiplication of the current value $\partial_{u \circ v}^f$ with the partial derivatives $\partial_u^{u \circ v}$ and $\partial_v^{u \circ v}$. The partial derivatives for $\circ \in \{+, -, *, /\}$ are:

$$\begin{array}{rclrcl}
\partial_u^{u+v} & = & 1 & \qquad \partial_v^{u+v} & = & 1 \\
\partial_u^{u-v} & = & 1 & \qquad \partial_v^{u-v} & = & -1 \\
\partial_u^{u*v} & = & v & \qquad \partial_v^{u*v} & = & u \\
\partial_u^{u/v} & = & 1/v & \qquad \partial_v^{u/v} & = & -u/v^2
\end{array}$$

Note that it is not needed to perform two divisions when backpropagating over a division node. As shown in [108] for example, one can perform local surgery on the computational graph to avoid unnecessary operations.

The backward evaluation algorithm requires that the values at the nodes be known. Evaluation of interval partial derivatives therefore consists of two parts. First, a forward evaluation is performed to initialize all nodes, including the initialization of the function node with the interval value of the function. This step is then followed by backward evalu-

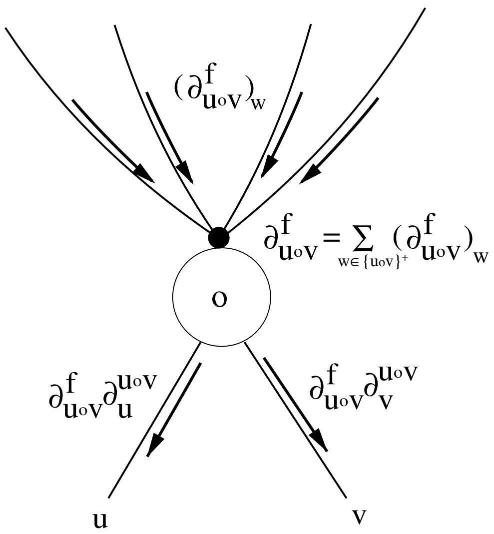

Figure 5.3: The Backward Evaluation Algorithm

ation of the partial derivatives from the function node f, starting with:

$$\partial^f_f = [1]$$

towards the independent variables. At this point, the partial derivative of the function w.r.t. each particular variable is available at the corresponding node.

Example 5.1

The execution of this algorithm on the expression `(+ (* (* x y) (+ y z)) x)` with $X = [2\,,\,3]$, $Y = [4\,,\,5]$ and $Z = [10]$ is illustrated in figure 5.4. First interval evaluation is performed, as shown on the left. It is followed by backward evaluation of the partial derivatives, depicted on the right.

Finally, we would like to observe that syntactic transformation of the compiled expressions can be used to avoid unnecessary multiplications or summations involving the constants 0 and 1, and single element summations.

Example 5.2

Compilation of the function:

```
(lambda (x y z) (+ (* (* x y) (+ y z)) x))
```

including the optimizing syntactic transformation, could for example produce the following `lambda`-expression: