HABILITATIONSSCHRIFT

Mathematical Modeling
and
Global Optimization

Hermann Schichl

November 2003

Contents

Draft of a Book, submitted to Cambridge University Press, assembled from material specially written for this book and the following publications

1. Hermann Schichl. Global optimization in the COCONUT project. In *Proceedings of the Dagstuhl Seminar "Numerical Software with Result Verification"*, Springer Lecture Notes in Computer Science, page 9, 2003. to appear.

2. Hermann Schichl and Arnold Neumaier. Exclusion regions for systems of equations. *SIAM J. Numer. Anal.*, page 29, 2003. to appear.

3. Hermann Schichl. Models and the history of modeling. In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization*, chapter 2, pages 25–36. Kluwer, Boston, 2003.

4. Hermann Schichl. Theoretical concepts and design of modeling languages. In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization*, chapter 4, pages 45–62. Kluwer, Boston, 2003.

5. Stefan Dallwig, Arnold Neumaier, and Hermann Schichl. GLOPT - a program for constrained global optimization. In I. Bomze et al., editor, *Developments in Global Optimization*, pages 19–36. Kluwer, Dordrecht, 1997.

6. Christian Bliek and Hermann Schichl. Specification of modules interface, internal representation, and modules api. Technical Report Deliverable D6, COCONUT project, August 2002.

7. Hermann Schichl. An introduction to the Vienna Database Library. Technical Report in "Upgraded State of the Art Techniques implemented as Modules", Deliverable D13, COCONUT project, July 2003.

8. Hermann Schichl. VGTL (Vienna Graph Template Library) version 1.0, reference manual. Technical Report Appendix 1 to "Upgraded State of the Art Techniques implemented as Modules", Deliverable D13, COCONUT project, July 2003. Version 1.1 (October 2003).

9. Hermann Schichl. The COCONUT API version 2.33, reference manual. Technical Report Appendix to "Specification of new and improved representations", Deliverable D5 v2, COCONUT project, November 2003. Version 2.13 (July 2003).

10. Hermann Schichl. Changes and new features in API 2.x. Technical Report in "Upgraded State of the Art Techniques implemented as Modules", Deliverable D13, COCONUT project, July 2003.

11. Hermann Schichl. Evaluators. Technical Report in "Upgraded State of the Art Techniques implemented as Modules", Deliverable D13, COCONUT project, July 2003.

12. Hermann Schichl. Uwien basic splitter, bestpoint, checkbox, check infeasibility, check number, exclusion boxes using karush-john conditions, karush-john conditions generator, linear relaxation generator using slopes, simple convexity, template for description of modules, tu darmstadt module donlp2-intv (with p. spellucci). Technical Report in "Upgraded State of the Art Techniques implemented as Modules", Deliverable D13, COCONUT project, July 2003.

13. Hermann Schichl. Management modules. Technical Report in "Set of Combination Algorithms for State of the Art Modules", Deliverable D14 of the COCONUT project, COCONUT project, July 2003.

14. Hermann Schichl. Report modules. Technical Report in "Set of Combination Algorithms for State of the Art Modules", Deliverable D14, COCONUT project, July 2003.

15. Hermann Schichl and Oleg Shcherbina. External converters. Technical Report in "Set of Combination Algorithms for State of the Art Modules", Deliverable D14, COCONUT project, July 2003.

16. Hermann Schichl et al. COCONUT environment 2.33 — open source solver platform for global optimization. Available from World Wide Web: `http://www.mat.univie.ac.at/coconut-environment`. 11.4 MByte source code, to be made officially available March 1st, 2004.

Preface

Optimization addresses the problem of finding the best possible choices with respect to a given target, not violating a number of restrictions.

In mathematical terms, optimization is the problem of minimizing (or maximizing) a prescribed function, the objective function, while obeying a number of equality and inequality constraints. Depending on the area of definition of these functions, one can differentiate various classes of optimization problems, continuous problems, discrete problems, and mixed-integer problems.

Since Danzig [40] 1947 invented the simplex algorithm, optimization problems have been solved in many contexts. Till today, a great number of all optimization problems used in industrial applications are linear.

Linear programming (solving linear problems) is special in many ways. One important property is that there is only one notion of optimality. Every point which is locally optimal (i.e. a point that satisfies all constraints and has the best objective function value of all such points in a neighborhood) is automatically globally optimal (i.e. has the absolutely best achievable objective within the given constraints). For non-linear problems this is in general far from true. Not all relations encountered in real life are linear, however, and non-linear functions are needed for the objective and/or the constraints to make the models fit reality better.

Solving nonlinear optimization problems traditionally means to search for a local optimum. Many applications nowadays are driven by economical interest (e.g., traveling salesman problem, packing problems, or cutting stock problems), and for earning money it usually is sufficient to gain an improvement of say 10% over the solution currently used, or used by the competitors. Solving local optimization problems can usually be done with moderate effort, even for industrial-size problems.

For a number of applications, however, knowing such a “good” feasible point is not enough, and for some other problems just *finding* a feasible point (i.e. a point satisfying all constraints) is exorbitantly difficult. There are problems in robotics (see Lee et al. [134, 135] and Neumaier[166]), in chemical engineering (cf. Floudas [53, 54], and Floudas et al. [56]), in protein folding (see Kostrowicki & Scheraga [124] and Neumaier [162]), in safety verification and worst case analysis, and in mathematics (e.g., Kepler's conjecture, see Hales [72]), which can only be solved by finding the *global* optimum of the problem.

Depending on the properties of the objective function and of the constraints, global optimization can be a highly non-trivial problem, in general, it is NP-hard. Therefore, for every generic algorithm there will, *very* likely, be problems for which finding the solution needs effort exponential in the size of the problem. This is a consequence of the famous $P \neq NP$ conjecture, see Sipser [211] or Baker et al. [9] for the definition of the conjecture, and Papadimitrou [177] and Pardalos & Schnitger [179] for the connection to optimization.

During the last decade the interest in global optimization has been ever increasing, partly because there are more and more applications, partly because the development of algorithms has already moved the field to the point where many small but already industrial-size applications can be solved reliably (see Shcherbina et al. [207]).

For a large number of applications even finding feasible points is difficult; in particular local optimization programs fail. Sometimes evaluating constraints or the objective function is expensive. For these situations a number of heuristic algorithms have been developed (see Jones et al. [105], Huyer & Neumaier [95, 96], and for a comparison of different heuristic methods Janka [100]). They perform an incomplete search guided by sophisticated concepts and usually end up at or near very good local optima, often at the global one. However, they can never be certain that they actually have found the global optimizer.

For safety applications, chemical phase and equilibrium problems, and mathematical applications incomplete methods are not applicable, because there it is required to find the global optimum with certainty. Performing a complete search, covering the whole feasible set, needs an effort exponential in the problem dimension if implemented naively. To overcome that difficulty, research groups have started to attack the field from various directions.

The basic principle, the *branch-and-bound method*, is common to most of these approaches. The search region is recursively split into smaller pieces. For each of these pieces the algorithm tries to prove that it does not contain the global optimum (bound step), if it succeeds, the piece is discarded, and if not the piece is broken into even smaller pieces (branch step). Once all pieces have either been discarded or have reached a very small size, the algorithm finishes.

The main difference in the approaches to global optimization taken by the different research teams lies in the bounding step. Depending on their tradition they use different methods for analyzing subproblems during the branch-and-bound scheme.

Methods based on *Interval Analysis* compute bounds on objective function and constraints using interval arithmetic and mean value forms with interval derivatives or slopes. Areas around local optima are handled by interval Newton methods like the Krawczyk iteration.

If the objective function is constant (or equivalently missing), the global optimization problem becomes a *constraint satisfaction problem.* For those problems a tradition exists, that comes from *constraint logic programming.* In the logic programming languages `Prolog V` and `ECLiPSe`, discrete constraint satisfaction problems are formulated. They are considered solved if one feasible point has been found. The solution algorithms use special combinations of backtracking and *constraint prop-*

agation for tracking down the solution. Combining these techniques with intervals leads to the generalized form of constraint propagation used for attacking continuous constraint satisfaction problems. Every global optimization problem can be reduced to a sequence of constraint satisfaction problems, so constraint propagation is a relevant technique for solving those problems, too.

Very successful for solving global optimization problems are *relaxation* techniques. In this approach the complicated nonlinear problems are replaced by linear or convex relaxations, which can easily be solved by standard software. Since they are relaxations, their solutions provide lower bounds on the possible values of the objective function for a given subproblem. If this lower bound is higher than the function value of the best point found so far, the subproblem can be discarded.

There are even more ideas, like symbolic transformations, semidefinite relaxation, methods from algebraic geometry like Groebner bases, which have been successfully applied to solving global optimization problems.

In the past all the research teams have pursued their techniques with limited interaction, only. The most interesting fact about all those approaches is that they complement each other. It is not like in local optimization, where the different classes of algorithms compete. In global optimization, the various ideas can be combined, and there are synergetic effects. E.g., the combination of constraint propagation, interval analysis and linear underestimation produces considerably stronger linear relaxations than any single method could achieve.

This book is intended as a short introduction to modeling and solving global optimization problems, keeping the integration of techniques from various fields in mind. The main focus of the book will be on complete methods, which can be used to yield guaranteed results. The techniques presented in this book require that objective function and constraints are given by mathematical formulas. There will be no discussion on black-box optimization and surrogate modeling. To make the integration technically feasible, the book also contains an exposition of an open-source solver platform for global optimization, which was developed during the COCONUT project [36] funded by the European Community.

The book, and the software platform presented in here, can be seen as an effort to provide the technological basis for the global optimization community to join their forces, and hopefully, this will yield the development of global optimization solvers which can solve bigger problems faster and more reliably.

The book is divided into two main parts: The first part concentrates on the theoretical background of the field. The second part presents the open solver platform for global optimization problems, the COCONUT environment.

Readers of this book should have a basic knowledge in numerical analysis ([38, 164]) and optimization ([67, 174]). The contents are kept compact but they should be self-contained. For further reading on global optimization I want to refer the reader to the survey by NEUMAIER [168].

Whoever wants to solve an optimization problem for an application faces the problem of translating his application into mathematical form, such that the resulting problem can be solved using a suitable algorithm. The application has to be modeled mathematically. Chapter 1 starts with a short historic review of models and modeling. After the introduction, *mathematical modeling*, the process of translating an application to a mathematical model is analyzed. The process is split into six subtasks, which are strongly interconnected and have to be reconsidered iteratively until a mathematical model is found, which describes the application with suitable accuracy and can be solved by available algorithms. Having found the mathematical model is not enough, however. The next task is the translation of the model into the input structure of the solution algorithm. This is a tedious task, and so there has been effort to provide computer support for the translation step, as well. *Mathematical modeling languages* (for a full review see KALLRATH [108]) have been invented exactly for that purpose, and nowadays a huge amount of real applications and test problems have been modeled in one of the most widely used modeling languages, `AMPL` [57] or `GAMS` [29]. Section 1.2 contains a theoretical introduction into the basic design features of algebraic modeling languages and a small example of their use. Unfortunately, the interaction of modeling languages with global optimization algorithms is not yet fully satisfactory. The most stringent difficulties for the connection between most modeling languages and global solvers are discussed in Section 1.3. The final section of the Chapter, Section 1.4, presents mathematical models of four important classes of global optimization problems, for which complete search is essential.

Chapter 2 starts with the theoretical background on nonlinear optimization, first defining the local optimization problem, then reviewing the basic theorems. Every solution of a global optimization problem, of course, has the property that it is locally optimal, too. A local optimizer of an optimization problem is, of course, a *feasible point*, i.e., a point satisfying all constraints, whose objective function value is minimal in a neighborhood of the point within the *feasible region*, i.e., the set of all feasible points. Computing local optima of an optimization problem is much easier than finding the global ones. So, a first step for solving the global optimization problem could be to compute local optima, with decreasing function values, until there are no more local minima. Section 2.1 discusses the most important facts from convex analysis, since for convex optimization problems, finding local and global minima is equivalent. The notion of duality is also presented in that section. Essential for the design of any local optimization algorithm are the theoretical results on conditions for local optimality of points, both sufficient and necessary ones. Those results are discussed in Section 2.2.

Chapter 3 is about the theory of global optimization. An important step in every optimization problem is to establish the existence of a feasible point. This can, e.g., be done by heuristic optimization. A review of the most important algorithms in this field is presented in Section 3.2. Afterwards, the branch-and-bound scheme is explained. The following sections of the chapter provide short introductions into the different methods for computing bounds and excluding subproblems. Section 3.4 contains a short introduction to interval analysis, on automatic range computation for functions and derivatives, and on the interval Newton method, Krawczyk's operator, and the Moore-Skelboe algorithm. A technique which originally stems from constraint logic programming is constraint propagation. This very powerful approach and its generalization from the discrete to the continuous case is discussed in Section 3.5. Linear relaxations are another efficient way of solving global optimization problems by sequential linear programming. The foundations for algorithms based on this technique are explained in Section 3.6. The next Section 3.7 focuses on a very important and very successful class of global optimization algorithms. These are based on convex underestimation and subsequent local optimization. BARON, the currently strongest global optimization solver, is based mainly on convexity techniques. Algorithms which approximate functions by piecewise linear functions, and solve mixed integer linear programs, those which are based on reformulation, and those using algebraic geometry are not explained here. An excelent summary of these methods can be found in the global optimization survey by Neumaier [168].

Chapter 4 concludes the theoretical part by introducing some new techniques developed during the COCONUT project. In Section 4.1 a method for representing global optimization problems by directed acyclic graphs (DAGs) is described. The representation is flexible, and has the advantage that it is equally fit for function evaluation, automatic differentiation, and constraint propagation. Evaluation schemes and constraint propagation are described in detail (Sections 4.1.3 and 4.1.4), and it is shown that the combination of both yields improved estimates on the ranges of functions and their derivatives (Section 4.1.5). A method for automatically computing linear relaxations of the optimization problems is presented in Section 4.1.6.

The second part of the book consists of three chapters describing the open source software platform for global optimization, the COCONUT environment.

Chapter 5 contains a detailed description of the environment, the basic C++ classes of the API, and the open-source solver kernel. Sections 5.1 and 5.2 describe two fundamental libraries, the VGTL (**V**ienna **G**raph **T**emplate **L**ibrary) an STL–like library for storing and manipulating graphs, and the VDBL (**V**ienna **D**ata**B**ase **L**ibrary), an in-memory database system, whose tables can hold columns of arbitrary types. Section 5.3 describes the main part of the application programmer's interface, expressions, models, and the search graph. A special section (5.6) is devoted to the description of the evaluators, which are especially important for the inference engines, described in Chapter 6. The description of the management modules follows in Section 5.8, these modules are used for manipulating the internal representation of the optimization problems, the search database, and the search graph. Section 5.9 is devoted to the report modules, which produce all output of the algorithm, while Section 5.7 describes the base class for the solver modules, which in detail are described in the following chapter. The last section of the chapter is devoted to the strategy engine, the central component of the environment, which has been developed by IRIN, University of Nantes. It directs the solution strategy, combining the various *inference engines* (the solver modules) in a user programmable way.

Chapter 6 describes a simple strategy, jointly developed by the University of Vienna and the EPFL Lausanne for solving a global optimization problem, together with a description of the public domain inference engines (solver modules) and some envelopes for commercial components, which are included in the environment.

- *Basic Splitter:* This module computes simple splits and tries to identify the hardest variables.
- *Box Covering Solver (BCS):* BCS computes coverings of the feasible area for

low dimensional constraint satisfaction problems.

- *Box Chooser:* This graph analyzer selects the next box from the search graph.
- *Linear Programming:* Three different linear solvers have been wrapped as inference engines, `CPLEX`, the high-end commercial linear solver by ILOG, `XPRESS-MP`, another high-end commercial LP solver by Dash Optimization, and the public domain LP solver `LP solve`.
- *Constraint Propagation:* Two different constraint propagation engines, `HULL`, a hull consistency solver provided by IRIN, and `C_propagator`, a priority driven constraint propagator.
- *Local Optimization:* There are envelopes for three different local optimization engines. `donlp2` by P. Spellucci from TH Darmstadt is a fast SQP solver using dense linear algebra, `ipfilter` by the University of Coimbra, a robust filter code, and `sSQP` also by the University of Coimbra, an SQP algorithm based on sparse linear algebra, which uses the sparse QP solver by P. Spellucci.
- *Exclusion Boxes:* This solver module computes exclusion boxes around local solutions using linear information only.
- *Karush-John condition generator:* This module computes symbolic derivatives on the expression DAG and generates the DAG for the Karush-John necessary first order optimality conditions.
- *Linear Relaxations:* Using the methods discussed in Section 4.1.6 this inference engine computes a linear relaxation of the optimization problem.
- *Convexity Detection:* This solver module tries to identify convex and concave functions and constraints in the DAG utilizing the methods of MAHESHWARI ET AL. [139].
- *STOP:* A heuristic global optimization engine based on a mixture of constraint propagation and multilevel coordinate search provided by IRIN.

The complete reference manuals for the COCONUT API and the basic libraries, the `VGTL` and the `VDBL`, would go beyond the scope of this book and can be accessed online at `http://www.mat.univie.ac.at/coconut-environment`. However, those `C++` definitions, which are directly referenced in Chapter 5, can be found in Appendix A.

I hope that this book and the solver platform will spark the interest of many research groups working on global optimization to join forces, to contribute to this platform, and to together advance the field of global optimization beyond the limits it is confined in today.

This book would not have been possible without the help of many other people. I want to thank the other COCONUT participants, especially Christian Bliek, Eric Monfroy, Brice Pajot, Christophe Jermain, Viet Nguyen, and Xuan-Ha Vu, who helped designing, developing and debugging the software, my colleagues working during the project at the Institute of Mathematics of the University of Vienna, Waltraud Huyer, Boglarka Toth, Oleg Shcherbina, Andrzej Pownuk, and Tamas Vinko, for their help and their tireless testing, and above all I want to thank Arnold Neumaier for his support, invaluable source of ideas, and his friendship. Last but not least I thank my wife Waltraud and my children Arthur and Konstanze for tolerating my lack of time and occasional bad mood during the preparation time of

this book.

Vienna, Austria Hermann Schichl
November 2003

Contents

List of Figures

Notation

We will need some standard notation from set theory. Notably, the following symbols for standard sets of numbers will be used.

$\emptyset$	the empty set
$\mathbb{N}$	the set of all *natural numbers*: $\{0, 1, 2, \dots\}$,
$\mathbb{Z}$	the set of all *integer numbers*: $\{\dots, -2, -1, 0, 1, 2, \dots\}$,
$\mathbb{R}$	the set of all *real numbers* (the *real line*),
$\mathbb{R}_*$	the *extended real line*: $\mathbb{R} \cup \{-\infty, \infty\}$,
$\mathbb{C}$	the set of all *complex numbers*,

the symbol $\in$ shall describe the is-element relation $x \in X$, and in some cases we will write $X \ni x$.

For two sets $A \subseteq B$ shall denote that A is a subset of B, here $A = B \quad\Longrightarrow\quad A \subseteq B$. If we explicitely exclude the case $A = B$ for the subset-relation the notation $A \subsetneq B$ will be chosen.

For two sets A and B the set $A \cup B$ denotes the *union*, $A \cap B$ the *intersection*, $A \setminus B$ the *set difference*, and $A \times B$ the *cartesian product* of A and B. For $A \times A$ we write A^2 and more generally define $A^n := A^{n-1} \times A$. Furthermore, for a finite set A we denote by $|A|$ the number of elements in A.

Vectors and Matrices

An element $x \in \mathbb{R}^n$ is called a *real (column) vector* of dimension n. We write

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix},$$

and call the x_i the *components* of x. For vectors we will need the three *norms*

$$\|x\|_1 := \sum_{i=1}^{n} |x_i|,$$
$$\|x\|_2 := \sqrt{\sum_{i=1}^{n} |x_i|^2},$$
$$\|x\|_\infty := \max\{x_i \mid i = 1, \dots, n\}.$$

The absolute value of a vector will be defined componentwise

$$|x| := \begin{pmatrix} |x_1| \\ \vdots \\ |x_n|, \end{pmatrix}$$

and the comparison operators $\leq$, $<$, $>$, and $\geq$ are likewise defined componentwise, e.g.,

$$x \leq y : \iff x_i \leq y_i \quad \text{for all } i = 1, \dots, n.$$

For vectors of dimension two or more, this defines only a partial order. For a subset I of $\{1, \dots, n\}$ we write x_I for the vector of dimension $|I|$ whose components are those components of x which have indices in I.

The standard operations $+$ and $-$ for vectors are defined componentwise, there is no ordinary product of column vectors. However, we define $x \star y$ as the componentwise product of x and y:

$$x \star y := \begin{pmatrix} x_1 y_1 \\ \vdots \\ x_n y_n \end{pmatrix}.$$

For two vectors x and y we set

$$\sup(x, y) := \begin{pmatrix} \max\{x_1, y_1\} \\ \vdots \\ \max\{x_n, y_n\} \end{pmatrix}, \quad \text{and} \quad \inf(x, y) := \begin{pmatrix} \min\{x_1, y_1\} \\ \vdots \\ \min\{x_n, y_n\} \end{pmatrix}.$$

For $y \in \mathbb{R}$ we set $y_+ := \max\{0, y\}$ and $y_- := \max\{0, -y\}$, and for a vector $x \in \mathbb{R}^n$ we define the vectors

$$x_+ := \sup(0, x), \quad x_- := \sup(0, -x).$$

The set $\mathbb{R}^{m \times n}$ is defined as the set of all real matrices with m rows and n columns

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn}. \end{pmatrix}$$

For subsets I of $\{1,\dots,m\}$ and J of $\{1,\dots,n\}$ the expression $A_{I:}$ ($A_{:J}$) defines the submatrix of A composed of the rows (columns) with indices in I (J), and A_{IJ} is the submatrix whose entries are restricted to those with row indices in I and column indices in J. We identify the set $\mathbb{R}^{m\times 1}$ with the column vectors $\mathbb{R}^m$. For $A \in \mathbb{R}^{m\times n}$ the matrix $A^T \in \mathbb{R}^{n\times m}$ denotes the *transposed* matrix.

Like for vectors we define for matrices $|A|$, $A \le B$, $\inf(A,B)$, $\sup(A,B)$, and $A \star B$.

The triangle inequality implies the following relations:

$$\begin{array}{ccc}
|x+y| \le |x|+|y|, & |x\star y| = |x|\star|y|, & \inf(x,y) \le x,y, \\
\sup(x,y) \ge x,y, & |A+B| \le |A|+|B|, & |A\star B| = |A|\star|B|, \\
\inf(A,B) \le A,B, & \sup(A,B) \ge A,B, & |x^Ty| \le |x|^T|y|, \\
|Ax| \le |A|\,|x|, & |AB| \le |A|\,|B|, & |x^TAy| \le |x|^T|A|\,|y|.
\end{array}$$

Intervals

The set $\boldsymbol{x} := [\underline{x},\overline{x}] \subseteq \mathbb{R}$ for $\underline{x},\overline{x} \in \mathbb{R}_*$ is the (closed) interval $\{x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x}\}$. We denote the set of all real intervals by $\mathbb{IR}$. Sometimes we will need intervals, for which certain border points are not included. For $\underline{x} \in \mathbb{R}$ we define $(\underline{x},\overline{x}] := \{x \in \mathbb{R} \mid \underline{x} < x \le \overline{x}\}$ and analogously $[\overline{x},\underline{x})$ and $(\underline{x},\overline{x})$.

Now take vectors $\underline{x},\overline{x} \in \mathbb{R}^n_*$. The *box* $\boldsymbol{x}$ is defined as $\boldsymbol{x} = [\underline{x},\overline{x}] = \{x \in \mathbb{R}^n \mid \underline{x} \le x \le \overline{x}\}$. We have $\boldsymbol{x}_i = [\underline{x}_i,\overline{x}_i]$ and $\boldsymbol{x} = \boldsymbol{x}_1 \times \cdots \times \boldsymbol{x}_n$. For the set of all n-dimensional boxes we write $\mathbb{IR}^n$.

In the same way, we define $\boldsymbol{A} = [\underline{A},\overline{A}]$ for matrices $\underline{A},\overline{A} \in \mathbb{R}^{m\times n}$ and $\mathbb{IR}^{m\times n}$.

Basics in topology

For a norm $\|\ \|$ on $\mathbb{R}^n$, $0 < r \in \mathbb{R}$, and a vector $x \in \mathbb{R}^n$ we definine the *open (norm)ball* with radius r and center x as

$$D_r(x) := \{y \in \mathbb{R}^n \mid \|y-x\| < r\}.$$

For a subset $X \subseteq \mathbb{R}^n$ we fix the following terminology: An element $x \in X$ is called an *interior point* of X if for any norm $\|\ \|$ on $\mathbb{R}^n$ there exists $r > 0$ with $D_r(x) \subseteq X$. The point is called an *exterior point* of X if it is an interior point of the complement $\mathbb{R}^n \setminus X$, and it is a *border point* otherwise.

The set $X \subseteq \mathbb{R}^n$ is *open*, if it contains only interior points, and it is *closed*, if its complement is open. The *interior* $\operatorname{int}(X)$ of X is the set of all interior points of X, the *border* ∂X of X is the set of all its border points. We define the *closure* $\overline{X} := \operatorname{int}(X) \cup \partial X$.

Functions

Let $X \subseteq \mathbb{R}^m$ and $Y \subseteq \mathbb{R}^n$.

The set of all continuous functions $f : X \to Y$ will be denoted as $C(X, Y) = C^0(X, Y)$.

We define set of all k-times continuously differentiable functions $f : X \to Y$ as $C^k(X, Y)$. A function $f \in C^k(X, Y)$ will in short be called a C^k function.

Throughout this book the objective function of optimization problems usually will be called f, and its gradient $g = \nabla f$. The Hessian matrix of second derivatives will be denoted $G = \nabla^2 f$.

For a function $F : \mathbb{R}^n \to \mathbb{R}^m$ we define $F(x)'$ as the Jacobian of F at x.

Part I

Theoretical background

Chapter 1
Mathematical Modeling

1.1 Models

An important part of part of the mathematician's work is translating phenomena of the real world into his or her own language. This process is called *modeling*, and a huge amount of literature for various levels of education is dedicated to the discussion of this theme, e.g., Williams [227], Kallrath [107], Bossel [28], Geoffrin [63], Bender [12], Giordano [68], or Meerschaert [147].

1.1.1 History

The word “model” has its origin in the Latin word *modellus*, a diminutive form of *modulus*, the word for measure or standard. The old Italian derivation *modello* referred to the mould for producing things. In the sixteenth century the word was assimilated in French (*modèle*), taking its meaning as a small representation of some object, spreading into other languages like English and German.

The associated verb “modeling” describes a typical human way of coping with the reality. Anthropologists think that the ability to build abstract models is the most important feature which gave homo sapiens a competitive edge over less developed human races like homo neandertalensis.

Although abstract representations of *real-world objects* have been in use since the stone age, a fact backed up by cavemen paintings, the real breakthrough of modeling came with the cultures of the Ancient Near East and with the Ancient Greek.

The development of philosophy in the Hellenic Age and its connection to mathematics lead to the deductive method, which gave rise to the first pieces of *mathematical theory*. Starting at about 600 BC, *geometry* became a useful tool in analyzing reality, and analyzing geometry itself sparked the development of mathematics independently of its application. Pythagoras of Samos is said to have been the first pure mathematician, developing among other things the theory of numbers, and most important initiating the use of *proofs* to gain new results from already known theorems.

Important philosophers like Aristotle and Eudoxos followed, and the summit was reached by Euclid of Alexandria at about 300 BC when he wrote *The Elements*, a collection of books containing most of the mathematical knowledge available at that time. The Elements held among other the first concise axiomatic description of geometry and a treatise on number theory. Euclid’s books became the means of teaching mathematics for hundreds of years, and around 250 BC Eratosthenes of Cyrene, one of the first “applied mathematicians”, used this knowledge to calculate the distances Earth-Sun and Earth-Moon and, best known, the *circumference of the Earth* by a mathematical/geometric model.

A further important step in the development of modern models was taken by Diophantus of Alexandria about 250 AD in his books *Arithmetica*, where he developed the beginnings of *algebra* based on symbolism and the notion of a *variable*.

Building models for real-world problems, especially mathematical models, is so important for human development that similar methods were developed independently in China, India, and Persia.

One of the most famous Arabian mathematicians is Abu Abd-Allah ibn Musa Al-Ḫwārizmī (late 8th century). His name, still preserved in the modern word *algorithm*, and his famous books *de numero Indorum* (about the Indian numbers — today called Arabic numbers) and *Al-kitab al-muḫtaṣar fi ḥisāb al-ǧabr wa’l-muqābala* (a concise book about the procedures of calculation by adding and balancing) contain many mathematical models and problem solving algorithms (actually the two were treated as the same) for real-life applications in the areas commerce, legacy, surveying, and irrigation. The term algebra, by the way, was taken from the title of his second book.

In the Occident it took until the 11th century to continue the development of mathematics and mathematical models, in the beginning especially for surveying, architecture, and arts.

It is important to note that despite preceding work of Diophant and Al-H̱wārizmī the systematic use of variables was really invented by Vietá (1540–1603), and it took another 300 years, until Cantor and Russell, that the true role of variables in the formulation of mathematical theory was fully understood.

Starting with Newton, *Physics* and the description of Nature's principles became the major driving force in modeling and the development of the mathematical theory. Hand in hand with the discovery of new mathematical principles, the complexity of the mathematical models increased.

Nowadays, lots of disciplines, not only from the hard sciences, use mathematical models to analyze data. There are among many others important applications in Anthropology, Archaeology, Arts, Economics, Finance, Linguistics, Medicine, Political Sciences, and Psychology.

See, e.g., Gericke [64, in German] for a much broader and more detailed view of the history of mathematics and mathematical modeling.

1.1.2 Mathematical Models

As a basic principle we may say:

A *model* is a *simplified version* of something that is real.

The traits of the model can vary according to its use. They can vary in their *level of formality*, *explicitness*, *richness in detail*, and *relevance*. The characteristics depend on the basic function of the model and the *modeling goal*.

In building models, everything starts with the *real-world object* we are considering. In a *model* real-world objects are replaced by other, simpler objects, usually carrying the same names. The *knowledge* we possess about the real-world is *structured* by the model, and everything is *reduced* to those phenomena and aspects which are considered important. Of course, a model can only describe a part of the real-world phenomenon, and hence its usefulness is restricted to its *scope of application*.

Models can have many different functions, they can, e.g., be used to explain phenomena, make predictions, or communicate knowledge. They give hints for the solution of problems, provide a thorough understanding of the modeled objects, and can be used as guides or even normative instances.

The models of interest for us are mathematical models. Here, the real world object is represented by mathematical objects in a *formalized mathematical language*.

The advantage of mathematical models is that they can be analyzed in a precise way by means of *mathematical theory* and *algorithms*.

As mentioned in Section 1.1.1, mathematical models have been used long ago, and many problems have been formulated in a mathematical way since a couple of hundred of years. However, the enormous amount of computational work needed for solving the models restricted their use to qualitative analysis and to very small and simple instances.

The improvement in algorithms since the turn of the last century and the fast development in computer technology since ENIAC was started in 1945 have made it possible to use mathematical modeling for *solving* practical problems of significant size, making it increasingly attractive for military and industry, and a special class of problems, *optimization problems*, became very important.

The success in solving real world problems increased the demand for better and more complex models, and the modeling process itself was investigated. There are now several books on modeling, e.g., [228] and [107], and branches of mathematics completely devoted to solving practical problems and developing theory and algorithms for working on models, like operations research. See also [91] for a short treatise of this theme.

Mathematical modeling is the art of finding a mathematical formulation, which is on the one hand tractable by the theory and the algorithms available, and on the other hand resembles the real-world object well enough to provide insight, answers and advice for the application of interest.

As every textbook on modeling describes, the process of building a model from a real-world problem is a tedious task. It often involves many iterations and interactions. Traditionally, this process is described as a cycle like the one of Figure 1.1.

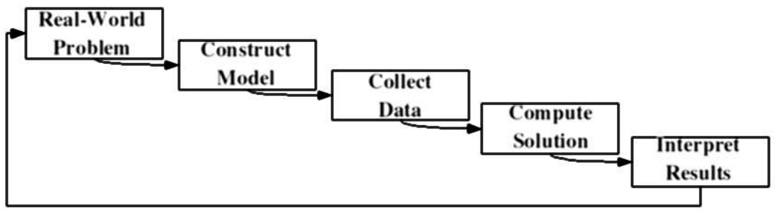


Figure 1.1. *Modeling cycle*

However, as was shown in NEUMAIER [167], the various stages of the *modeling cycle* are much more interconnected, demanding even more interaction between the subtasks. The author points out that the nodes of the diagram of Figure 1.2 represent information to be collected, sorted, evaluated, and organized.

The edges of the diagram represent the flow of relevant information between the corresponding sources of information, a two-way interaction rather than a one-way process like in the modeling cycle.

In analyzing the diagram lets start at the nodes.

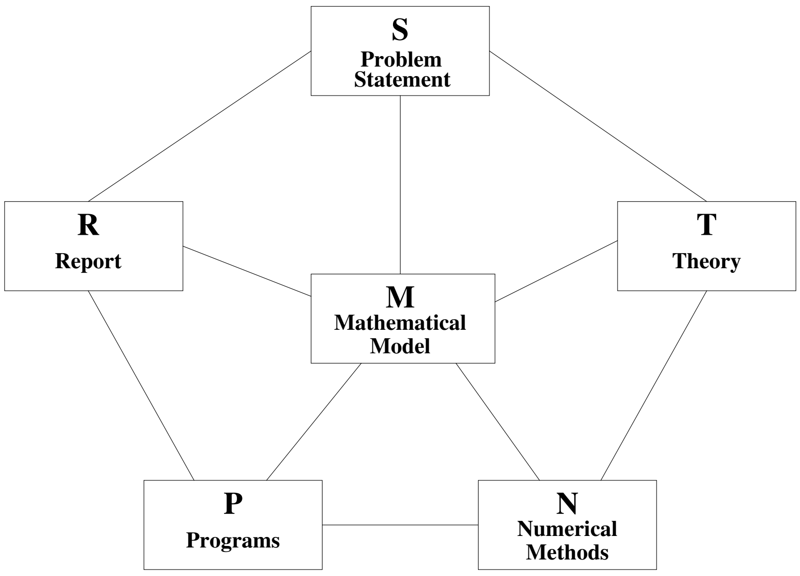


Figure 1.2. *Modeling Diagram*

S. Problem Statement. Usually, the problem to be solved comes from the real world. Often, a company will be interested in a specific application, perhaps a design or an improvement in product quality or productivity. In order to end up with a useful problem formulation, strong interaction with the end-user is necessary. The first obstacle to overcome is the language differences between the modeling mathematician and the user. The modeler has to acquire enough knowledge about the field of the application to ensure that the chances for misinterpretation and misconception are small.

If the communication problem is mastered, the modeler next faces the difficulty that the problem specification he or she receives from the end-user is ambiguous and most of the time misses important aspects. Academic training and later experience leads to implicit knowledge. In expert discussions within the field, this knowledge usually is not repeated since it is shared between all participants. Therefore, the end-user normally does not mention those parts of the problem formulation which are related to the implicit knowledge when communicating with the modeler. So the problem formulation tends to remain incomplete, sometimes ambiguous. In many cases, the end-user himself does not have a clear enough view (from the mathematical point of view) of the problem to be modeled, making the modeling process even more tedious.

If the problem is put into a complete, formalized state, the next hurdle arises, that the wishes the end-user has are usually not well compatible or even contradictory. E.g., an oncologist designing a radiation plan for the radio therapy of cancerous tumors wants to destroy all malignant cells without destroying healthy tissue. However, if the tumor is at an unfavorable position, this might not be possible. The end-user will then have to specify which wishes are more important than others, and to what extent he or she is willing to set aside some of them.

M. Mathematical Model. The mathematical model is written in very clear, unambiguous terms and formal language. To date a mathematical model consists of *concepts* like

variables: These represent *unknown* or changing parts of the model, e.g., whether to take a decision or not (decision variable), how much of a given product is being produced, the thickness of a beam in the design of a ceiling, an unknown function in a partial differential equation, an unknown operator in some equation in infinite dimensional spaces as they are used in the formulation of quantum mechanics, etc.

relations: Different parts of the model are not independent of each other, but connected by relations usually written down as *equations* or *inequalities*. These pose *restrictions* on the variables and the data. E.g., the amount of a product manufactured has influence on the number of trucks needed to transport it, and the size of the trucks has an influence on the maximal dimensions of individual pieces of the product. The speed of flow of a fluid is connected with the pressure gradient, as is the form of the tube.

data: All numbers needed for specifying instances of the model. E.g., the maximal forces on a building, the prices of the products, and the costs of the resources, or the viscosity of a fluid.

In the formal description of a model we will also have to determine the modeling goals, and the range of its application. Finally, we will have to specify the minimal quality assignments needed for data and results for making most use of the model.

T. Theory. In order to produce a useful model, the modeler has to become firm in both the theoretical background in the subject of the application and, of course, in the mathematical structures associated with it. Literature Search, which has become much more efficient in the Internet era, is a very important part of the modeling process, and end-user interaction should not be neglected. However, it is essential that the modeler keeps its scientific independence from the end-user. Otherwise, the danger is high that the model is too strictly governed by the traditions of the field of application rather than by a balance between them and the mathematical necessities and the most recent mathematical developments.

Understanding the mathematical theory well is important for various reasons. First, being able to prove qualitative statements about the model helps on the one hand to choose the proper setting for numerical methods, and on the other hand helps to assess quality and correctness of the computational results.

Second, essential for the applicability of a model in real-life situations is, whether it can be used to solve problems of industry-relevant sizes. Whether this is possible greatly depends on the solution time of the algorithm chosen.

Until the middle of the twentieth century most of the mathematical models were used only to describe phenomena and to make qualitative statements about the real world problems described. Since then the situation has changed dramatically. The tremendous increase in computing power has shifted the interest in mathematical models more and more from problem *description* towards problem *solving*.

This has an important consequence for mathematical modeling itself and for the structure of the models: If the most important conclusions which can be drawn from a model are qualitative statements, which are derived by analytic means using a lot of mathematical theory, it is important to formulate the model in a very concise way, especially tailored to the analytical methods available.

Although modern algorithms are making more and more model classes solvable, it can still make a huge difference which model formulation from a set of mathematically equivalent ones is taken, especially if solution time is considered. Choosing the "right" *structure* is a matter of experience and insight, and it distinguishes good modelers from bad ones.

N. Numerical Methods. The more practical issues regarding model solving have to be considered at this node of the diagram. If the model is of industry-relevant size, methods have to be found which can compute the solutions in a time interesting for the application. The maximum time available for computing the solution will have an influence on the model complexity.

The urge for numerical solutions in the last decades, made it necessary to change the model structure, to adapt it to the solution algorithms available.

The modeler faces many difficult decisions. The model structure will, e.g., be greatly influenced by the available software libraries, and the license costs. For most mathematical problems free software from the Internet is available. However, commercial products usually provide greater flexibility, stability, performance, and support.

Special care has to be taken for data processing, especially in the situation of Section 3.1.1, where algorithms are used as part of a mathematical proof. If the problem size is huge, the available computer memory has to be taken into account, etc. As we see, many constraints influence the modeler's decisions.

P. Programs. Once numerical algorithms are available, they have to be implemented on a computer for performing all calculations. The first step is to design the program and its components properly. The user interfaces for data input and for data visualization have to be developed, best together with the end-user, and a test set for debugging has to be prepared.

If the algorithm is straightforward and the data handling is simple, a very basic design using flow charts and a monolithic structure will be sufficient. However, if the tasks increase in complexity a more professional approach to algorithm design should be taken (see, e.g., SOMMERVILLE [215]). One possibility would be the use of an UML (universal modeling language, not to be confused with mathematical modeling languages as described in Section 1.2) based CASE (computer aided software engineering) tool like `Rational Rose` [184], `Borland Together` [26], `Enterprise Architect` [46], or `AllFusion Component Modeler` [2]. A list of most available tools can be found in [132]. As a rule of thumb, it can be said that the time invested into a proper design pays off twice during the implementation phase.

A very important part in the development of programs, which is often neglected in mathematical software, is the documentation phase. The usability and maintainability of a program rises with the quality of its documentation. Also debugging and testing is simplified if proper documentation is written.

The user interface has to be described in detail to enable the end-user to train operators for the software even if the original designer is no longer available.

R. Report. A very important part of modeling is the proper design of reports generated from the model. Again, this is a task which needs strong interaction with the end-user. In the translation step from problem statement to mathematical model the modeler has put lots of effort into replacing application specific terms by mathematical objects. In order to make the model useful for the end-user, the reverse translation has to be performed in report generation.

The most important feature of a report is that it provides the information which the end-user needs for his application. The results computed from the input data by the program often are presented using visualization techniques. For the end-user the result reports often generate deeper insight into the application, show limitations, or provide recommendations, either by themselves or with the help of the modeler.

Another very important aspect is that the reported results can help to validate the model. An end-user, using his experience, usually has a good feeling what to expect from the model. If the results differ considerably, there is a good chance that the model is lacking important constraints or still contains modeling errors. In any case, a thorough analysis is needed.

Next we will focus on the diagram's edges. The ten edges, which represent the flow of information have to be considered repeatedly. Working on an edge usually enriches both connected nodes simultaneously, requiring additional work there. The state of the nodes changes in time, and so information traverses the whole diagram spreading to all the nodes. During this process the problem frequently changes as the model develops. The vague parts become clearer, the hidden knowledge unveils; the initial ambiguities or conflicts soften — ideally they vanish completely, or if they don't a clear priority list becomes available, giving some relations and restrictions precedence over other ones.

When the modeling process advances, the conditions at all nodes in the modeling diagram ameliorate and stabilize. This reduces the tension and the flow of information at all the diagram edges. If flow and tension on all edges have reached insignificant levels, the modeling process can be considered complete. Latest at this point, even the vaguest and most contradictory initial problem should have turned into a reasonably well-defined statement; then the mathematical model in the center is precisely defined, its scope of application is described, and its inherent inaccuracies are determined. At the same time, the theoretical background will give information on the qualitative behaviour of the model, and the numerical methods for solving the problem will have been identified. A design of the algorithm to be implemented will have been fixed, together with the system and the available software libraries.

When the modeling process is finished and the end-user is satisfied with the reports produced, the modeler can step back. For those, who just want to use the algorithms, only the left part of the modeling diagram remains as in Figure 1.3.

As we have seen, several of the *modeling* steps require the help of the end user. For complex models, it is widely accepted that computers are needed for *solving*

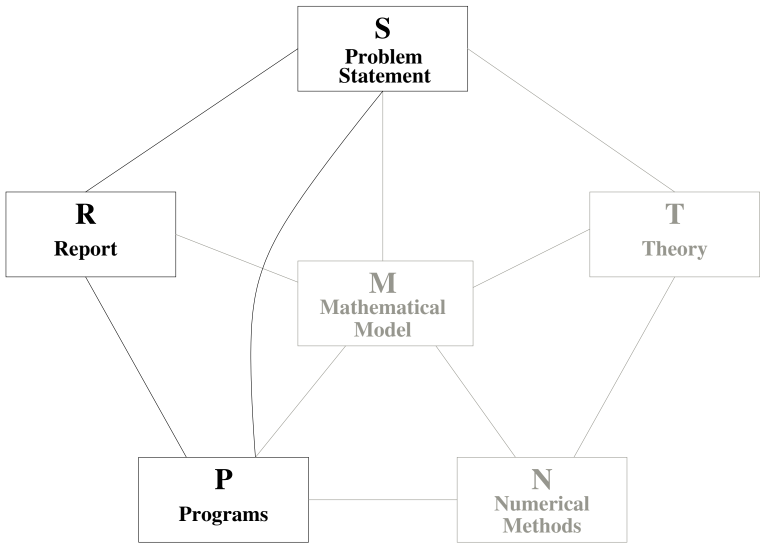


Figure 1.3. *The part of the model needed in the finished application*

the problem. For huge data sets or data which has to be retrieved from different sources, computer support for *collecting* the data is accepted, as well.

1.2 Mathematical Modeling Languages

In this section we will shed light on the assistance computers can give for translating mathematical models from paper into the special input format the solution algorithms require.

As stated before, there is an important consequence from computer-assisted model solving. Somebody has to translate the model into a form accessible by the computer.

Lets look again at the modeling diagram in Figure 1.2. In reality, the edges in the diagram contain many steps which have not been explicitly described before. If we consider the revised and more detailed model of the modeling process depicted in Fig. 1.4, we observe that some of the additional steps in model building and solving involve translations from one format to another: (*translate model to solver input format*, *translate data to solver input format*, *write status report*). These tasks are full of routine work and error prone. Furthermore, since during the reformulation process sometimes the solution algorithm, and hence the program, is changed, these "rewriting" steps have to be performed again and again.

A special case is the task to *construct derived data*. Many solvers require data which is a priori not part of the model, but also not provided by the solver. E.g., *gradients*, sometimes *Hessians*, of all functions involved are needed for an efficient local optimization algorithm (see Chapter 2). Computing these data usually is neither boring nor straightforward but needs mathematical work.

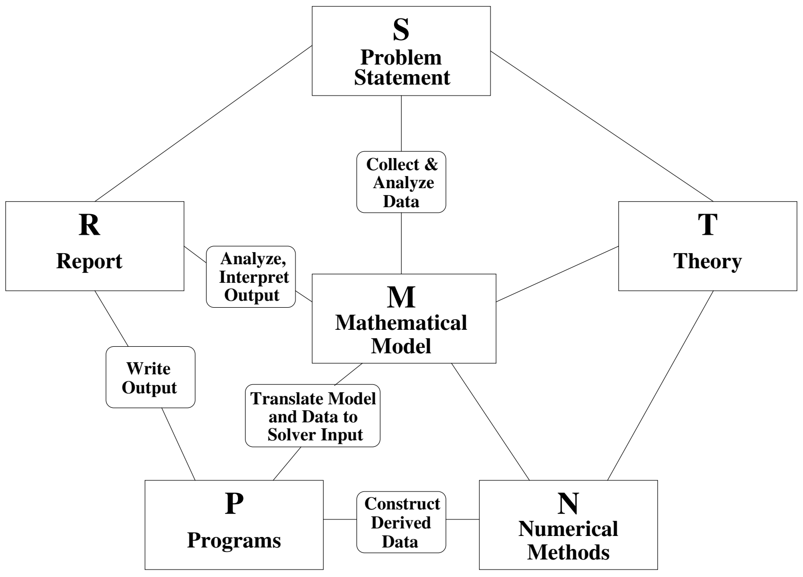


Figure 1.4. *Detailed modeling cycle*

In the early days of application solving all these translation steps were performed by hand and *programs* were produced, which represented the mathematical model in an algorithmic way. Data was stored in hand-written files. This was very inflexible and error prone. Maintenance of the models was close to impossible, and the models were neither very scalable nor re-usable in other applications. This made the modeling process very expensive. In addition, a lot of expert knowledge in mathematics, modeling, and software engineering was necessary for building applicable models.

Many of the important models for military applications and for the industry were *optimization problems*, so methods were designed to reduce the modeling costs and to reduce the error rate in the translation steps. This led to the development of modeling languages and modeling systems. The next section will describe their most important features.

1.2.1 Basic Features

The development of modeling languages started in the late 1970s when `GAMS` [29] was designed, although there had been some precursors before.

Since that time it is known how all the error prone routine tasks in Fig. 1.4 can be performed by the computer. The process of modeling has become much more convenient, and the flexibility has increased a lot.

In a modeling language, the model can be written in a form which is close to the mathematical notation, which is an important feature of an *algebraic modeling language* (see Section 1.2.2).

The formulation of the model is *independent of solver formats*. Different *solvers* can be connected to the modeling language, and the translation of models and data to the solver format is done automatically. So the routine translation steps are done by the computer, and, after thorough testing of the interface, errors are very unlikely.

In a modeling language, models and model data are kept separately. There is a clear division between the model structure and the data. Therefore, many different instances of the same model class with varying data can be solved. Many systems provide an ODBC (open database connectivity) interface for automatic database access and an interface to the most widely used spreadsheet systems. This relieves the user from the laborious duty of searching for the relevant data every time the model is used. A second advantage of this concept is that during the development phase of the model, the approach can be tested on *toy problems* with small artificial data sets, and later the model can be applied without change for industry-relevant large scale instances with real data.

The computation of *derived data* can be automatized using *automatic differentiation*. Most modeling systems nowadays can generate derived information (gradients, sparse Hessians,...) from the model description without assistance of the user.

Today, there are several types of modeling systems in use. We will analyze the most prominent type, the algebraic modeling languages, here. A thorough description of the most important modeling languages available together with some historical and theoretical background can be found in [108].

1.2.2 Algebraic Modeling Languages

The most important representatives of this biggest group of modeling languages are `GAMS` [29], `AMPL` [58], `Xpress-MP` [7], `LINGO` [204], `NOP` [171], `NOP-2` [203], `Numerica` [85], and `MINOPT` [206].

In [93] and [92] Tony Hürlimann describes why algebraic modeling languages can be viewed as a new paradigm of programming. Usually programming languages are divided into three different classes: *Imperative* or *procedural* languages like `C`, `C++`, `Pascal`, `FORTRAN`, and `Java`, *Functional Languages* like `LISP`, and *Logic Programming Languages* like `Prolog V` and `ECLiPSe`.

All languages from these classes specify a problem in an algorithmic way. Like in Babylonian mathematics, it is not specified *what the problem is* but rather *how to solve the problem.* For this reason we can subsume these languages under the general term *algorithmic languages*.

In contrast to that, modeling languages store the *knowledge* about a model, they *define the problem* and usually do not specify how to solve it. They are *declarative languages*, specifying only the properties of a problem: Starting with a state space X (usually $\mathbb{R}^n \times \mathbb{Z}^m$), they specify a set of *constraints* by mathematical formulas (usually *equations*, *inequalities*, and *optimality requirements*), which together define a relation $R : X \to \{\text{true}, \text{false}\}$. We say $x \in X$ is *admissible*[1], if $R(x) = \text{true}$,

[1] I have chosen the word *admissible*, since in optimization the word *feasible* is used to describe all points which obey the equality- and inequality constraints but not necessarily the optimality

and we define the *mathematical model* $M := \{x \in X \mid R(x)\}$ for the problem we are considering. This leads to a formulation which is very similar to the modern mathematical approach. However, since no solution algorithm is specified, there is a priori no reason why a problem defined in the declarative way should be solvable at all, and there is definitely no reason why there should be an algorithm for finding a solution.

Fortunately, however, there are large classes of problems (like *linear programming*) for which algorithms to explicitly find the set M exist, and the number of solution algorithms for various problem categories grows steadily.

If we direct our attention again to declarative languages, we can identify their three most important aspects:

- Problems are represented in a declarative way.
- There is a clear separation between problem definition and the solution process.
- There is a clear separation between the problem structure and its data.

Algebraic modeling languages are a special class of declarative languages, and most of them are designed for specifying *optimization problems*. Usually they are capable of describing problems of the form

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & F(x) = 0 \\ & G(x) \leq 0 \\ & x \in \mathbf{x}, \end{array} \tag{1.1}$$

or something slightly more general, or constraint satisfaction problems, the special cases with $f \equiv 0$. Here $\mathbf{x}$ denotes a sub-box of $X = \mathbb{R}^m \times \mathbb{Z}^n$.

The problem is flattened, i.e. all variables and constraints become essentially one-dimensional, and the model is written in an *index-based* formulation, using *algebraic expressions* in a way which is close to the mathematical (actually the TEX) notation. Conceptually similar entities like *variables*, *parameters*, and *constraints* are grouped in *sets*. The entities in the sets are later referenced by *indices* to the elements of those sets. Groups of entities (variables, constraints) can then be compactly represented and used in algebraic expressions.

In `AMPL`, for instance, the expression

$$\sum_{i \in S} x_i^2$$

would be written as

```
sum { i in S } x[i]**2;
```

requirement.

This leads to a problem formulation which is very close to a mathematical formulation. Therefore, translation of mathematical models to declarations in a modeling language usually involves syntactic issues only.

The algebraic modeling language is then responsible for creating a *problem instance* that a solution algorithm can work on. This automatic translation step relieves the user from translating the mathematical model into some queer internal solver representation.

As a small example we consider the problem

$$\begin{aligned} \min \quad & x^T A x + a^T x \\ \text{s.t.} \quad & Bx \le b \\ & \|x\| \le c \\ & x \in \boldsymbol{x}, \end{aligned} \tag{1.2}$$

with an $N \times N$-matrix A and an $M \times N$-matrix B.

Writing down all matrix products in index form we transform problem (1.2) into the “flat” or “scalar” model

$$\begin{aligned} \min \quad & \sum_{i=1}^{N} \Big(\sum_{j=1}^{N} A_{ij} x_j + a_i \Big) x_i \\ \text{s.t.} \quad & \sum_{j=1}^{N} B_{ij} x_j \le b_i \qquad \text{for all } i = 1, \dots, M \\ & \sqrt{\sum_{i=1}^{N} x_i^2} \le c \\ & x_j \in [\underline{x}_j, \overline{x}_j] \qquad \text{for all } j = 1, \dots, N. \end{aligned} \tag{1.3}$$

This model can then easily be specified in AMPL:

```
### PARAMETERS ###
param N>0 integer;
param M>0 integer;
param c;
param a {1..N};
param b {1..M};
param A {1..N,1..N};
param B {1..M,1..N};
param xl {1..N};
param xu {1..N};
### VARIABLES ###
var x {1..N};

### OBJECTIVE ###
minimize goal_function:
  sum {i in 1..N} (sum {j in 1..N} A[i,j]*x[j] + a[i]) * x[i];
```

```
### CONSTRAINTS ###
subject to linear_constraints {j in 1..M}:
   sum {i in 1..N} B[j,i]*x[i] <= b[j];
norm_constraint:  sqrt(sum {j in 1..N} x[j]^2) <= c;
box_constraints {j in 1..N}: xl[j]<=x[j]<=xu[j];

################ DATA #####################
data sample.dat;

###########################################
solve; display x;
```

In the `AMPL` description, we can easily identify the various parts of the flat model, the declarations of parameters and variables. The actual model data is to be read from the file `sample.dat`. The last line contains the only *procedural statements*: `solve` which calls the solver, and `display x` which prints the solution.

We conclude with a short summary on the most important design features of algebraic modeling languages:

- Variables, constraints with arbitrary names,
- Sets, indices, algebraic expressions (possibly non-linear),
- Notation close to the mathematical formulation,
- Data independent problem structure,
- Models scalable (by a change of the parameters a switch can be made from *toy problems* to *real-life problems.*
- Declarative statements, except for *conditionals*, *loops*, and very few procedural statements,
- Flexibility in the types of models that can be specified,
- Convenient for the modeler (little overhead in model formulation. . .),
- Simple interface between modeling language and solver
    - It must be easy to connect a solver to the modeling language and to have easy access to the model, the data, and the derived information like derivatives.
    - Sending data back from the solver to the modeling language like progress reports, solutions, or error messages should be straightforward.
- Simple and powerful data handling (ODBC, spreadsheet interfaces, data files,. . .),
- Automatic differentiation.

For a more thorough analysis see [59].

There are also completely different approaches to computer aided modeling, especially in areas where models are highly structured, or in constraint logic programming, where the solution process cannot be easily separated from the model itself, because generic solution algorithms would be inefficient.

There is also nowadays a trend to *Integrated Modeling Environments*, where a more graphical approach to model building is taken, relying almost completely on *graphical user interfaces* (GUI), and a database representing the model. The model building process is performed in a menu-driven way, and they contain strong visualization tools, as well. `AIMMS` and `OPL Studio` by ILOG are typical representatives.

Recently, several languages try to bridge the gap between the algorithmic languages used in constraint programming and the declarative languages mainly used in mathematical optimization. The `Mosel` [78] approach overall looks much like a programming language, even the model declaration part has to be programmed, whereas the language `LPL` [94] by Hürlimann provides a declarative part which looks similar to mathematical notation, while the algorithmic part has the look and feel of an imperative programming language.

1.3 Modeling Languages and Global Optimization

Modeling languages were first developed to support linear programming applications. As the computing power increased and the solution algorithms became more effective, higher and higher dimensional nonlinear programs were made accessible. The modeling systems had to keep up with the development, and ultimately this was one reason for the emergence of algebraic modeling languages. The algorithms available solved *local optimization* problems in an approximate way, so most of the now available modeling systems provide very good connectivity for linear solvers and local optimization algorithms.

There are, however, applications which cannot be solved by local approximation. E.g., *computer assisted proofs* (see Section 3.1.1), certain chemical engineering problems (see Section 1.4.1), and protein folding (see Section 1.4.2). Most of these problems are global optimization problems.

When solving such models harmful approximations must not be performed, roundoff errors have to be avoided or controlled, and a lot of techniques have to be applied which are not usually necessary for local optimization. The most important tool is *interval arithmetic* (see Section 3.4) with *directed rounding*.

Compared to models with low requirements in rigor, for global optimization much more care has to be taken in data handling. For computer-assisted proofs all *data* items must be *known exactly. Round-off* in the translation of the input data from modeling system to the solver can destroy important model properties. E.g. the very simple system

$$\begin{aligned} 0.4x + 0.6y &\geq 1 \\ x, y &\in [0,1] \end{aligned} \tag{1.4}$$

has exactly one solution $x = y = 1$. The system becomes infeasible when rounded to IEEE floating point representation.

For other problems the *data* might be *uncertain*, see Section 3.1.2. For example the elasticity module for steel bars can vary by more than 15%. For solving problems with uncertain data it is important that all parameters can be entered together with their amount of uncertainty.

Most modeling systems around have originally been developed for nonlinear local optimization and linear programming. When trying to connect a solver for global optimization problems to such a modeling language, the programmer usually runs into one or all of the following problems.

The difficulties start with the fact that most modeling systems pass a presolved problem, and all input data is rounded to floating point representation. Sometimes it is not even possible to decide whether a datum the solver sees is integer or only some real number very close to an integer. This involves round-off and a loss of important knowledge and makes mathematical rigor impossible.

The second point is that the performance of global optimization algorithms usually depends on whether specific structural knowledge about the problem can be exploited. Unfortunately, the problem which is passed from the modeling system to the solver is “flat” and has lost most of its mathematical structure, except for the sparsity pattern.

Automatic differentiation is not sufficient for constructing all derived information needed for a global optimization algorithm. There is no inherent support for *interval analysis*, and hence by the current generation of modeling systems essential data such as implied bounds cannot be generated automatically (except in `Numerica`).

There is no support for matrix operations, so all matrix operations have to be flattened, hence the structure is lost (consider, e.g., the constraint $A^TDAx = b$). A number matrix constraints cannot be used inside a model at all, like $\det(A) \geq c$, or A is *positive semidefinite*, specifying these constraints is impossible in all modeling languages I know of.

Finally, many global optimization problems can only be solved by using several *mathematically equivalent formulations* of the same model at the same time. A very nice example for that principle from an application in robotics can be found in [136]. There is no modeling language which supports equivalent formulations.

Therefore, the global optimization community has developed its own input languages or modeling languages, like `Numerica` [85], `NOP` [171], or `NOP-2` [203].

1.4 Applications

Many applications in optimization nowadays are driven by economical interest. It is not a coincidence that another term for the objective function, especially in linear programming is *cost function*. In most of the economical applications, like in logistics (the Traveling Salesman Problem) or cutting stock applications, it is necessary to find a “good” solution, i.e., a feasible point with sufficiently low objective function value. Usually, an improvement over the solution currently used in the company by 10% is more than enough to satisfy the expectation of the company’s CEO. Finding the globally best point usually is not required.

On the other hand, there are indeed applications, for which it is essential to find the global minimum, and to prove its globality.

- There are certain non-linear optimization or feasibility problems, for which finding a local optimum is sufficient. Due to the strong non-linearities in problem formulation, sometimes local solvers have extreme difficulties in detecting a feasible point. This can be because they are stuck in local minimizers of the merit function, many times because the Lagrange multipliers are unbounded. In these cases, complete search can be used to stabilize the local search by identifying good starting points. An example for such a hard feasibility problem from a robotics application can, e.g., be found in LEE ET AL. [134, 135].
- Many problems from mathematics are global optimization problems, e.g., the *maximum clique* problem of finding the maximal number of mutually adjacent vertices in a graph Γ is equivalent to an indefinite quadratic optimization problem, which was proved by MOTZKIN & STRAUSS [153]. Here A is the adjacency matrix of Γ, and e is the vector of ones.
  Other problems from mathematics, like Kepler's conjecture will be presented in Section 3.1.1.
- The Human Genome Project [90] is trying to identify the approximately 30.000 genes in the human DNA. It has already sequenced the chemical base pairs that make up human DNA. However, this information is not sufficient for real applications like better pharmaceutical products and the like. Most of the biological processes in the human body are caused or at least influenced by proteins, strings of aminoacids. The *3D structure* of those proteins is the reason for their function, and their failure. The base pair information only determines the sequence of aminoacids in a protein. Determining the 3D structure from this sequence is a highly non-trivial task, involving a global optimization problem, see Section 1.4.2.
- Predicting the fluid phase behaviour of most chemical separation processes depends on the knowledge of the number of phases existing at the equilibrium state, and the distribution of the chemical components in these phases has to be known as well. The knowledge has to be very accurate, or a process simulator will not be able to capture the behaviour of a chemical plant properly. The chemical phase and equilibrium problem is a global optimization problem, which must be solved by a complete method. Non-global local minima, even good ones, usually lead to wrong solutions and bad predictions, see also Section 1.4.1.
- Worst case studies are another field of application for global optimization. If the structural integrity and stability of a bridge or building has to be ensured, usually finite element models are used. In many applications the maximal forces and stresses in the various structural parts are determined by local optimization in combination with Monte-Carlo simulation. BALAKRISHNAN & BOYD [10] have shown that this might severely underestimate the true maximal forces, posing the threat that structures might break for

In the following, some models will be presented as an example for the modeling process, of course in polished form. Some more models for applications requiring global optimization are presented in Section 3.1.

1.4.1 Chemical Engineering

In chemical engineering various optimization applications exist. Many of them require the solution of a global optimization problem. This is the reason, why research groups in chemical engineering are among the most active and most successful groups in the global optimization community. There are many published global optimization problems from chemical engineering, see e.g. Floudas et al. [56], Esposito & Floudas [47], Floudas [55].

S. Problem Statement

An important application in chemical engineering is the simulation of chemical processes, one class of them are the separation processes, e.g., liquid chromatography, capillary electrophoresis, and flow injection analysis. Separation techniques are inevitable for most real chemical analysis, because most samples are mixtures far too complex to be analyzed by any direct method.

The behaviour of such a separation processes in the fluid phase depends on the chemical equilibrium, the number of different phases existing at the equilibrium state, and the distribution of the components of the mixture in these phases.

So, our task will be to predict the chemical equilibrium, the number of phases, and the distribution of the components in the phases for a given mixture for a liquid phase system.

T. Theory

The equilibrium problem has already drawn attention in the early days of optimization. Dantzig et al. [42] have tried to attack the problem by linear programming methods.

A survey through the literature will reveal a number of articles and books devoted to the subject, e.g., Gibbs [65, 66], Smith & Missen [214], and Baker et al. [8]. From those we can see that today the problem is modeled via the Gibbs free energy function. For defining that term, first we need to recall some thermodynamics.

There are four *thermodynamic potentials*, which are useful in the description of chemical reactions and non-cyclic chemical processes, we will need two of them. The *internal energy* U of a system is the energy associated with the random, disordered motion of molecules. It must not be confused with the macroscopic ordered energy related to moving objects, but it refers to the invisible microscopic energy on the atomic and molecular scale, i.e., vibration of chemical bonds, fast movement of molecules and their collision,.... If you consider the interaction with the environment, for creating a system you have to invest the internal energy needed plus the energy required for making room for it, pV is that contribution. We also have to take into account the spontaneous energy transfer to the system from the environment. This contribution is related to the absolute temperature T of the environment and to the entropy S (a measure of the amount of energy which is unavailable to do work); the term is TS. If we sum up all energy contributions, we end up with the *Gibbs free energy*

$$G = U + pV - TS.$$

The Gibbs free energy describes the amount of work needed for creating the given system in an environment of constant temperature T from a negligible initial volume.

At constant temperature and pressure the equilibrium condition for a chemical process is that the Gibbs free energy attains its global minimum, subject to the usual chemical constraints like elemental and molar balances.

We are interested in the phase and equilibrium problem for liquid phases. For modeling those, we find (see [56, Section 6.2.2]) that the Gibbs free energy is expressed in terms of the activity coefficient for liquid phases. The vapor phases can be assumed ideal. Temperature T and pressure p are considered fixed.

M. Model

A proper model for this situation (from [56, Section 6.2.2]) can be written as follows:

Let C be set of components at fixed temperature and pressure, and let P be the set of phases postulated at the equilibrium state. The components themselves are composed of a set E of elements. Now write

- n_c for the total number of moles of component $c \in C$, and write n for the vector of n_c,
- b_e for the number of elements $e \in E$ in the mixture, write b for the vector consisting of those numbers.
- A_e^c for the number of elements $e \in E$ in component $c \in C$, and write A for the matrix composed of those numbers,
- e^P for the the vector of ones indexed by elements of P.

In addition we introduce variables

- N_c^p for the unknown number of moles of component $c \in C$ in phase $p \in P$, and N for the matrix composed of the N_c^p.

Then the law of conservation of atoms states that the *elemental balance equation* for reacting systems

$$ANe^P = b \tag{1.5}$$

holds, and the law of conservation of mass implies the *molar balance equation* for non-reacting systems

$$Ne^P = n. \tag{1.6}$$

Of course, we have $N \geq 0$ component wise and

$$N_c^p \leq n_c \quad \text{for all } p \in P \text{ and } c \in C; \tag{1.7}$$

this constraint is a consequence of $N \geq 0$ and (1.6) in the case of a non-reacting system.

The objective function of the optimization problem is the Gibbs free energy function

$$G = \operatorname{tr}(N\mu), \tag{1.8}$$

where μ is the matrix of chemical potentials μ_p^c of component c in phase p, and tr denotes the trace of a matrix. These μ can be modeled as follows, if the vapor phase is assumed ideal and the liquid phases P_L are described by an activity coefficient equation:

$$\frac{\mu_p^c}{RT} = \begin{cases} \frac{\Delta G_{p,f}^c}{RT} + \log \frac{\hat{f}_p^c}{f_{p,0}^c} & p \in P \setminus P_L \\ \frac{\Delta G_{p,f}^c}{RT} + \log \frac{\hat{f}_p^c}{f_{p,0}^c} + \log \gamma_p^c & p \in P_L, \end{cases} \tag{1.9}$$

where $R = 8.31451\, m^2\, kg/s^2\, K\, mol$ is the gas constant, and T is the (constant) system temperature. $\Delta G_{p,f}^c$ denotes the Gibbs free energy for forming component c in phase p. The $\hat{f}_c^p$, $f_{p,0}^c$, and γ_p^c depend on the components involved and the phases present.

N. Numerical Analysis

There are texts suggesting solutions, e.g., Trangenstein [221], Jiang et al. [103], McDonald & Floudas [144, 145], and McKinnon & Mongeau [146], and they provide different approaches. We learn from them that incomplete methods tend to fail because they become stuck in one of the numerous local minima. Hence, we need a complete method (see Section 3.3).

Complete methods nowadays still have the disadvantage that the problem has to be good structured in order to be solvable at all. If the mathematical structure is bad, methods which process the whole search space need exponential effort in the problem dimension.

P. Programs

There are a number of complete methods around. A very successful one is αBB by Androulakis, Maranas & Floudas [3], which works very well for biconvex problems[2].

T. Theory

In Floudas et al. [56, Sections 6.3 and 6.4] we find that for two liquids, Renon & Prausnitz [185] have derived NRTL equation for representing liquid-liquid immiscibility for multicomponent systems using binary parameters only, and that McDonald & Floudas have managed to transform the equation into biconvex form.

[2]A biconvex problem has the following property: The set of variables can be partitioned into two subsets X and Y. If all variables in either of these sets is fixed, the problem is convex in the other variables.

M. Model

The Gibbs free energy formulation for a liquid–liquid mixture in the biconvex formulation of the NRTL equation can be found below.

$$\begin{aligned}\min_{N,\Psi} G = &\sum_{p\in P}\sum_{c\in C} N_c^p\left(\frac{\Delta G_{p,f}^c}{RT} + \log\frac{N_c^p}{\sum_{c'\in C} N_{c'}^p}\right)\\ &+\sum_{c\in C}\sum_{p\in P_L} N_c^p\left(\sum_{c'\in C}\mathcal{G}_{cc'}\tau_{cc'}\Psi_c^p\right)\end{aligned}$$

where

$$\mathcal{G}_{cd} = e^{-\alpha_{cd}\tau_{cd}}.$$

In addition to the standard constraints found above, we have the constraint due to the variable substitution (additional variable Ψ_c^p), which is necessary for the transformation to biconvex form:

$$\Psi_c^p\left(\sum_{c'\in C}\mathcal{G}_{cc'}N_{c'}^p\right) - N_c^p = 0, \quad \text{for all } c\in C \text{ and } p\in P_L.$$

Here τ_{cd} and $\mathcal{G}_{cd}$ are non-symmetric binary interaction parameters. All other variables are like in the general model.

For a special case, the mixture of n-Butyl-Acetate and Water, the problem data is (see [56, 6.3.2])

$$\begin{aligned}P &= 1\,\text{bar}\\ T &= 298\,K\\ n &= (0.5, 0.5)^T\\ \alpha &= \begin{pmatrix} 0 & 0.391965\\ 0.391965 & 0\end{pmatrix}\\ \tau &= \begin{pmatrix} 0 & 3.00498\\ 4.69071 & 0\end{pmatrix}\\ \Delta G_{p,f}^c &= 0.\end{aligned}$$

We need the molar balance constraint, since the mixture is non-reacting.

More data and special versions of the Gibbs free energy functions tailored for special mixtures can be found in [56, 6.3, 6.4].

P. Program

McDonald & Floudas [145] have solved some problems in this formulation with their GLOPEQ algorithm, especially the example in the previous section.

R. Report

The global minimizer of our special example is

$$N = \begin{pmatrix} 0.4993 & 0.0007 \\ 0.3441 & 0.1559 \end{pmatrix},$$

the number of moles of Water and n-Butyl-Acetate in the two phases of the chemical equilibrium state.

R. Report, S. Problem Statement

The Gibbs free energy function usually is highly non-convex, and for nonideal systems, the problem usually has multiple local (perhaps even global) solutions. Which of them is the correct one?

In addition, we have *assumed* the number of phases in the mathematical model. This assumption might be wrong. How do we check, whether the number of predicted phases is correct?

T. Theory

We have to determine the stability of a computed solution in order to check, whether it is the correct solution, and whether the number of phases has been predicted correctly.

The *tangent plane criterion* can be used to decide which one of the solutions is the true equilibrium solution. For a candidate point x the tangent plane distance function is the difference between the Gibbs energy surface for a phase, which is yet unaccounted for, and the tangent plane to the Gibbs energy surface at x. If this function is non-negative over the whole composition space, x is the true equilibrium solution.

M. Model

Let x be the vector built from the x_c, the mole fractions of the components $c \in C$ in the new phase. Then conservation of atoms induces the elemental balance equation for reacting systems

$$Ax = b, \tag{1.10}$$

and conservation of mass implies the molar balance equation for non-reacting systems

$$\|x\|_1 = 1, \tag{1.11}$$

and $0 \le x \le 1$ component wise. For a solution z of the Gibbs free energy problem, the tangent function is given by

$$F = x^T g(x, z), \tag{1.12}$$

where

$$g(x,z)_c = \Delta G_c^f + \log x + \log \hat{\gamma}_c - \mu_c(z). \tag{1.13}$$

If the global minimum of this function with respect to the constraints above is non-negative, the solution z is stable, otherwise it is unstable.

S. Problem Statement

In fact, the separation has to be considerable, and meta-stability is a problem, as well. Therefore, we would like to require $F(x) > 0$. If we reverse the formulation, we can ask, whether there is a point x which satisfies the conservation constraints and $F(x) \leq 0$. If such a point exists, the solution z of the Gibbs free energy problem is unstable, and if it can be proved that no such point exists, z is stable.

N. Numerical Analysis

In this new formulation, the stability problem becomes a *constraint satisfaction problem*, i.e. just constraints and no objective function, see Chapter 3.

P. Program

We might want to use a state-of-the-art constraint satisfaction solver or a global optimization algorithm for solving the problem. In any case, it must be a complete method. `BARON`, α`BB`, `GLOPEQ` [145] might be a choice.

1.4.2 Protein Folding

The following model is rather descriptive. This time I will give just a summary, not the whole modeling process, since it is far too complex for a short section. For a complete summary see, e.g., the survey by Neumaier [162].

Proteins are the building blocks of life as we know it. Proteins are the biological machines within the human body, and within the bodies of all lifeforms on Earth. There are billions of proteins around, performing all sorts of different functions. They are building the skeletal structure of the cells, digest our food, they are responsible for our movement. The hormones are proteins, as are pheromones, and many pharmaceutical products are proteins, as well.

Surprisingly, nature uses a sort of Lego® bricks for producing that variety. All proteins are constructed as a, probably long, string of amino acids, only 20 different amino acids. The sequence of amino acids, the *primary structure*, can be determined with comparably little effort. This structure is coded in the genetic information, in the DNA, and the code is well-known. Three base-pairs determine one amino acid, and via mRNA (messenger RNA) the information is carried into the ribosomes, the protein factories inside cells, where the string of amino acids is formed with the help of rRNA (ribosomal RNA) and tRNA (transfer RNA). So one could think that knowing the complete genetic information of a cell suffices to determine all proteins and their biological function, hence to unveil the secrets of life itself.

However, before proteins can carry out their biological and biochemical function, they undergo a dynamical process, they *fold*. Like with Lego® bricks, the beautiful structures are not created by mere successive arrangement. The three dimensional structure is, what appeals to the eye. Proteins are one of nature's masterpieces, because they remarkably assemble themselves into a 3D structure, the *tertiary structure*. It is this structure, which causes the proteins to perform their function. The process of protein folding, fundamental to all of biology, its dynamics and the final results, still are unknown. There are many mathematical models (see, e.g., the introductory paper by Richards [186] in Scientific American, and from a mathematical point of view the aforementioned survey by Neumaier [162] and the article by Kostrowicki & Scheraga [124]) around, but most of them are not completely convincing, and the more convincing ones are extremely difficult to solve.

A protein can be unfolded and refolds itself properly (this was shown in the 1960s by Christian Anfinsen, a work for which received the 1972 Nobel prize for chemistry). Well, usually this works. Sometimes, proteins can change the folding, and sometimes they return to the "wrong" shape. This is known since the stone age, since *cooking* makes use of just that fact. When an egg is being boiled, the heat causes the protein in the white to unfold and refold in a different structure, in which it is unable to perform any meaningful biological function, well except for tasting well.

Determining the tertiary structure of a protein is essential in many ways.

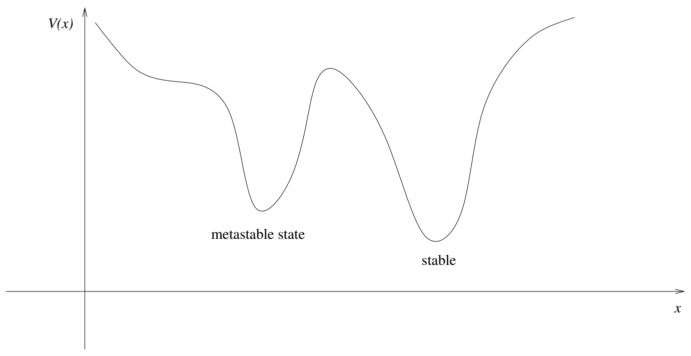


Figure 1.5. *Stable and metastable state*

- There are a number of well known diseases caused by misfolding of proteins, e.g., Alzheimer's disease, BSE (mad cow disease), Creutzfeld-Jakob disease (CJD), Amyotrophic Lateral Sclerosis (ALS), and Parkinson's disease. This is probably related to the existence of *metastable* states, i.e., substantially different configurations of an amino acid sequence having not very different potential energy but are separated by high energy barriers (see Figure 1.5).
- Pharmaceutical products made from proteins could be made better fitting, if their chemical structure would be predictable.

- Viruses and bacteria have structural hulls made from proteins. The immune systems tries to identify typical structures of these surface proteins and counteracts by manufacturing antibodies specifically targeting parts of these proteins. Diseases difficult for the immune system (HIV, Herpes,. . .) and autoimmune diseases like MS (multiple sclerosis) have a strong connection to proteins and their tertiary structure.

Because of its importance, many research teams worldwide work on a solution of the protein folding problem: Given a sequence of amino acids, determine the tertiary structure of the resulting protein.

Most mathematical models describe the potential energy of possible 3D structures of a protein, depending either on the coordinates of all atoms, or of the C_α-atoms (the base atoms of the amino acids) by a function, the potential energy function V.

Thermal movement causes the protein molecules to change shape, and hence change their potential energy. This happens until the protein has reached an energy level low enough that its structure remains sufficiently stable. This state has to be at least a local minimum of the potential energy function. The biochemical processes in nature heavily rely on the predictability and the stability of the 3D structure of a protein. So, it is very likely that the proteins prevalent in biological tissue have exceptional folding states. There are only few possibilities of being stable despite the molecular movements cause by thermal influence. The protein structure corresponds to the *global* optimum of the potential energy function, or at least it is at a local optimum with very steep potential barriers separating it from the global one. There are indeed some indications that the second case is indeed possible in nature. E.g., insulin, the hormone responsible for managing the level of glucose in the blood, is produced in a remarkable way. A string of amino acids is formed which is considerably longer than the one needed for insulin, and after this longer protein has folded, the ends of the strings are chopped off resulting in a properly folded insulin molecule.

Mathematically, we are left with a global optimization problem

$$\begin{aligned} &\min V(x) \\ &\text{s.t. } C(x) \in \mathbf{C} \end{aligned}$$

with objective function V, the potential energy function, and a set of *configuration constraints*. These contain maximum bonding lengths between the atoms in the structure, minimal atom distances, restrictions on the bonding angles (e.g. Ramachandran diagrams), surface terms, polarity restrictions, and the like.

The exact form of the optimization problem is still disputed and subject of ongoing research. A typical potential energy function is, e.g., the CHARMM potential

$$\begin{aligned} V(x) = &\sum_{\text{bonds}} c_l(b - b_0) \quad (b \text{ a bond length}) \\ &+ \sum_{\text{bond angles}} c_a(\theta - \theta_0) \quad (\theta \text{ a bond angle}) \\ &+ \sum_{\substack{\text{improper} \\ \text{torsion angle}}} c_i(\tau - \tau_0) \quad (\tau \text{ an improper torsion angle}) \\ &+ \sum_{\text{dihedral angles}} \operatorname{trig}(\omega) \quad (\omega \text{ a dihedral angle}) \\ &+ \sum_{\text{charged pairs}} \frac{Q_i Q_j}{D r_{ij}} \quad (r_{ij} \text{ is the Euclidean distance of the charged atoms}) \\ &+ \sum_{\text{unbonded pairs}} c_w \varphi\left(\frac{R_i + R_j}{r_{ij}}\right) \quad \begin{matrix} (R_i \text{ atom radius, } \varphi \text{ the} \\ \text{Lennard-Jones potential).} \end{matrix} \end{aligned}$$

These optimization problems can be *very* high dimensional, for biologically relevant proteins with the simplest models the number of variables can reach several hundreds. Finding the global optimum is essential, and heuristic approaches have been developed tailored just for this problem. However, the number of local minima in the potential energy function can be high, and the probability that incomplete methods get stuck in one of these local optima is considerable. Complete methods, however, are not yet fit for problems of this size and complexity.

In addition, solving just the global optimization problem is not enough. Finding *almost global* local optima is important, because they might help identifying metastable states or biochemically relevant configurations, which are not global minimizers of the potential energy function.

Chapter 2

Local Optimization

Optimization is, as mentioned before, the task of finding the best possible choices with respect to a given target, not violating a number of restrictions. Historically, optimization problems first came from *physical* models. Right after the invention of differential calculus the first optimality criteria (vanishing of the gradient) were found by Sir Isaac Newton [172, 173] and Gottfried Wilhelm Leibniz [137], and already at the turn of the 18th to the 19th century Joseph Louis Lagrange [130, 131] gave a criterion for optimality in the presence of equality constraints, the famous Lagrange multiplier rule.

Before the 1940s, however, only few optimization applications had been solved, because little theory on the numerical solution of optimization problems was available. The Newton method in more than one variable and the steepest descent method were known, but it required legions of "computers" (most of them lowly-paid women) to solve applications of reasonable size.

Things changed when in the 1940s two things happened which changed the field completely. ENIAC, switched on in 1945, heralded the computer era, and Dantzig [41] invented in 1947 the simplex algorithm. During the next decade computers and algorithms improved so much that solving linear optimization problems (linear programming problems) more and more became a routine task.

Non-linear problems in more than ten dimensions were out of reach, though. In 1959, Davidon [43] developed his variable metric method, which boosted the size of locally solvable problems from ten to hundreds of variables.

Nowadays, solving local optimization problems in tens of thousands of variables is possible, not only due to computers of much higher computing power but also because of the vast progression made in mathematical theory and algorithm design.

Global optimization is still in its beginnings, although there are a number of solvers around (e.g., BARON [193] or αBB [3]), which can routinely find the global optimum for most problems up to ten variables, solve a lot of problems in up to 100 variables and selected problems with more than 1000 variables.

To define the optimization problem in a mathematical way, let $X = \mathbb{R}^n \times \mathbb{Z}^m$, and take a continuous function $f : D \subseteq X \to \mathbb{R}$ and a set $C \subseteq D$. We consider the **minimization problem**

$$\begin{aligned} &\min f(x) \\ &\text{s.t.}\ \ x \in C. \end{aligned} \tag{2.1}$$

The function f is called the **objective function**, and the set C the **feasible domain**. The elements of C are called **feasible point**s.

A **local minimizer** of (2.1) is a feasible point $\hat{x} \in C$ such that for some neighborhood U of $\hat{x}$ in C we have

$$f(x) \geq f(\hat{x}) \quad \text{for all } x \in U. \tag{2.2}$$

If this inequality is valid for all $x \in C$, then $\hat{x}$ is called a **global minimizer**. The function value $f(\hat{x})$ at a global (local) minimizer is called a **global (local) minimum**.

If the first line of problem (2.1) reads instead

$$\max f(x),$$

we are considering a **maximization problem**. Solving it is equivalent to finding the solution to $\min -f(x)$. A global (local) minimizer of $-f$ is called a **global (local) maximizer**, and its function value a **global (local) maximum**. The term **global (local) optimizer** shall henceforth refer to either minimizers or maximizers, and analogously for the terms **global (local) optimum** and **optimization problem**. Sometimes we will say **(global) extremum** for (global) optimum.

Since minimization and maximization are the same up to a change of sign, we will restrict all our considerations in the following to *minimization* problems.

Several types of optimization problems can be distinguished. If $n = 0$, i.e. $X \subseteq \mathbb{Z}^m$, the problem is called a **discrete optimization problem**, and if $m = 0$, i.e. $X \subseteq \mathbb{R}^n$, it is called a **continuous optimization problem**. If both $m \neq 0$ and $n \neq 0$, we are considering a **mixed integer optimization problem**. Often, the term **program**[3] is used instead of optimization problem and **programming** is another expression for solving an optimization problem.

In this book we will restrict ourselves to smooth continuous problems, so from now on we take $X = \mathbb{R}^n$.

The feasible set usually is defined by imposing **constraint**s, restricting the domain D of f. A constraint of the form

$$x \leq u \quad \text{or} \quad x \geq l$$

is called a **bound constraint**. More complicated constraints are

$$g(x) = a, \quad h(x) \leq b \quad \text{and} \quad k(x) \in [c, d]$$

equality and **(twosided) inequality constraint**s, where g, h, and k are continuous functions. We can collect all constraint functions into a single vector $F : D' \subseteq \mathbb{R}^n \to \mathbb{R}^k$. Furthermore, we can collect all right hand sides into a vector of intervals, a box $\boldsymbol{F} \subseteq \mathbb{R}^k$, including infinitely wide intervals for constraints which have only one bound. All bounds from the bound constraints are collected in $\boldsymbol{x} \subseteq D \cap D'$, another box, again possibly unbounded. We end up with the optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) \in \boldsymbol{F} \\ &\qquad x \in \boldsymbol{x}. \end{aligned} \tag{O}$$

If there are no constraints, we call the problem **unconstrained**. If the function F is absent, it is a **bound constrained** problem.

If F is linear, we have a **linearly constrained** problem, and if in addition f is linear, we speak of a **linear program**. If f is quadratic, it is called a **quadratic program**.

In all other cases the optimization problem is called **nonlinear program** or **nonlinear optimization problem**, although further fine grained distinctions exist.

If the objective function f is constant (equivalently altogether missing) the problem is called a **feasibility problem** or **constraint satisfaction problem**.

A continuous optimization problem is called **smooth**, if f and F are continuously differentiable, otherwise **nonsmooth**.

All optimization problems considered in this book will be smooth unless otherwise explicitly stated.

Before we dive deeper into the mathematics of the subject, we note that there is one immediate result.

[3]This term is a remnant of the times when optimization was primarily used in military logistics. It means a schedule, a plan, or just a set of actions, and programming is the process of designing such a program.

Theorem 2.1 (Existence of solutions).

(i) *Let C be nonempty and compact. If f is continuous in C, the optimization problem* (2.1) *has a global solution.*

(ii) *If there is a feasible point $x_0 \in C$ and the level set*

$$C_0 := \{x \in C \mid f(x) \le f(x_0)\}$$

is compact, the optimization problem (2.1) *has a global optimizer.*

Proof.

(i) This holds, since continuous functions attain their minimum in every nonempty compact set.

(ii) This is a consequence of (i), because the level set C_0 is nonempty ($x_0 \in C_0$) and compact.

$\square$

The remainder of the chapter is with few exceptions devoted to *local* optimization only, and the presentation benefitted a lot from the unpublished lecture notes of Neumaier [163].

Being able to solve the optimization locally in an efficient way is a prerequisite for any efficient global optimization algorithm. This has two reasons: The first one, which is obvious, is that every global optimizer of problem (O) is a local optimizer. The other one, which is less obvious, is that algorithms for global optimization tend to be more efficient if they can derive additional redundant constraints which help to reduce the search space by constraint propagation (see Section 3.5) or improve relaxations (see Sections 3.6 and 3.7). A very powerful constraint in this respect is

$$f(x) \le f_*, \tag{2.3}$$

where $f_* = f(x_*)$, and x_* is the best known feasible point.

For many global optimization problems, especially those involving equality constraints, it is virtually impossible to find feasible points by trial and error, so stronger methods are needed. Local optimization of a measure of infeasibility is very well suited for solving that problem. However, one problem has to be addressed. As we will see later, local optimization algorithms need a reasonable point to start from. Finding a good *starting point* sometimes is a demanding challenge. Complete search methods (see Section 3.3) and heuristic global optimization algorithms (see Section 3.2), in return, can help there. Thus, local and global optimization complement each other well. Nevertheless, local optimization is the easier task so it will be the first subject we put our focus on.

Good introductions to local optimization can be found, e.g., in Fletcher [52], Bazaraa et al. [11], and Nocedal & Wrigth [174].

2.1 Convex Analysis and Duality

Optimization is a mathematically rich field. Its results depend as much on analysis as on geometric ideas. Especially powerful are the techniques of convex analysis, whose foundations will be presented in this section. Readers who want to dive deeper into the field, can start with the excellent reference by ROCKAFELLAR & WETS [188] or its predecessor by ROCKAFELLAR [187]. Introductions which focus on the connection between optimization and convexity are, e.g., BAZARAA ET. AL [11], HIRIART-URRUTY & LEMARÉCHAL [86], BORWEIN & LEWIS [27], or STOER & WITZGAL [218].

We start by a short summary of the basic definitions.

Let $x, y \in \mathbb{R}^n$ be two points. The **line segment** connecting x and y is defined by

$$\overline{xy} := \{\lambda y + (1-\lambda)x \mid \lambda \in [0,1]\}.$$

Given a finite number of points $x_0, \dots, x_n$, the expression

$$\sum_{k=0}^{n} \lambda_j x_k, \quad \text{with} \sum_{k=0}^{n} \lambda_k = 1 \text{ and } \lambda_k \geq 0 \text{ for } k = 0, \dots, n$$

is called a **convex combination** of the points x_k.

A set $C \subseteq \mathbb{R}^n$ is **convex** if for all $x, y \in C$ the line segment $\overline{xy} \subseteq C$. The following lemma is a direct consequence of the definition.

Lemma 2.2. *If C_1 and C_2 are convex sets in $\mathbb{R}^n$ then $C_1 \cap C_2$, λC, and $C_1 + C_2$ are convex, as well.*

Let $S \subseteq \mathbb{R}^n$ be arbitrary. The **convex hull** $\mathrm{ch}(S)$ of S is the intersection of all convex sets $C \supseteq S$. Another simple proof shows

Lemma 2.3. *The convex hull $\mathrm{ch}(S)$ of S is the smallest convex set containing S, and it can be explicitly described as the set of all convex combinations of points in S.*

If $S \subseteq \mathbb{R}^n$ is a finite set the convex hull $\mathrm{ch}(S)$ is called a **polytope** (for the beginnings in this field see MINKOWSKI [148]). If the elements of S are affinely independent, we call $\mathrm{ch}(S)$ is a **simplex**.

The following result is due to CARATHÉODORY [30].

Theorem 2.4 (Carathéodory).
For an arbitrary subset S of Rz^n every point in the convex hull $\mathrm{ch}(S)$ is a convex combination of at most $n+1$ points of S.

Proof. Take $x \in \mathrm{ch}(S)$. We know from Lemma 2.3 that x can be expressed as a convex combination of points in S. Let N be the minimal number of points necessary. If $N \leq n$, we are finished. So $N > n$, and let

$$x = \sum_{j=0}^{N} \lambda_j x_j$$

be the linear combination with non-negative λ_j, and $\sum_{j=0}^{N} \lambda_j = 1$.

We define the set $K := \{x_j - x_0 \mid j = 1, \dots, N\}$. Since $K \subseteq \mathbb{R}^n$ the set is linearly dependent, and so there are coefficients μ_j not all zero, so

$$\sum_{j=1}^{N} \mu_j (x_j - x_0) = 0, \qquad \text{and w.l.o.g. } \mu_1 > 0,$$

$$\sum_{j=0}^{N} \mu_j x_j = 0, \qquad \text{with } \mu_0 = -\sum_{j=1}^{N} -\mu_j.$$

This implies for all $\nu \in \mathbb{R}$ that

$$x = \sum_{j=0}^{N} (\lambda_j - \nu \mu_j) x_j.$$

We set

$$\nu := \min \left\{ \frac{\lambda_j}{\mu_j} \,\middle|\, \mu_j > 0,\ j = 0, \dots, N \right\},$$

and w.l.o.g. we have $\nu = \frac{\lambda_N}{\mu_N} > 0$. Furthermore, for all $j = 0, \dots, N-1$ we have $\alpha_j := \lambda_j - \nu \mu_j \geq 0$. If $\mu_j \leq 0$, this is obvious, and if $\mu_j > 0$ we have $\frac{\lambda_j}{\mu_j} \geq \nu$ by construction of ν. Since $\alpha_N = 0$, we have

$$x = \sum_{j=0}^{N-1} \alpha_j x,$$

and

$$\sum_{j=0}^{N-1} \alpha_j = \sum_{j=0}^{N} \lambda_j - \nu \sum_{j=0}^{N} \mu_j = 1 - \nu (\sum_{j=1}^{N} \mu_j + \mu_0) = 1 - 0\nu = 1.$$

So we have found a convex combination with fewer points, hence N was not minimal.
$\square$

Let S be an arbitrary subset of $\mathbb{R}^n$. We denote by $\overline{S}$ the topological closure of S (for an introduction to topology see, e.g. Willard [226]).

Proposition 2.5. *The closure $\overline{C}$ of any convex set $C \subseteq \mathbb{R}^n$ is convex.*

Proof. This follows from the continuity of addition and multiplication with scalars in $\mathbb{R}^n$: $\lambda\overline{C} + \mu\overline{C} \subseteq \overline{\lambda C + \mu C}$, and thus $\lambda\overline{C} + \mu\overline{C} \subseteq \overline{C}$ if $\lambda C + \mu C \subseteq C$. For $\lambda \geq 0$ and $\mu \geq 0$ with $\lambda + \mu = 1$ this is exactly what is required. $\square$

For every non-empty set S we define the **closed convex hull** $\operatorname{cch}(S) := \overline{\operatorname{ch}(S)}$ of S. It is the intersection of all closed convex sets containing S.

Corollary 2.6. *If $S \subseteq \mathbb{R}^n$ is bounded then $\operatorname{cch}(S)$ is compact.*

Proof. Take any norm $\|.\|$ on $\mathbb{R}^n$. Let x be any point in $\operatorname{ch}(S)$. From Theorem 2.4 we know that $x = \sum_{j=0}^n \lambda_j x_j$, a convex combination. Since S is bounded, we know $\|x_k\| \leq M$ for all x_k, hence $\|x\| \leq \sum_{j=0}^n \lambda_j \|x_j\| \leq M \sum_{j=0}^n \lambda_j = M$. So $\operatorname{ch}(S)$ is bounded, and since the closure of a bounded set is bounded, we know that $\operatorname{cch}(S)$ is bounded. By the theorem of Bolzano and Weierstraß $\operatorname{cch}(S)$ is compact. $\square$

The following theorem, essentially due to Minkowski [149], shows that for a convex closed set C and point $x \notin C$ we can find a **separating hyperplane** so that the set is on one side and the point is on the other.

Theorem 2.7 (Separation Theorem).
For a nonempty closed and convex set $C \in \mathbb{R}^n$ and a point $x \notin C$ we can find a point $c \in C$ and a vector $p \in \mathbb{R}^n$ with

$$p^T x < p^T c \leq p^T z \quad \text{for all } z \in C. \tag{2.4}$$

Proof. We consider the optimization problem

$$\begin{aligned} &\min \|z - x\|_2 \\ &\text{s.t. } z \in C. \end{aligned} \tag{2.5}$$

By assumption, there exists a point y feasible for this problem. The level set $C_0 = \{z \in C \mid \|z - x\|_2 \leq \|y - z\|_2\}$ is compact, since it is an intersection of C with a closed norm ball, hence closed and bounded. By Theorem 2.1 the problem admits a solution $c \in C$. We have

$$p := c - z \neq 0, \quad \text{since } z \notin C.$$

For $\lambda \in (0, 1)$ and $z \in C$ we set $z^\lambda := c + \lambda(z - c) \in C$. By construction, we get

$$\begin{aligned} 0 &\leq \|z^\lambda - x\|_2^2 - \|c - x\|_2^2 = \|p + \lambda(z - c)\|_2^2 - \|p\|_2^2 \\ &= 2\lambda p^T (z - c) + \lambda^2 \|z - c\|^2. \end{aligned}$$

Divide by 2λ and take $\lambda \to 0$. This implies $p^T(z - c) \geq 0$, and thus

$$p^T x \geq p^T c = p^T x + p^T p > p^T x.$$

$\square$

We say that for a convex set C a function $f : C \to \mathbb{R}$ is **convex** in C if

$$f(\lambda y + (1 - \lambda)x) \leq \lambda f(y) + (1 - \lambda) f(x) \quad \text{for } x, y \in C \text{ and } \lambda \in [0, 1]. \tag{2.6}$$

It is called **strictly convex** in C if (2.6) holds and equality implies $x = y$ or $\lambda \in \{0, 1\}$. We say that a function $F : C \to \mathbb{R}^n$ is (strictly) convex, if all the component functions $F_k : C \to \mathbb{R}$ are (strictly) convex.

Lemma 2.8. *Any (affine) linear function is convex but not strictly convex.*

Proof. Trivial. $\square$

Lemma 2.9. *A C^1 function f on a convex set C is convex in C if and only if*

$$f(z) \geq f(x) + \nabla f(x)(z - x) \tag{2.7}$$

for all $z \in C$. Furthermore, f is strictly convex if and only if, in addition, equality in (2.7) holds only for $z = x$.

Proof. If f is convex and $x, z \in C$. Then the definition (2.6) implies for all $\lambda \in (0, 1]$

$$f(z) - f(x) \geq \frac{f\big(x + \lambda(z - x)\big)}{\lambda}.$$

Since f is differentiable, the result follows for $\lambda \to 0$. Now assume that f is strictly convex and that equality holds in (2.7) for some $x \neq z$, i.e., $\nabla f(x)(z - x) = f(z) - f(x)$. This together with the definition of strict convexity implies that

$$f((1-\lambda)x+\lambda z) < (1-\lambda)f(x)+\lambda f(x)+\lambda\nabla f(x)(z-x) = f(x)+\lambda\nabla f(x)(z-x) \tag{2.8}$$

for $0 < \lambda < 1$. Since f is convex and $(1 - \lambda)x + \lambda z \in C$, we know

$$f((1 - \lambda)x + \lambda z) \geq (1 - \lambda)f(x) + \lambda\nabla f(x)(z - x),$$

which contradicts (2.8). So equality in (2.7) holds only for $z = x$.

Now suppose that the inequality holds. For $x, y \in C$ and $\lambda \in [0, 1]$ set $z := \lambda y + (1 - \lambda)x$. We get

$$\begin{aligned}\lambda f(y) + (1 - \lambda)f(x) &= \lambda\big(f(y) - f(z)\big) + (1 - \lambda)\big(f(x) - f(z)\big) + f(z)\\ &\geq \lambda\nabla f(z)(y - z) + (1 - \lambda)\nabla f(z)(x - z) + f(z)\\ &= \nabla f(z)\big(\lambda y + (1 - \lambda)x - z\big) + f(z) = f(z),\end{aligned}$$

so f is convex. If the inequality is strict whenever $y \neq x$, this implies strict inequality in the equation above, so f is strictly convex. $\square$

For C^2 functions we get the following result:

Lemma 2.10. *Let f be a C^2 function on a convex set C. We have f is convex in C if and only if $\nabla^2 f(z)$ is positive semidefinite for all $z \in \operatorname{int}(C)$.*

If furthermore, $\nabla^2 f(z)$ is positive definite for all $z \in C$, then f is strictly convex. The converse is not true, however (a counterexample is $f(x) = x^4$).

Proof. Suppose f is convex. Now let $h \in \mathbb{R}^n$ be an arbitrary vector, and take $z \in \operatorname{int}(C)$. Thus, there exists a $\lambda > 0$ with $z + \lambda h \in C$. Since f is convex, Lemma 2.9 implies

$$f(z + \mu h) - f(z) - \mu \nabla f(z) h \geq 0$$

for all $0 < \mu < \lambda$. Taylor's theorem implies that

$$f(z + \mu h) - f(z) - \mu \nabla f(z) h = \tfrac{1}{2}\mu^2 h^T \nabla^2 f(z) h + \mu^2 R^2(z, \mu h)\|h\|^2.$$

Thus,

$$\tfrac{1}{2}\mu^2 h^T \nabla^2 f(z) h + \mu^2 R^2(z, \mu h)\|h\|^2 \geq 0$$

for all $0 < \mu < \lambda$. Since $R^2(z, \mu h) \to 0$ for $\mu \to 0$, this implies

$$h^T \nabla^2 f(z) h \geq 0,$$

so the Hessian of f is positive semidefinite.

Assume, conversely, that the Hessian is positive semidefinite. By the Taylor theorem and the integral remainder term we have

$$f(y) = f(z) + \nabla f(z)(y - z) + \tfrac{1}{2}(y - z)^T \nabla^2 f\big(z + \theta(y - z)\big)(y - z), \quad 0 < \theta < 1.$$

The last term on the right is nonnegative since the Hessian is positive semidefinite, hence f is convex. If the Hessian is positive definite, the third term is positive, if $y \neq z$, and thus f is strictly convex. □

The following definition is due to MANGASARIAN [141, 142]. We will need the property later in the statement of the generalized Karush-John conditions (see Section 2.2.4).

Definition 2.11. *Let f be a function defined on an open subset $D \subseteq \mathbb{R}^n$. Let $S \subseteq D$ be any set. We say f is pseudoconvex at $x \in S$ (with respect to S) if it is differentiable at x and*

$$z \in S,\ \nabla f(x)(z - x) \geq 0 \quad \Longrightarrow \quad f(z) \geq f(x). \tag{2.9}$$

It is called ***pseudoconvex*** *on S if it is pseudoconvex with respect to S at all $x \in S$.*

Every convex function is pseudoconvex by Lemma 2.9.

The function f is called **(strictly) concave** if $-f$ is (strictly) convex. It is called **pseudoconcave** on a set S if $-f$ is pseudoconvex on S. Note that any (affine) linear function is convex and concave.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called **unimodal** in C if

$$f(z) < \max\big(f(x), f(y)\big) \quad \text{for } z \in \overline{xy},\ x, y \in C \setminus \{z\}. \tag{2.10}$$

A strictly convex function is unimodal, as a direct consequence of the definitions.

An immediate connection between convex functions and convex sets is given by the following result, which follows directly from the definitions.

Proposition 2.12. *For every convex set C and every convex function $f : C \to \mathbb{R}$, the set*

$$S := \{x \in C \mid f(x) \le 0\}$$

is convex.

An important consequence of this result is the following corollary for linear inequalities:

Proposition 2.13. *For $C \subseteq \mathbb{R}^n$ convex, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ the sets*

$$C_+ := \{x \in C \mid Ax \ge b\}, \quad C_0 := \{x \in C \mid Ax = b\}, \quad C_- := \{x \in C \mid Ax \le b\}$$

are convex.

Now we have assembled enough material for proving the first results on optimization problems:

Theorem 2.14 (Optimization on convex sets).
Consider the optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } x \in C. \end{aligned} \tag{2.11}$$

with convex C.

(i) *If f is convex in C then every local optimizer of* (2.11) *is a global optimizer, and the set of all optimizers is convex.*
(ii) *If f is unimodal in C, then* (2.11) *has at most one solution.*

Proof.

(i) Let x be a local solution of (2.11), and $y \in C$ arbitrary. Since f is convex, we have
$$f(y) - f(x) \ge \frac{f(x + h(y-x)) - f(x))}{h}$$
for all $0 < h \le 1$. Since x is a local optimizer, for h small enough $f(x + h(y - x)) \ge f(x)$, hence $f(y) - f(x) \ge 0$, and x is a global optimum.
(ii) Suppose, we have a local solution $x \in C$. Take $z \in C$ with $z \ne x$ and $\lambda > 0$ sufficiently small. Then $f(x + \lambda(z - x)) \ge f(x)$. By unimodality $f(x + \lambda(z - x)) < \max\bigl(f(x), f(z)\bigr)$, and so $f(z) > f(x)$, and z is not a global solution.

□

One of the central lemmas in optimization theory is the lemma of Farkas [49].

Lemma 2.15 (Farkas).
Let $A \in \mathbb{R}^m \times n$, and $g \in \mathbb{R}^n$. Then exactly one of the following conditions can be satisfied:

(i) $g^T p < 0$, $Ap \geq 0$ *for some* $p \in \mathbb{R}^n$,
(ii) $g = A^T q$, $q \geq 0$ *for some* $q \in \mathbb{R}^m$.

Proof. If (i) and (ii) are both true, we have

$$g^T p = (A^T q)^T p = q^T (Ap) \geq 0,$$

a contradiction.

If (ii) is false, we have $g \notin C := \{A^T q \mid q \geq 0\}$. Since $0 \in C$, we have $C \neq \emptyset$, and the Separation Theorem 2.7 shows the existence of a vector p with

$$p^T g < p^T x \quad \text{for all } x \in C.$$

Since x is an arbitrary vector of the form $x = A^T q$ with nonnegative q, we get for all $q \geq 0$

$$g^T p < q^T Ap.$$

For $q = 0$ we have $g^T p < 0$, and for $q = \varepsilon^{-1} e_i$ and $\varepsilon > 0$ the inequality implies $(Ap)_i > \varepsilon g^T p$. For $\varepsilon \to 0$ we get the required $(Ap)_i \geq 0$. Thus (i) is possible. □

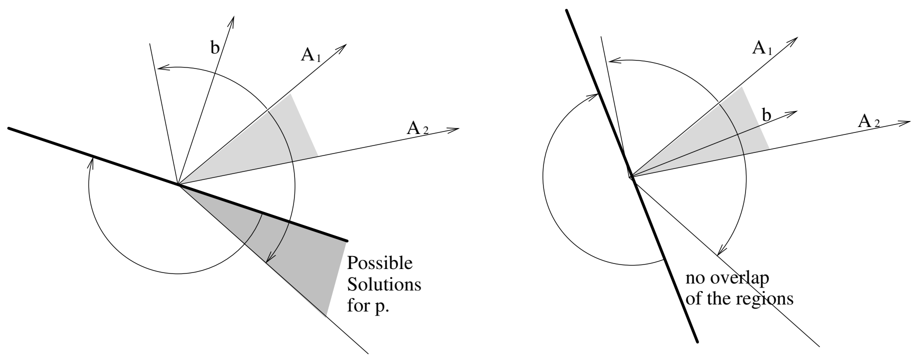


Figure 2.1. *Two incompatible properties in Farkas' lemma*

Geometrically, see Figure 2.1, property (i) requires that p is a vector, which forms acute angles ($\leq \pi/2$) with all rows of A but a strictly obtuse angle ($> \pi/2$) with the vector g. On the other hand, property (ii) demands that g is a nonnegative linear combination of the rows of A, i.e., is in the positive cone formed by them.

A useful generalization of the Lemma of Farkas is the Transposition Theorem.

Theorem 2.16 (Transposition Theorem).
Let $B \in \mathbb{R}^{m\times n}$ be any matrix, and consider a partition (I, J, K) of the set $\{1, \dots, m\}$. Then exactly one of the following conditions hold:

(i) $(Bv)_I = 0$, $(Bv)_J \geq 0$, $(Bv)_K > 0$ *for some* $v \in \mathbb{R}^n$,
(ii) $B^T w = 0$, $w_{J\cup K} \geq 0$, $w_K \neq 0$ *for some* $w \in \mathbb{R}^m$.

Proof. The theorem follows directly by applying the Lemma of Farkas 2.15 to

$$g = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad A = \begin{pmatrix} B_{I:} & 0 \\ -B_{I:} & 0 \\ B_{J:} & 0 \\ B_{K:} & e \end{pmatrix}, \quad e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}, \quad p = \begin{pmatrix} v \\ -\lambda \end{pmatrix}, \quad \text{and} \quad q = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}.$$

Then $g^T p < 0$ and $Ap \geq 0$ hold if and only if

$$\lambda > 0, \quad (Bv)_I \geq 0, \quad -(Bv)_I \geq 0, \quad (Bv)_J \geq 0, \quad (Bv)_K - \lambda e \geq 0,$$

which is clearly equivalent to (i).

Exactly if in the Lemma of Farkas (ii) holds, i.e. $g = A^T q$, $q \geq 0$, we have

$$0 = B_{I:}^T a - B_{I:}^T b + B_{J:}^T c + B_{K:}^T d, \quad 1 = e^T d, \quad a, b, c, d \geq 0.$$

Setting $w_I := a - b$, $w_J := c$, and $w_K := d$, this is equivalent to (ii), since every w with (ii) can be rescaled to satisfy $e^T w_K = 1$. $\square$

For the optimality conditions described in Section 2.2 we further need the notion of **complementarity**. We call two vectors $x, y \in \mathbb{R}^n$ **complementary**, if one, hence all, of the following equivalent conditions holds.

Lemma 2.17. *For $x, y \in \mathbb{R}^n$ the following conditions are equivalent:*

(i) $\inf(x, y) = 0$,
(ii) $x \geq 0$, $y \geq 0$, $x \star y = 0$ *(i.e., the componentwise product)*,
(iii) $x \geq 0$, $y \geq 0$, $x^T y = 0$,
(iv) $x = z_+, y = z_-$ *for some* $z \in \mathbb{R}^n$.

Linear programming has several special properties. Some of them can be carried to more general optimization problems. However, since linear functions are the only ones, which are convex and concave at the same time, linear optimization problems are very special. In the following, we collect a few results directly connected with linearly constrained optimization problems.

A **polyhedron** P is an intersection of finitely many closed half spaces, i.e., sets of the form

$$H_{p,\alpha} = \{x \in \mathbb{R}^n \mid p^T x \geq \alpha\}$$

for $p \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. If we collect all the finitely many (m) inequalities into one matrix inequality, we can define P shorter by

$$P := \{x \in \mathbb{R}^n \mid Ax \geq b\}, \quad \text{for some } A \in \mathbb{R}^{m \times n} \text{ and } b \in \mathbb{R}^m. \tag{2.12}$$

A polyhedron is closed and convex (Proposition 2.13), a good treatise of polyhedra is Schrijver [205].

A point $z \in S$ is called an **extreme point** if

$$z \in \overline{xy} \text{ for } x, y \in S \qquad \Longrightarrow \qquad z \in \{x, y\}.$$

An extreme point of a polyhedron is called a **vertex**.

An interesting connection between concave functions, convex sets, and extremal points is provided by the following theorem

Theorem 2.18. *Let C be a nonempty closed convex set. If $f : C \to \mathbb{R}$ is concave, then every extremal point of the set G of global minima of f on C is an extremal point of C.*

If f is strictly concave, then every local minimum of f on C is an extremal point of C.

Proof. Take an extremal point x of G. If x is not extremal in C, we can find $y, z \in C$ with $x \in \overline{yz}$ and $x \notin \{y, z\}$, so

$$\begin{aligned} x &= \lambda y + (1-\lambda) z & &\text{for some } \lambda \in (0,1) \qquad (2.13)\\ f(x) &\geq \lambda f(y) + (1-\lambda) f(x) & &\qquad (2.14)\\ &\geq \min(f(x), f(y)). \end{aligned}$$

Since x is a global minimizer, $f(x) = f(y)$ or $f(x) = f(z)$, and since $\lambda \in (0,1)$ this in turn implies $f(x) = f(y) = f(z)$, so $y, z \in G$, a contradiction to the extremality of x in S.

Not let f be strictly concave. Assume that x is a local optimum not extremal in C. We can again find points $y, z \in C$ both different from x satisfying (2.13). Since x is a local minimum, there are $\underline{\lambda}$ and $\overline{\lambda}$ with $\underline{\lambda} < \lambda < \overline{\lambda}$ and

$$\begin{aligned} x_1 := \underline{\lambda} y + (1-\underline{\lambda}) z, &\quad x_2 := \overline{\lambda} y + (1-\overline{\lambda}) z,\\ f(x_1) \geq f(x), &\quad f(x_2) \geq f(x). \end{aligned}$$

Since f is strictly concave, this implies $x = x_1$ or $x = x_2$ (actually one only needs unimodality of $-f$). Since $\lambda \in (\underline{\lambda}, \overline{\lambda})$ we have $x = y = z$, a contradiction. □

For linear functions this has an important consequence

Corollary 2.19. *Let C be a closed convex nonempty set, and $f : C \to \mathbb{R}$ affine linear. Then there exists a global minimizer of f, which is an extremal point of C.*

The following important theorem on extremal points is due to KREIN & MILMAN [127].

Theorem 2.20 (Krein–Milman).

(i) *If a nonempty closed convex set $C \subseteq \mathbb{R}^n$ is contained in any halfspace, then C contains an extremal point.*

(ii) *Every compact and convex set $C \subseteq \mathbb{R}^n$ is the convex hull of its extremal points.*

Proof.

See, e.g., [86] □

Theorem 2.21.

(i) *Let $A \in \mathbb{R}^{m\times n}$ be a matrix, and $b \in \mathbb{R}^m$ a vector, and $C := \{x \in \mathbb{R}^n \| Ax \geq b\}$ a polyhedron. A point $x \in C$ is extremal if and only if the matrix $A_{J:}$ with $J = \{j \mid (Ax)_j = b_j\}$ has rank n.*

(ii) *A polyhedron has at most finitely many extremal points.*

Proof. See, e.g., [86] □

2.2 Optimality Conditions

In this section we will derive a number of theorems for identifying local (sometimes global) extrema of optimization problems. First we will restrict ourselves to special classes of problems, and afterwards we will generalize the results until we end up with optimality conditions for general smooth nonlinear programming problems.

Since this section will need a lot of gradients and Hessians, we introduce abbreviations $g(x)^T = \nabla f(x)$ and $G(x) = \nabla^2 f(x)$.

2.2.1 Unconstrained Problems

The simplest optimality conditions are known since Newton and Leibniz.

Theorem 2.22 (Unconstrained optimality conditions).
Let $f : \mathbb{R}^n \to \mathbb{R}$ be a C^1 function and $\hat{x} \in \mathbb{R}^n$. If $\hat{x}$ is a local optimizer $g(\hat{x}) = 0$.

Now consider a C^2 function f. If $\hat{x}$ is a local minimum (maximum), $G(\hat{x})$ is positive (negative) semidefinite.

If $g(\hat{x}) = 0$ and $G(\hat{x})$ is positive (negative) definite, then $\hat{x}$ is a local minimum (maximum).

Proof. We consider the one-dimensional function

$$f_{(k)}(y) := f(\hat{x}_1, \dots, \hat{x}_{k-1}, y, \hat{x}_{k+1}, \dots, \hat{x}_n).$$

Since $\hat{x}$ is a local minimizer of f, we have that $\hat{x}_k$ is a local minimizer of $f_{(k)}$. Hence,

$$\frac{\partial f}{\partial x_k}(\hat{x}) = f'_{(k)}(x_k) = 0.$$

This is valid for all k, so $g(\hat{x}) = 0$.

By Taylor's theorem we know that

$$\frac{f(\hat{x}+h) - f(\hat{x})}{\|h\|^2} = \tfrac{1}{2}\big(\tfrac{h}{\|h\|}\big)^T G(\hat{x})\big(\tfrac{h}{\|h\|}\big) + R(h) \tag{2.15}$$

with $\lim_{h\to 0} R(h) = 0$.

If $G(\hat{x})$ is positive definite, and since f is C^2 we can find $\nu > 0$ with

$$\big(\tfrac{h}{\|h\|}\big)^T G(\hat{x})\big(\tfrac{h}{\|h\|}\big) \geq \nu.$$

We choose $\|h\|$ so small that $\|R(h)\| < \frac{\nu}{2}$, so by (2.15) $f(\hat{x}+h) > f(\hat{x})$ and $\hat{x}$ is a local minimizer.

If $G(\hat{x})$ is not positive semidefinite, there exists y with $\|y\| = 1$ and $y^T G(\hat{x}) y < 0$. Since for all $\lambda \neq 0$

$$\big(\tfrac{\lambda y}{\|\lambda y\|}\big)^T G(\hat{x})\big(\tfrac{\lambda y}{\|\lambda y\|}\big) = y^T G(\hat{x}) y = \alpha < 0,$$

we can choose λ so small that $|R(\lambda y)| < -\frac{\alpha}{2}$. Thus, we see $f(\hat{x}+\lambda y) - f(\hat{x}) < 0$ and $\hat{x}$ is not a local optimizer, contradiction. ☐

Note that there is a gap, seemingly small, between the necessary condition for minimality and the sufficient condition. However, this gap cannot not be closed (see $f(x) = x^3$), and we will meet it ever again during this section.

A solution of the equation $g(x) = 0$ is called a **critical point** or a **stationary point**. Not all stationary points are optima.

There is a special class of functions, for which the stationary point property is a sufficient optimality condition. A C^1 function f is called **uniformly convex** in C if there exists a positive constant α such that

$$f(y) - f(x) - g(x)^T(y - x) \geq \alpha\|y - x\|_2^2.$$

Proposition 2.23. *Let U be an open set and $f : U \to \mathbb{R}$ uniformly convex with stationary point $\hat{x} \in U$.*

(i) *Then $\hat{x}$ is the global minimum of f in U.*
(ii) *If f is a C^2 function, $\hat{x}$ is the only stationary point of f.*

Proof.

(i) Since $g(\hat{x}) = 0$ we have $f(y) - f(\hat{x}) \geq \alpha\|y - \hat{x}\| \geq 0$, for all $y \in U$, hence $\hat{x}$ is a global optimizer.
(ii) Proof from the literature.

□

Corollary 2.24 (Sufficient conditions for optimality).
If f is uniformly convex in a neighborhood U of the stationary point $\hat{x}$, then $\hat{x}$ is a local minimizer. This local minimizer is a so called ***strong (nondegenerate) minimizer****, because for all $\hat{x} \neq y \in U$ we have $f(y) > f(\hat{x})$.*

In particular, this result is true if f is C^2 and has positive definite Hessian at $\hat{x}$.

Proof. This is just a reformulation of Proposition 2.23.(i). □

Since now, we have only been talking about local minimizers in open regions, i.e., in the interior of the feasible region. If we consider as feasible area a bounded subset of $\mathbb{R}^n$ the optimality conditions have to adapt to that situation. The reason is that then local and global optimizers can also lie on the border of the feasible area, and there the optimality conditions of Theorem 2.22 need not be valid anymore. See, e.g., Figure 2.2 for a simple one-dimensional example.

We start analyzing the situation with an abstract optimality condition valid for problems with a convex feasible region.

Theorem 2.25. *Consider problem* (2.1) *with a convex set C and a C^1 function f. If $\hat{x} \in C$ is a solution of* (2.1), *we have*

$$g(\hat{x})^T(z - \hat{x}) \geq 0 \quad \textit{for all } z \in C. \tag{2.16}$$

If f is in addition convex in C, then $\hat{x}$ is a solution of (2.1) *iff* (2.16) *holds.*

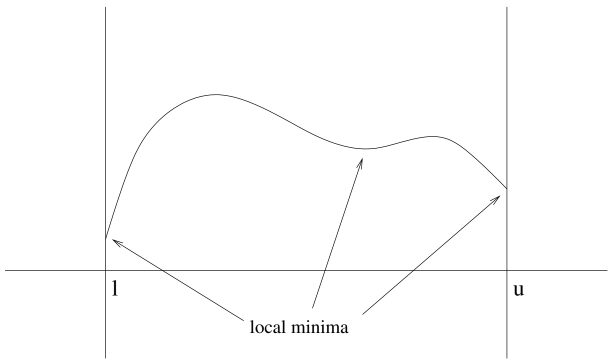


Figure 2.2. *Local minima on the border of $C = [a, b]$*

Proof. For $z \in C$ we have $\overline{\hat{x}z} \in C$, because C is convex. Since $\hat{x}$ is a local minimum, for $h > 0$ small enough we have

$$0 \le f\big(\hat{x} + h(z - \hat{x})\big) - f(\hat{x}).$$

Division by h and taking the limit $h \to 0$ shows (2.16).

If f is convex and (2.16) is satisfied, we can use Lemma 2.9 and find for all $z \in C$

$$f(z) - f(\hat{x}) \ge g(\hat{x})^T(z - \hat{x}) \ge 0.$$

Hence, $\hat{x}$ is a global minimizer. ☐

Now lets take a closer look at Figure 2.2. It provides a hint what a useful optimality condition might be. If $f : [l, u] \subseteq \mathbb{R} \to \mathbb{R}$ has a local optimum at l, the gradient $f'(l) \ge 0$, and at the other end u we have $f'(u) \le 0$. This can be *almost* reversed. If $f'(l) > 0$ (< 0) then f has at l (u) a local minimum (maximum).

Theorem 2.26 (Optimality conditions for bound constrained problems). *Consider the bound constrained optimization problem*

$$\begin{aligned} &\min f(x)\\ &\text{s.t. } x \in \boldsymbol{x}, \end{aligned} \tag{OB}$$

and take $\hat{x} \in \boldsymbol{x}$.

(i) *If $\hat{x}$ is a local minimizer the **first order optimality conditions** are satisfied:*

$$\begin{aligned} g_i(\hat{x}) &\ge 0 \quad \textit{if } \underline{x}_i = \hat{x}_i < \overline{x}_i,\\ g_i(\hat{x}) &= 0 \quad \textit{if } \underline{x}_i < \hat{x}_i < \overline{x}_i,\\ g_i(\hat{x}) &\le 0 \quad \textit{if } \underline{x}_i < \hat{x}_i = \overline{x}_i. \end{aligned} \tag{2.17}$$

(ii) *For a local minimizer $\hat{x}$ and a C^2 function f the matrix $G(\hat{x})_{J_iJ_i}$ is positive semidefinite, where*

$$J_i := \{j \mid x_j \in \operatorname{int} \boldsymbol{x}\}.$$

(iii) *If f is C^2 and the first order conditions* (2.17) *are satisfied, and if in addition the matrix $G(\hat{x})_{J_0J_0}$ with $J_0 = \{k \mid g_k(\hat{x}) = 0\}$ is positive definite, then $\hat{x}$ is a strict local minimizer of* (OB).

Constraints with index $j \in J_i$ are called **inactive** and if $j \notin J_i$ the corresponding constraint is called **active**. Note in addition that (2.17) can be written shorter as **complementarity condition**

$$\inf(g(x), x - \underline{x}) = \inf(-g(x), x - \overline{x}) = 0. \tag{2.18}$$

Proof. We will prove a much more general result in Section 2.2.4, and since we do not use the results of this theorem there, the details are left to the reader. □

Now it is time to move on to more general problems. Lets first recall another famous result of calculus, due to LAGRANGE [130, 131].

Theorem 2.27 (Lagrange multiplier rule).
Let $U \subseteq \mathbb{R}^n$ be open, and let the functions $f : U \to \mathbb{R}$ and $F : U \to \mathbb{R}^m$ be continuously differentiable. Further, let $\hat{x}$ be a local minimizer of the optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) = 0. \end{aligned} \tag{OE}$$

If $F'(\hat{x})$ has rank m (which implies $m \le n$) then there is a vector $\hat{y} \in \mathbb{R}^m$ with

$$g(\hat{x}) + F'(\hat{x})^T \hat{y} = 0. \tag{2.19}$$

The numbers y are called the **Lagrange multipliers** corresponding to the optimization problem (OE). The property that $F'(\hat{x})$ has rank m is a restriction on the structure of the constraints, the simplest version of a **constraint qualification**.

Proof. The result is an immediate consequence of the inverse function theorem. Since $\hat{x}$ is a local solution of (OE), we have $\hat{x} \in \{x \in U \mid F(x) = 0\}$, and because F is C^1 and the rank of $F'(\hat{x})$ is maximal, we can partition the variables x in two subsets $x = (s, t)$ such that in a small neighborhood of $\hat{x}$ we can express the s in terms of the t by the implicit function theorem: $s = h(t)$ and $F(h(t), t) = 0$.

Now we consider $\varphi(t) := f(h(t), t)$ and differentiate

$$\nabla\varphi(\hat{t}) = \frac{\partial f(\hat{x})}{\partial s}\nabla h(\hat{t}) + \frac{\partial f(\hat{x})}{\partial t} = 0, \tag{2.20}$$

where the last equation is true, since local optimality of $\hat{x}$ for f with respect to $F(x) = 0$ implies local optimality of $\hat{t}$ for φ.

At the same time we have

$$\frac{\partial F(\hat{x})}{\partial s}\nabla h(\hat{t}) + \frac{\partial F(\hat{x})}{\partial t} = 0,$$

since $F(h(t),t) = 0$ for all t. Since $\frac{\partial F(\hat{x})}{\partial s}$ is invertible by construction, we can compute

$$\nabla h(\hat{t}) = -\left(\frac{\partial F(\hat{x})}{\partial s}\right)^{-1} \frac{\partial F(\hat{x})}{\partial t}. \tag{2.21}$$

If we insert equation (2.21) into (2.20) we get

$$-\frac{\partial f(\hat{x})}{\partial s}\left(\frac{\partial F(\hat{x})}{\partial s}\right)^{-1} \frac{\partial F(\hat{x})}{\partial t} + \frac{\partial f(\hat{x})}{\partial t} = 0. \tag{2.22}$$

Since the product of the first two factors is a vector of dimension m, we can set

$$\hat{y}^T := -\frac{\partial f(\hat{x})}{\partial s}\left(\frac{\partial F(\hat{x})}{\partial s}\right)^{-1}. \tag{2.23}$$

That implies by (2.22) and a transformation of (2.23)

$$\begin{aligned}\frac{\partial F(\hat{x})}{\partial s}^T \hat{y} + \frac{\partial f(\hat{x})}{\partial s} &= 0\\ \frac{\partial F(\hat{x})}{\partial t}^T \hat{y} + \frac{\partial f(\hat{x})}{\partial t} &= 0,\end{aligned}$$

which together yield equation (2.19). □

Note that in case the constraint qualification is violated, there is a non-trivial linear combination of the rows of $F'(\hat{x})$, which vanishes, i.e. there is $y \neq 0$ with $F'(\hat{x})^T y = 0$. We can then reformulate the multiplier rule as follows: There is a number $\kappa \geq 0$ and a vector $y \in \mathbb{R}^m$ not both of them vanishing with

$$\kappa g(\hat{x}) + F'(\hat{x})^T y = 0. \tag{2.24}$$

This general Lagrange multiplier rule is a typical result for an optimality condition without constraint qualification, and we will meet the structure again later in Theorem 2.33.

If we take a closer look at equation (2.19), we can see that $\hat{x}$ is a critical point of the function L'

$$L'(x) = L(x,\hat{y}) = f(x) + \hat{y}^T F(x)$$

for the given multipliers $\hat{y}$. However, $\frac{\partial L(\hat{x},y)}{\partial y} = F(\hat{x}) = 0$ because of the constraints. So $(\hat{x},\hat{y})$ is a critical point of the function

$$L(x,y) = f(x) + y^T F(x), \tag{2.25}$$

the **Lagrange function (Lagrangian)** for the optimization problem (OE). The vector $\frac{\partial L(x,y)}{\partial x} = g(x) + F'(x)^T y$ is called a **reduced gradient** of f at x.

Now we have two results (Theorem 2.26 and Theorem 2.27), which ask for being unified.

To gain more insight, we first have a look at the linearly constrained case.

Theorem 2.28 (First order optimality conditions for linear constraints). *If the function f is C^1 on the polyhedron $C := \{x \in \mathbb{R}^n \mid Ax \geq b\}$ with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$, we have*

(i) *If $\hat{x}$ is a solution of the linearly constrained problem*

$$\begin{aligned} &\min f(x) \\ &\text{s.t.}\ \ Ax \geq b, \end{aligned} \tag{OL}$$

we can find a vector $y \in \mathbb{R}^m$ with

$$g(x) = A^T y \tag{2.26}$$

$$\inf(y, Ax - b) = 0. \tag{2.27}$$

(ii) *If f is convex in C then any $x \in C$ for which a y exists with* (2.26) *and* (2.27) *is a global minimizer of* (OL).

Equation (2.27) is called the complementarity condition.

Proof. Follows from Theorem 2.25 and the Lemma of Farkas 2.15. □

In principle, this theorem provides a method for solving the optimization problem (OL): Find a solution to the system (2.26), (2.27), a so called **complementarity problem**. These are $n + m$ equations in $n + m$ unknowns.

2.2.2 Duality

Now we will attack the optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t.}\ \ F(x) \leq 0 \\ &\qquad x \in C \end{aligned} \tag{OI}$$

with f and F being C^1 functions and C a convex set. If $\hat{x}$ is a local minimizer of (OI) with $F(\hat{x}) = 0$ and $\hat{x} \in \operatorname{int}(C)$ the Lagrange multiplier criterion remains true. This motivates to start our investigation of problem (OI) with the Lagrangian $L(x, y) = f(x) + y^T F(x)$.

The easiest situation is when f and F are convex functions. In this situation, the feasible set $\mathcal{F} = \{x \in C \mid F(x) \leq 0\}$ is convex, and by Theorem 2.14 every local optimizer of problem (OI) is a global one. Using the Lagrangian, we can usually find a lower bound on the global minimum:

Proposition 2.29. *If for problem* (OI) *with convex f and F, there is a $x \in C$ and a $0 \leq y \in \mathbb{R}^n$ with*

$$g(x) + F'(x)^T y = 0 \tag{2.28}$$

then

$$\min\{f(z) \mid x \in \mathcal{F}\} \geq L(x, y). \tag{2.29}$$

Proof. We have for arbitrary $z \in \mathcal{F}$

$$\begin{aligned} f(z) - f(x) &\geq g(x)^T(z-x) = -y^T F'(x)(z-x) && \text{by convexity of } f \text{ and (2.28)} \\ &\geq -\big(y^T F(z) - y^T F(x)\big) && \text{by convexity of } F \\ &= y^T F(x) - y^T F(z) \geq y^T F(x) && \text{because } F(z) \leq 0. \end{aligned}$$

Thus, we find $f(z) \geq f(x) + y^T F(x) = L(x,y)$. □

This result can be viewed from a different angle. We can try to find the best lower bound on the minimum of problem (OI) by solving

$$\begin{aligned} &\max L(x,y) \\ &\text{s.t. } \ g(x) + F'(x)^T y = 0 \\ &\qquad x \in C, \quad y \geq 0. \end{aligned} \tag{OD}$$

By Proposition 2.29 is the global maximum of (OD) always smaller or equal to the global minimum of (OI).

The optimization problem (OD) is called the **dual problem** to (OI). The latter one is denoted **primal problem**. The two optima do not need to agree. If this is indeed not the case, the distance between global maximum of the dual and global minimum of the primal problem is called the **duality gap**.

An ideal situation is when the optima of primal and dual program coincide, i.e., the duality gap closes. Then, if the minimizer $\overline{x}$ of (OI) and the maximizer $(\hat{x},\hat{y})$ agree in the sense that $\hat{x} = \overline{x}$, we have $f(\hat{x}) = f(\hat{x}) + \hat{y}^T F(\hat{x})$, thus $\hat{y}^T F(\hat{x}) = 0$. Since $F(\hat{x}) \leq 0$ and $\hat{y} \geq 0$ we can write by Lemma 2.17

$$\inf(\hat{y}, -F(\hat{x})) = 0. \tag{2.30}$$

Conversely, if (2.30) holds, then maximum and minimum coincide and so the point $\hat{x}$ is the global minimizer of the primal problem. We summarize the result in the following theorem.

Theorem 2.30 (Sufficient optimality conditions for convex problems). *Let C be a convex set, and let $f : C \to \mathbb{R}$ and $F : C \to \mathbb{R}^m$ be convex functions. If there are $\hat{x} \in C$ and $\hat{y} \in \mathbb{R}^m$ satisfying the first order sufficient optimality conditions*

$$g(\hat{x}) + F'(\hat{x})^T \hat{y} = 0 \tag{2.31}$$

$$\inf(y, -F(x)) = 0, \tag{2.32}$$

then $\hat{x}$ minimizes (OI) *globally and $(\hat{x},\hat{y})$ maximizes* (OD) *globally, and the primal minimum and the dual maximum coincide.*

2.2.3 The Karush-John conditions

This section is devoted to general smooth nonlinear nonconvex optimization problems. We will derive generalizations of the first order optimality conditions proved in Theorems 2.26, 2.27, and 2.28. Unfortunately, they will either be valid only under a slight restriction of the admissible constraints, a **constraint qualification**, or they will involve an additional parameter making the optimality conditions more difficult to handle.

The situation is simplest when the constraints are concave, i.e., of the form $F(x) \geq 0$ with convex F.

Theorem 2.31 (First order optimality conditions for concave constraints).

Let $\hat{x} \in \mathbb{R}^n$ be a solution of the nonlinear program

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) \geq 0 \end{aligned} \tag{2.33}$$

where $f : C_0 \to \mathbb{R}$ and $F : C_0 \to \mathbb{R}^r$ are continuously differentiable functions defined on their domain of definition.

If F is convex then there is a vector $\hat{z} \in \mathbb{R}^r$ such that

$$g(\hat{x}) = F'(\hat{x})^T \hat{z}, \tag{2.34}$$

$$\inf(\hat{z}, F(\hat{x})) = 0. \tag{2.35}$$

Proof. This directly follows from Theorem 2.33 proved in the next section. $\square$

This is the simplest situation involving nonlinear constraints, because due to the concave structure of the feasible set descent can be achieved with linear paths, whereas in general curved paths may be needed to get descent. The following result is due to JOHN [104] and already implicitly in KARUSH [111].

Theorem 2.32 (Karush-John first order optimality conditions).
Let $\hat{x} \in \mathbb{R}^n$ be a solution of the nonlinear program

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) = 0 \\ &\qquad x \in \boldsymbol{x}, \end{aligned} \tag{2.36}$$

where $f : C_0 \to \mathbb{R}$ and $F : C_0 \to \mathbb{R}^r$ are continuously differentiable functions defined on their domain of definition, and $\boldsymbol{x} = [\underline{x}, \overline{x}]$ is a box.

There are a constant $\kappa \geq 0$ and a vector $\hat{z} \in \mathbb{R}^r$ such that

$$\hat{y} := \kappa g(\hat{x}) - F'(\hat{x})^T \hat{z} \tag{2.37}$$

satisfies the two-sided complementarity condition

$$\begin{aligned} \hat{y}_k &\geq 0 \quad \textit{if } \hat{x}_k = \underline{x}_k, \\ \hat{y}_k &\leq 0 \quad \textit{if } \hat{x}_k = \overline{x}_k, \\ \hat{y}_k &= 0 \quad \textit{otherwise}, \end{aligned} \tag{2.38}$$

and either $\kappa > 0$, or $\kappa = 0$ and $z \neq 0$.

Proof. This is an immediate consequence of Theorem 2.33 proved in the next section. □

2.2.4 The refined Karush-John necessary first order optimality conditions

We next prove a refined version of the Karush-John first order optimality conditions which reduces the number of constraints, for which a constraint qualification is needed. This version is a generalization both of the Karush-John conditions and of the first order optimality conditions for concave constraints.

In many local and global optimization algorithms (e.g., [116] or [216]) the Karush-John conditions play a central role for the solution process. However, the Karush-John conditions in their most general form do pose problems, especially because of the factor in front of the gradient term.

Therefore, most of the local solvers require a constraint qualification, like Mangasarian-Fromowitz ([216]), to be able to reduce the Karush-John conditions to the more convenient Kuhn-Tucker conditions [128].

Deterministic global optimization algorithms cannot take this course, and so they have to use the Karush-John conditions in their general form. Unfortunately, the additional constraints needed involve all multipliers and are very inconvenient for the solution process.

There are several situations for which it is well known that no constraint qualification is required (see SCHRIJVER [205, p.220], for a history), like concave, hence also for linear, problems (see Theorems 2.28 and 2.31).

In this section we derive optimality conditions for general smooth nonlinear programming problems. The first-order conditions generalize those obtained in Theorem 2.31 for concavely constrained problems, and the derived Kuhn-Tucker conditions require constraint qualifications for fewer constraints, and the constraint qualifications presented here are a little more general than those proved in MANGASARIAN [142]. The theorem itself is due to NEUMAIER & SCHICHL [170].

For general nonlinear constraints it is useful to introduce slack variables to transform them to equality form. We are doing that for all non-pseudoconcave constraints only and write the non-linear optimization problems in the form

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } C(x) \geq 0 \\ &\qquad F(x) = 0. \end{aligned} \tag{2.39}$$

The form (2.39) which separates the pseudoconcave (including the linear) and the remaining nonlinear constraints is most useful to obtain the weakest possible constraint qualifications. However, in computer implementations, a transformation to this form is not ideal, and the slack variables should not be explicitly introduced.

Theorem 2.33 (General first order optimality conditions).
Let $\hat{x} \in \mathbb{R}^n$ be a solution of the nonlinear program (2.39), where $f : U \to \mathbb{R}$, $C : U \to \mathbb{R}^m$, and $F : U \to \mathbb{R}^r$ are functions continuously differentiable on a neighborhood U of $\hat{x}$. In addition, C shall be pseudoconvex on U. Then there are vectors $\hat{y} \in \mathbb{R}^m$, $\hat{z} \in \mathbb{R}^r$ such that

$$\kappa g(\hat{x}) = C'(\hat{x})^T \hat{y} + F'(\hat{x})^T \hat{z}, \tag{2.40}$$

$$\inf(\hat{y}, C(\hat{x})) = 0, \tag{2.41}$$

$$F(\hat{x}) = 0, \tag{2.42}$$

and

$$\text{either } \kappa = 1 \quad \text{or} \quad \hat{z} \neq 0, \kappa = 0. \tag{2.43}$$

Proof. In the beginning we observe that $\hat{x}$ is a feasible point for the optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } C'(\hat{x})_{J:} x \geq C'(\hat{x})_{J:} \hat{x} \\ &\qquad F(x) = 0, \end{aligned} \tag{2.44}$$

where J is the set of all components j, for which $C(\hat{x})_j = 0$. For the indices k corresponding to the inactive set J_+, we choose $y_{J_+} = 0$ to satisfy condition (2.41). Since C is pseudoconvex, we have $C(x) \geq C(\hat{x}) + C'(\hat{x})(x - \hat{x})$. Restricted to the rows J we get $C(x)_J \geq C'(\hat{x})_{J:}(x - \hat{x})$. This fact implies that problem (2.39) is a relaxation of problem (2.44) on a neighborhood U of $\hat{x}$. Note that since C is continuous we know that $C(x)_j > 0$ for $k \in J_+$ in a neighborhood of $\hat{x}$ for all constraints with $C(\hat{x})_j > 0$. Since $\hat{x}$ is a local optimum of a relaxation of (2.44) by assumption and a feasible point of (2.44), it is a local optimum of (2.44) as well.

Together with the choice $y_{J_+} = 0$ the Karush-John conditions of problem (2.44) are again conditions (2.40)–(2.42). So we have successfully reduced the problem to the case where C is linear.

To simplify the notation we drop the hats from $\hat{x}$, etc., and set $A := C'(x)_{J:}$ and $b := C'(x)_{J:} x$.

Let x be a solution of (2.44). If $\mathrm{rk} F'(x) < r$ then $z^T F'(x) = 0$ has a solution $z \neq 0$, and we can solve (2.40)–(2.43) with $y = 0$, $\kappa = 0$. Hence we may assume that $\mathrm{rk} F'(x) = r$.

This allows us to select a set R of r column indices such that $F'(x)_{:R}$ is nonsingular.

Let B be the $(0,1)$-matrix such that Bs is the vector obtained from $s \in \mathbb{R}^n$ by discarding the entries indexed by R. Then the function $\Phi : C \to \mathbb{R}^n$ defined by

$$\Phi(z) := \begin{pmatrix} F(z) \\ Bz - Bx \end{pmatrix}$$

has at $z = x$ a nonsingular derivative

$$\Phi'(x) = \begin{pmatrix} F'(x) \\ B \end{pmatrix}.$$

Hence, by the inverse function theorem, Φ defines in a neighborhood of $0 = \Phi(x)$ a unique continuously differentiable inverse function Φ^{-1} with $\Phi^{-1}(0) = x$. Using Φ we can define a curved search path with tangent vector $p \in \mathbb{R}^n$ tangent to the nonlinear constraints satisfying $F'(x)p = 0$. Indeed, the function defined by

$$s(\alpha) := \Phi^{-1}\begin{pmatrix} 0 \\ \alpha Bp \end{pmatrix} - x$$

for sufficiently small $\alpha \geq 0$, is continuously differentiable, with

$$s(0) = \Phi^{-1}(0) - x = 0, \quad \begin{pmatrix} F(x + s(\alpha)) \\ Bs(\alpha) \end{pmatrix} = \Phi\left(\Phi^{-1}\begin{pmatrix} 0 \\ \alpha Bp \end{pmatrix}\right) = \begin{pmatrix} 0 \\ \alpha Bp \end{pmatrix},$$

hence

$$s(0) = 0, \quad F(x + s(\alpha)) = 0, \quad Bs(\alpha) = \alpha Bp. \tag{2.45}$$

Differentiation of (2.45) at $\alpha = 0$ yields

$$\begin{pmatrix} F'(x) \\ B \end{pmatrix}\dot{s}(0) = \begin{pmatrix} F'(x)\dot{s}(0) \\ B\dot{s}(0) \end{pmatrix} = \begin{pmatrix} 0 \\ Bp \end{pmatrix} = \begin{pmatrix} F'(x) \\ B \end{pmatrix}p,$$

hence $\dot{s}(0) = p$, i.e. p is indeed a tangent vector to $x + s(\alpha)$ at $\alpha = 0$.

Now we consider a direction $p \in \mathbb{R}^n$ such that

$$g^T p < 0, \quad g = g(x), \tag{2.46}$$

$$Ap > 0, \tag{2.47}$$

$$F'(x)p = 0. \tag{2.48}$$

(In contrast to the purely concave case, we need the strict inequality in (2.47) to take care of curvature terms.) Since $Ax \geq b$ and (2.47) imply $A(x + s(\alpha)) = A(x + \alpha\dot{s}(0) + o(\alpha)) = Ax + \alpha(Ap + o(1)) \geq b$ for sufficiently small $\alpha \geq 0$, (2.45) implies feasibility of the points $x + s(\alpha)$ for small $\alpha \geq 0$. Since

$$\frac{d}{d\alpha} f(x + s(\alpha))\bigg|_{\alpha=0} = g^T\dot{s}(0) = g^T p < 0,$$

f decreases strictly along $x + s(\alpha)$, α small, contradicting the assumption that x is a solution of (2.39). This contradiction shows that the condition (2.46)–(2.48) are inconsistent. Thus, the transposition theorem applies with

$$\begin{pmatrix} -g^T \\ A \\ F'(x) \end{pmatrix}, \quad \begin{pmatrix} \kappa \\ y_J \\ z \end{pmatrix} \quad \text{in place of } B,\ q$$

and shows the solvability of

$$-g\kappa + A^T y_J + F'(x)^T z = 0, \quad \kappa \geq 0, \quad y_J \geq 0, \quad \begin{pmatrix} \kappa \\ y_J \end{pmatrix} \neq 0.$$

If we add zeros for the missing entries of y, and note that x is feasible, we find (2.40)–(2.42).

Suppose first that $\kappa = 0, z = 0$, and therefore

$$A^Ty = 0, \quad y \neq 0. \tag{2.49}$$

In this case the complementarity condition (2.41) yields $0 = (Ax-b)^Ty = x^TA^Ty - b^Ty$, hence $b^Ty = 0$. Therefore any point $\tilde{x} \in U$ satisfies $(A\tilde{x} - b)^Ty = \tilde{x}^TA^Ty - b^Ty = 0$, and since $y \geq 0$, $A\tilde{x} - b \geq 0$, we see that the set

$$K := \{i \mid (A\tilde{x})_i = b_i \text{ for all } \tilde{x} \in U\}$$

contains all indices i with $y_i \neq 0$ and hence is nonempty.

Since U is nonempty, the system $A_{K:}x = b_K$ is consistent, and hence equivalent to $A_{L:}x = b_L$, where L is a maximal subset of K such that the rows of A indexed by L are linearly independent. If M denotes the set of indices complementary to K, we can describe the feasible set equivalently by the constraints

$$A_{M:}x \geq b_M, \quad \begin{pmatrix} A_{L:}x - b_L \\ F(x) \end{pmatrix} = 0.$$

In this modified description the feasible set has no equality constraints implicit in $A_{M:}x \geq b_M$. For the equivalent optimization problem with these constraints, we find as before vectors y_M and $\binom{y_L}{z}$ such that

$$\kappa g(x) = A_{M:}^T y_M + \begin{pmatrix} A_{L:} \\ F'(x) \end{pmatrix}^T \begin{pmatrix} y_L \\ z \end{pmatrix}, \tag{2.50}$$

$$\inf(y_M, A_{M:}x - b_M) = 0, \tag{2.51}$$

$$F(x) = 0, \quad A_{K:}x - b_K = 0, \tag{2.52}$$

$$\begin{pmatrix} \kappa \\ y_M \end{pmatrix} \neq 0 \tag{2.53}$$

But now we cannot have $\kappa = 0$ and $\binom{y_L}{z} = 0$ since then, as above, $A_{M:}^T y_M = 0$ and for all $i \in M$ either $y_i = 0$ or $(Ax)_i = b_i$ for all $x \in U$. Since $K \cap M = \emptyset$ the first case is the only possible, hence $\kappa \neq 0$, which is a contradiction.

Thus, $\kappa \neq 0$ or $\binom{y_L}{z} \neq 0$. Setting $y_{K\setminus L} = 0$ we get vectors y, z satisfying (2.40) and (2.41). However, is $\kappa = 0$ we now have $z \neq 0$. Otherwise, $y_L \neq 0$, and all indices i with $y_i \neq 0$ lie in K. Therefore, $y_M = 0$, and (2.50) gives $A_{L:}^T y_L = 0$. Since, by construction, the rows of $A_{L:}$ are linearly independent, this implies $y_L = 0$, contradicting (2.53).

Thus either $\kappa \neq 0$, and we can scale (κ, y, z) to force $\kappa = 1$, thus satisfying (2.43). Or $\kappa = 0$, $z \neq 0$, and (2.43) also holds. This completes the proof. $\square$

The case $\kappa = 0$ in (2.43) is impossible if the **constraint qualification**

$$C'(\hat{x})_{J:}^T y_J + F'(\hat{x})^T z = 0 \quad \Longrightarrow \quad z = 0 \tag{2.54}$$

holds. This forces $\mathrm{rk}F'(\hat{x}) = r$ (to see this put $y = 0$), and writing the left hand side of (2.54) as $y_J^T C'(\hat{x})_{J:} = -z^T F'(\hat{x})$, we see that (2.54) forbids precisely common nonzero vectors in the **row spaces** (spanned by the rows) of $C'(\hat{x})_{J:}$ and $F'(x)$. Thus we get the following important form of the optimality conditions:

Corollary 2.34. *Under the assumption of Theorem* 2.33*, if* $\mathrm{rk} F'(\hat{x}) = r$ *and if the row spaces of* $F'(\hat{x})$ *and* $C'(\hat{x})_{J:}$*, where* $J = \{i \mid C(\hat{x})_i = 0\}$*, have trivial intersection only, then there are vectors* $\hat{y} \in \mathbb{R}^m, \hat{z} \in \mathbb{R}^r$ *such that*

$$g(\hat{x}) = C'(\hat{x})^T \hat{y} + F'(\hat{x})^T \hat{z}, \tag{2.55}$$

$$\inf(\hat{y}, C(\hat{x})) = 0, \tag{2.56}$$

$$F(\hat{x}) = 0. \tag{2.57}$$

(2.55)–(2.57) are refined **Kuhn-Tucker conditions** for the nonlinear program (2.39), cf. [128], and a point satisfying these conditions is called a **Kuhn-Tucker point**.

Example. 2.35. *Lets consider the nonlinear program*

$$\begin{aligned} &\min\ x_1^2 \\ &\text{s.t.}\ \ x_1^2 + x_2^2 - x_3^2 = 0 \\ &\qquad x_2 = 1 \\ &\qquad - x_3 \geq 0. \end{aligned}$$

The point $x^* = (0, 1, -1)$ *is the global minimizer for this problem.*

The generalized Karush-John conditions from Theorem 2.31 read as follows, after we split the linear equation into two inequalities $x_2 - 1 \geq 0$ *and* $1 - x_2 \geq 0$*:*

$$2\kappa \begin{pmatrix} x_1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} + 2z \begin{pmatrix} x_1 \\ x_2 \\ -x_3 \end{pmatrix}$$
$$\inf(y_1, x_2 - 1) = 0$$
$$\inf(y_2, 1 - x_2) = 0$$
$$\inf(y_3, -x_3) = 0$$
$$x_1^2 + x_2^2 - x_3^2 = 0$$
$$\kappa = 1 \text{ or } z \neq 0.$$

At the solution point the conditions become

$$y_1 - y_2 + 2z = 0 \tag{2.58}$$
$$-y_3 + 2z = 0 \tag{2.59}$$
$$y_3 = 0, \tag{2.60}$$

and from (2.59) *and* (2.60) *we get* $z = 0$*, which in turn implies* $\kappa = 1$*, so this example fulfills the constraint qualifications of Corollary 2.34.*

If we do not make any substitution of slack variables, what is useful for implementation, Theorem 2.33 becomes

Theorem 2.36 (General Karush-John conditions).
Let $A_B \in \mathbb{R}^{m_B \times n}$, $A_E \in \mathbb{R}^{m_E \times n}$, $b_L \in \mathbb{R}^{m_L}$, $b_U \in \mathbb{R}^{m_U}$, $b_E \in \mathbb{R}^{m_E}$, $F_L \in \mathbb{R}^{k_L}$, $F_U \in \mathbb{R}^{k_U}$, $F_E \in \mathbb{R}^{k_E}$, and $\mathbf{b_B} = [\underline{b}_B, \overline{b}_B]$ and $\mathbf{F_B} = [\underline{F}_B, \overline{F}_B]$, where $\underline{b}_B, \overline{b}_B \in \mathbb{R}^{m_B}$ and $\underline{F}_B, \overline{F}_B \in \mathbb{R}^{k_B}$. Consider the optimization problem

$$
\begin{aligned}
&\min f(x)\\
&\text{s.t.}\ \ A_B x \in \mathbf{b_B}, \quad A_E x = b_E,\\
&\qquad C_L(x) \ge b_L, \quad C_U(x) \le b_U,\\
&\qquad F_B(x) \in \mathbf{F_B}, \quad F_E(x) = F_E,\\
&\qquad F_L(x) \ge F_L, \quad F_U(x) \le F_U,
\end{aligned}
\tag{OGF}
$$

with C^1 functions $f : \mathbb{R}^n \to \mathbb{R}$, $F_B : \mathbb{R}^n \to \mathbb{R}^{k_B}$, $F_L : \mathbb{R}^n \to \mathbb{R}^{k_L}$, $F_U : \mathbb{R}^n \to \mathbb{R}^{k_U}$, $F_E : \mathbb{R}^n \to \mathbb{R}^{k_E}$, and $C_L : \mathbb{R}^n \to \mathbb{R}^{m_L}$ pseudoconvex on the feasible set, $C_U : \mathbb{R}^n \to \mathbb{R}^{m_U}$ pseudoconcave on the feasible set.

Then there are $\eta \in \mathbb{R}$ and vectors $y_B \in \mathbb{R}^{m_B}$, $y_E \in \mathbb{R}^{m_E}$, $y_L \in \mathbb{R}^{m_L}$, $y_U \in \mathbb{R}^{m_U}$, $z_B \in \mathbb{R}^{k_B}$, $z_E \in \mathbb{R}^{k_E}$, $z_L \in \mathbb{R}^{k_L}$, and $z_U \in \mathbb{R}^{k_U}$, with

$$
\begin{aligned}
\eta g(x) - A_B^T y_B + A_E^T y_E - C_L'(x)^T y_L + C_U'(x)^T y_U&\\
- F_B'(x)^T z_B - F_L'(x)^T z_L + F_U'(x)^T z_U + F_E'(x) z_E = 0&,
\end{aligned}
$$

$$
\begin{gathered}
\inf(z_L, F_L(x) - F_L) = 0, \quad z_L \ge 0,\\
\inf(z_U, F_U(x) - F_U) = 0, \quad z_U \ge 0,\\
z_B \star (F_B(x) - \underline{F_B}) * (F_B(x) - \overline{F_B}) = 0,\\
z_B \star (F_B(x) - \underline{F_B}) \le 0, \quad z_B \star (F_B(x) - \overline{F_B}) \le 0,\\
F_E(x) = F_E,\\
\inf(y_L, C_L(x) - b_L) = 0, \quad y_L \ge 0,\\
\inf(y_U, C_U(x) - b_U) = 0, \quad y_U \ge 0,\\
y_B \star (A_B x - \underline{b_B}) * (A_B x - \overline{b_B}) = 0,\\
y_B \star (A_B x - \underline{b_B}) \le 0,\\
y_B \star (A_B x - \overline{b_B}) \le 0,\\
A_E x = b_E, \quad \eta \ge 0,\\
\eta + z_B^T z_B + e^T z_L + e^T z_U + z_E^T z_E = 1,
\end{gathered}
\tag{2.61}
$$

where $e = (1, \ldots, 1)^T$.

Proof. This follows directly from Theorem 2.33. $\square$

This form of the Karush-John conditions is used in the implementation of the Karush-John condition generator (see Section 6.1.6) in the COCONUT environment.

2.2.5 Second Order Optimality Conditions

Until now we have found generalizations of the first order Theorems 2.27 and 2.28. We now extend Theorem 2.22 containing the statements about the Hessian to the constrained case.

In the course of this section, we will transform the problem by introducing slack variables and by bounding all variables, if necessary by huge artificial bounds, to the form

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) = 0 \\ &\quad\ x \in \boldsymbol{x}. \end{aligned} \tag{ON}$$

The following result is due to NEUMAIER [161]

Theorem 2.37. *Let $\hat{x}$ be a Kuhn-Tucker point of* (ON). *Let $\hat{y}$ be the Lagrange multiplier. Set*

$$\hat{z} := g(\hat{x}) + F'(\hat{x})^T\hat{y}, \tag{2.62}$$

$$D = \operatorname{Diag}\left(\sqrt{\frac{2|\hat{z}_1|}{u_1 - l_1}}, \dots, \sqrt{\frac{2|\hat{z}_n|}{u_n - l_n}}\right). \tag{2.63}$$

If for some continuously differentiable function $\varphi : \mathbb{R}^m \to \mathbb{R}$ with

$$\varphi(0) = 0, \quad \varphi'(0) = \hat{y}^T, \tag{2.64}$$

*the **general augmented Lagrangian***

$$\hat{L}(x) := f(x) + \varphi(F(x)) + \tfrac{1}{2}\|D(x - \hat{x})\|_2^2 \tag{2.65}$$

is convex in $[l, u]$, then $\hat{x}$ is a global solution of (ON). *If, moreover, $\hat{L}(x)$ is strictly convex in $[l, u]$ this solution is unique.*

Proof. in NEUMAIER [163, 1.4.10] □

Having this tool at hand, we derive the following generalization of the optimality conditions in Theorem 2.22.

Theorem 2.38 (Second order necessary optimality conditions).
Consider the optimization problem (ON). *Let f and F be C^2 functions on the box $[l, u]$. If $\hat{x}$ is a local solution of* (ON) *we define the set of active indices as*

$$J_a := \{j \mid \hat{x}_j = l_j \text{ or } \hat{x}_j = u_j\}.$$

If in addition the constraint qualification (2.54) *is satisfied, then the following equivalent conditions are true:*

(i) *If $F'(\hat{x})s = 0$ and $s_{J_a=0}$ then $s^T\hat{G}s \geq 0$.*
(ii) *For some matrix (and hence all matrices) Z_0 whose columns form a basis of the subspace defined by $F'(\hat{x})s = 0$ and $s_{J_a} = 0$ the matrix $Z_0^T\hat{G}Z_0$ is positive semidefinite.*

Here $\hat{G}$ is the Hessian of the Lagrangian.

Proof. in Neumaier [163, 1.4.11]. ▯

As in Theorem 2.22 there is a sufficient condition, as well.

Theorem 2.39 (Second order sufficient optimality conditions).
In the setting of Theorem 2.38 we define

$$J_1 = \{j \mid \hat{y}_j \neq 0\}, \quad J_0 = \{j \mid \hat{y}_j = 0\}.$$

If the point $\hat{x}$ satisfies any of the following, equivalent conditions, it is an isolated local minimizer.

(i) *If $F'(\hat{x})s = 0$ and $s_{J_1=0}$ then $s^T\hat{G}s > 0$.*
(ii) *For some matrix (and hence all matrices) Z whose columns form a basis of the subspace defined by $F'(\hat{x})s = 0$ and $s_{J_1} = 0$ the matrix $Z^T\hat{G}Z$ is positive definite.*
(iii) *For some matrix (hence all matrices) A whose rows form a basis of the row space of $F'(\hat{x})_{:J_0}$, the matrix*

$$\hat{G}_{J_0J_0} + \beta A^TA$$

is positive definite for some $\beta \geq 0$.

If (i)–(iii) hold, $\hat{x}$ is called a **strong local minimizer**. The matrices $Z_0^T\hat{G}Z_0$ and $Z^T\hat{G}Z$ are called **reduced Hessians** of the Lagrangian.

Proof. in Neumaier [163, 1.4.13] ▯

Because of the first order optimality conditions, the sets J_a and J_1 always fulfill $J_a \subseteq J_1$. Therefore, the conditions in Theorem 2.39 are much stronger than those in Theorem 2.38, even more than the strictness of the inequality (or the positive definiteness in contrast to positive semidefiniteness) would suggest. However, if we have $J_a = J_1$ the **strict complementarity condition** at least for some multiplier $\hat{y}$, the gap between necessary and sufficient conditions is as small as in the unconstrained case (Theorem 2.22).

Chapter 3
Global Optimization and Constraint Satisfaction

In this chapter we shall attack the global optimization problem. First we will review a few more applications, where finding the *global* minimizer is essential. Then we will consider various paths different research teams have taken to solve the optimization problem.

For most of the chapter we will consider the problem

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t. } F(x) \in \boldsymbol{F} \\ &\qquad x \in \boldsymbol{x} \end{aligned} \tag{GO}$$

with $f : \mathbb{R}^n \to \mathbb{R}$ and $F : \mathbb{R}^n \to \mathbb{R}^m$ continuously differentiable (sometimes twice), and $x \in \mathbb{IR}^n$, $\boldsymbol{F} \in \mathbb{IR}^m$ hypercubes. For a definition of $\mathbb{IR}$ see Section 3.4 on interval analysis. Now, in contrast to Chapter 2 now we are interested in the global minimum and the global minimizer of (GO). Most algorithms presented in this chapter require some structural information on f and F, as well as a possibility to automatically compute bounds on the function value or other global information. In order to do so, one usually needs to know *explicit expressions* defining the functions. An optimization problem in which all functions can be expressed by a finite number of arithmetical operations and elementary functions, is called **factorable**.

In 1975 Dixon & Szegö [45] edited the first book completely devoted to global optimization, and they have provided the basis for that evergrowing field (in interest and theory). Since then, various research groups have made huge progress, and they have pushed the field to a point, where it starts to become interesting for applications in engineering, industry, and theoretical sciences. In this chapter I will discuss in an informal manner the various strands of development around. Contents and presentation of this chapter were greatly influenced by the survey work by Neumaier [168].

Global optimization programs need methods for automatically generating estimates on the functions f and F in (GO). Very well suited for that task is interval analysis (see Section 3.4). Some of the ideas used to solve discrete constraint satisfaction problems, notably constraint propagation (see Section 3.5), carry over to continuous CPs and to global optimization problems, too.

In Chapter 2 we developed theory about the optimizers of (GO) and similar problems, especially optimality conditions. Some of these results do not only provide local but also global information. Theorem 2.14 states that for *convex* problems local and global minimizers coincide, and that in combination with Theorem 2.1 for unimodal functions (e.g. for strictly convex functions) exactly one global minimizer exists. For nonconvex problems Theorem 2.37 provides a generalization of the result. However, this result is also connected to convexity, so it is not astonishing that convexity plays an important role in many approaches to global optimization (see Section 3.7).

We have also seen that linear programs are very special convex programs with nice additional properties, e.g., Theorem 2.18. Since Dantzig's simplex algorithm, linear programming is the field which is most developed, so the use of linear programs during the solution process is another promising tactic, see Section 3.6.

Methods which use piecewise linear approximation of functions and mixed integer linear programming, those which use reformulation techniques, and the algorithms based on algebraic geometry are not presented here. An excellent summary and an extensive list of references can be found in the survey by Neumaier [168].

The chapter closes with a short review of the existing global optimization software packages based on complete techniques.

Neumaier [168] suggests that global optimization algorithms can be classified into four groups, according to the degree of mathematical rigor with which they try to solve the solve problem (GO):

Incomplete: A method of this class is based on heuristics for searching. It has no safeguards against getting stuck in a local minimum, and it usually does not even have means to detect how close to the global minimum it has got. So termination criteria have to be heuristic, too.

Asymptotically Complete: An algorithm belongs to this graph, if it can be proved that it will eventually reach the global optimum (up to a given tolerance) at least with probability one, if the running time is not limited. However, these methods do not have means to determine when they have reached the global optimum, and so termination criteria are heuristic.

Complete: This group of methods certainly finds the global minimum, if exact computations and infinite running time are assumed. If the goal is to find an approximation to the global minimum within a specified tolerance, the algorithm is guaranteed to end in finite time.

Rigorous: This subgroup of the complete methods finds the global minimum even in the presence of rounding errors, except in degenerate or near-degenerate cases.

In the literature, other terms are in regular use, as well. Incomplete methods are often referred to as *heuristic methods*, though *genetic algorithms*, a special class of incomplete methods, often get sorted into their own group. Another term used for asymptotic methods is *stochastic methods*, and the last two groups are often subsumed in the term *deterministic methods*. Especially the last name is slightly confusing, because most heuristic and even some "stochastic" methods are deterministic (i.e. do not depend on randomness), as well.

The main focus in this chapter will be on complete methods, aiming at full rigor.

3.1 Applications

In Section 1.4 we have seen that there is a number of applications requiring solution algorithms, which perform a complete search and thereby *proof* global optimality of the solution found.

In Sections 3.1.1 and 3.1.2 I will present two more important applications for global optimization.

3.1.1 Computer-assisted Proofs

A computer-assisted proof is a piece of mathematics, which would not have been possible without the help of one or a set of algorithms running on a computer. Till today, computer-assisted proofs leave a certain amount of uneasiness in the minds of the mathematicians, since checking them for errors is a highly non-trivial task.

Moreover, it has to be made sure that computational errors, like round-off errors cannot have a negative influence on the result. This requires the use of interval analysis (see Section 3.4).

Nontheless, since the 1970 several famous mathematical conjectures have been proved in a computer-assisted way.

The Four Color Theorem

Almost every atlas contains, so called *political maps*, maps where different countries, states, or districts are colored differently, so that we can easily identify the boundaries in between.

In the year 1852 Francis Guthrie, while he tried to color the map of counties of England, noticed that four colors sufficed. He asked Frederick, his brother, if any map can be colored using four colors in such a way that regions sharing a common boundary segment receive different colors. Frederick Guthrie then asked DeMorgan, and the first printed reference is due to Cayley in 1878. Then proofs were given, e.g., by Kempe 1879 and Tait 1880, which both turned out to contain faults.

After a result by Birkhoff, Franklin proved 1922 that the four color conjecture was true for maps with at most 25 countries. Finally, Heesch discovered the two main ingredients of the later proof — reducibility and discharging, and he conjectured that a development of a method based on the two facts would ultimately solve the Four Color Problem.

In the end, 1976 Appel & Haken [4, 5, 6] proved the conjecture. That proof, based on Heesch's ideas, is very long, very difficult to check, and it heavily relies on computer power.

The idea of the proof is as follows. First, the map is replaced by a graph $\Gamma = (V, E)$ (for a short introduction to graph theory see, e.g., Wilson [229]). Every country corresponds to a vertex of Γ, and two vertices are connected by an edge, iff the corresponding contries have a common border. The resulting graph is a planar graph. A graph is called n-colorable, if there is a map $\varphi : V \to \{1, \dots, n\}$ such that the endpoints of every edge E of Γ have different φ-values. It is easy to show that a planar graph is 6-colorable (a popular exercise in basic graph theory courses), and the proof that every planar graph is 5-colorable is more elaborate but still presentable in a graph theory course. Since it is obvious that there are planar graphs, which are not 3-colorable, the remaining question was: "Is the minimal number of colors 4 or 5?"

If not all planar graphs are 4-colorable, there must be a minimal counterexample. In 1913 Birkhoff [22] proved that every minimal counterexample to the Four Color Theorem is an internally 6-connected triangulation.

The proof of Appel and Haken first shows that a minimal counterexample to the Four Color Theorem contains at least one of a set of 1476 basic graphs. For those it had to be decided whether they are reducible, a techical term implying that the minimal counterexample cannot contain such a subgraph. This large number of *discrete constraint satisfaction problems* was finally solved with the support of a super computer.

The proof remained disputed for a long time, because it was very difficult to check, not just the algorithmic part, but in the end it was accepted to be true.

In 1996 ROBERTSON ET AL. gave a new proof, which is easier to check but still needs the solution of 633 constraint satisfaction problems. It is based on the result of Birkhoff, and proceeds by showing that at least one of the 633 configurations appears in every internally 6-connected planar triangulation. This is called proving unavoidability. The 633 discrete constraint satisfaction problems are proved by computer support.

One offspring of the proof is the *Seymour problem*, formulated by Seymour and contained in the MIPLIB [23] library of mixed integer linear problems. Its solutions produce the smallest set of unavoidable configurations which have to be reduced for proving the Four Color Theorem. This is a very hard *discrete global optimization problem* which was solved 2001 by FERRIS, BATAKI & SCHMIETA [51] using massive parallel computing within 37 days using the `Condor` [37] system.

Kepler's conjecture

In 1611, Kepler proposed that cubic or hexagonal packing, both of which have maximum densities of $\frac{\pi}{3\sqrt{2}}$, are the densest possible sphere packings in $\mathbb{R}^3$, the Kepler conjecture.

The first proof was claimed 1975 by Buckminster Fuller, but as pointed out by Sloane 1998, it was flawed. Another wrong proof of the Kepler conjecture was put forward by W.-Y. Hsiang 1992 and 1993, but was subsequently determined to be incorrect. According to J.H. Conway, "nobody who has read Hsiang's proof has any doubts about its validity: it is nonsense".

HALES devised in 1997 a detailed plan describing how the Kepler conjecture might be proved using a different approach by making extensive use of computer calculations. Hales subsequently managed to complete a full proof, which appears in a series of papers of more than 250 pages [72]. A broad outline of the proof in elementary can be found in [73]. The proof is based on ideas from global optimization, linear programming, and interval arithmetic. The computer files containing all results, the computer code, and the data files for combinatorics, interval arithmetic, and linear programs need more than 3 gigabytes (!) of storage space.

Hales' proof has been so difficult to verify that after its submission to the Annals of Mathematics it took a team of 12 reviewers more than four years to verify the proof, and they are not more (but also not less) than 99% certain that it is correct. Very likely, the proof will be published in 2004, and when it appears, it will carry an unusual editorial note stating that parts of the paper have not been possible to check.

In recent years, many more theorems have been proved in a computer assisted way. For an overview on some results see FROMMER [60].

The *double bubble conjecture*, i.e., the conjecture that the smallest surface enclosing two given volumes of equal sizes is a double bubble, a fusion of two spheres intersecting each other, was proved by HASS ET AL. [77] in 1995.

Feffermann & Seco proved in the years 1994 and 1995 [50] the *Dirac–Schwinger conjecture* about the asymptotic behaviour of the ground state energy $E(N,Z)$ of an atom with N electrons and kernel charge Z

$$\lim_{Z\to\infty} E(N,Z) = -c_0 Z^{7/3} + \tfrac{1}{8}Z^2 - c_1 Z^{5/3} + o(Z^{5/3})$$

by hard analysis and computer-assistence using interval analytic methods for ordinary differential equations.

The most recent result from 2002 by Tucker [222], who solved Smale's 14th problem for the new century [213] of proving that the Lorenz-attractor is a strange attractor, by theoretical work and computer support using interval methods for ordinary differential equations.

3.1.2 Worst Case Studies

Whenever real world objects like buildings or brigdes are built, they are planned on a computer, and their stability is verified using *finite element methods*. One possible way of solving the stability problem is to approximate structures by systems of simple elements, mostly by truss systems. The material parameters of the original structure are translated to material parameters of the bars, and then the truss structure is analyzed. A wall could, e.g., be modeled as in Figure 3.1.

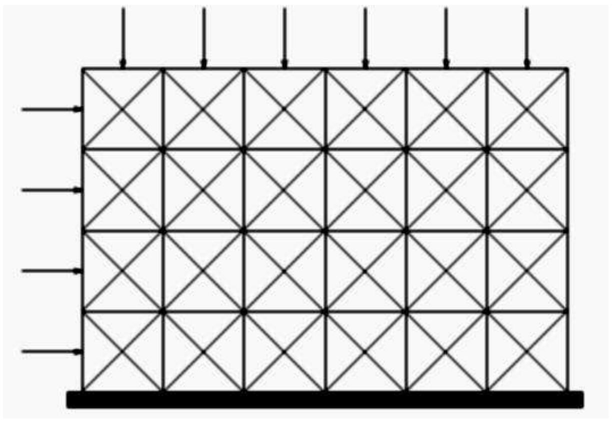


Figure 3.1. *Finite element representation of a wall*

The material parameters are not known exactly for any real building. The elasticity module of steel bars usually is only known up to an error of 15%, even the length of bars is accurate only up to ca. 1%. The distribution of forces on the structure is not constant, and even if the maximum load is known, the exact distribution of loads cannot be predicted beforehand.

Still, it is important to know the maximum stresses and torsion forces, which are possible in the structure. This maximal strain has to be within a certain limit, otherwise the structure will be in danger of collapsing.

In engineering the problem of finding the maximum stress is approached by a combination of Monte-Carlo simulation and local optimization. However, Balakrishnan & Boyd [10] have shown that this method may severly underestimate the true global maximum, thereby grossly underestimating the true risk.

In mathematical terms, the problem is a global optimization problem. The linear finite element equations become nonlinear if uncertainties are involved. In this case, the transfer matrix K depends on the uncertainties, and the problem becomes

$$A(x)u = b,$$

where the u are the displacements, the b depend on the forces, and $A(x)$ is the transfer matrix depending on the uncertain material parameters x. The problem is to compute the maximal displacement u, a non-linear global optimization problem. Investigating a 20×20 wall like the 6×4 structure in Figure 3.1 is about the smallest case already interesting for the industry. This problem has 1640 material parameters, and 840 displacements. Solving this problem to global optimiality is out of reach for current global optimization algorithms, although there have been recent promising results by Muhanna & Mullen [154] on computing strong upper bounds for such problems; Neumaier & Pownuk [169] and have successfully solved similar problems.

Still, the solution of finite element worst case studies will remain a challenging problem for years to come.

3.2 Heuristic Approaches and other Incomplete Methods

For solving global optimization probles a big number of incomplete methods have been invented. The easiest is *multiple random start*, which performs many local optimizations from random starting points in the search space. The more prominent techniques are *simulated annealing* (Ingber [98]), *genetic algorithms* (Holland [87]), see Mongeau et al. [150], Janka [100] for comparisons.

There are some asympotically complete methods, which only use point evaluations, which lie between heuristic approaches and complete methods, e.g., MCS by Huyer & Neumaier [95] or DIRECT by Gablonsky & Kelley [61]. Anyway, they are still incomplete and do not guarantee finding the global minimum in finite time.

Incomplete methods are not directly related to this work, and so we will not get into detail. However, a fast incomplete method can serve the need of finding a good feasible point, perhaps the global optimum at a very early stage in the solution process of a complete method. This makes constraint propagation more efficient and helps to reduce the search space. For a short summary of the various techniques used in incomplete methods see Neumaier [168, 7.].

3.3 Complete Methods — Branch and Bound

A *complete method* for solving problem (GO) is a method which in exact arithmetic guarantees to find in finite time *all* solutions within a prespecified tolerance. In order to realize that goal the methods have to analyze the whole feasible set. Since *finding* the set of feasible points in general is at least as difficult as solving the optimization problem, most complete methods search analyze a superset of the feasible region. This superset is called the **search space**.

The simplest method for examining the search space is **grid search**, which puts a hierarchy of finer and finer grids on the space, computes function values on the grid points, and uses those points as starting points for local optimization algorithms. Grid search is not very efficient in dimensions higher than two, since the number of grid points grows exponentially with the dimension.

Therefore, the vast majority of complete algorithms take another approach. They are variants of the **branch-and-bound** scheme (see Figure 3.2). If we denote the search space by S, a set L of subsets of S is kept, initialized by $L = \{S\}$. Then the branch-and-bound loop starts by choosing a set P from L. Afterwards, depending on the algorithm, a number of mathematical methods is applied to check, whether the set P contains solutions to problem (GO), the **bound** phase. If this is not the case, P is removed from L and the branch-and-bound loop continues. If P cannot be proved to be disjoint with the solution set S_s, it is **split** in a familiy of subsets $P_1, \dots, P_k$ (most of the time $k = 2$, then the split is called **bisection**), the **branch** phase. In L the problem P is replaced by all the P_i, and the branch-and-bound loop continues. If the set P is smaller than the tolerances, it is stored in the set of possible solutions S_p and removed from L, and again the loop continues. If the set L is empty the algorithm stops. When the algorithm terminates, we have $S_s \subseteq \bigcup_{U \in S_p} U \subseteq S$, and S_p is a covering of the solution set with small subsets of the search space which are smaller than the tolerances.

More advanced algorithms (see Figure 3.3) enhance the scheme by using analyzation methods, which prove statements of the form: If we only consider solutions in $P \subseteq S$, then there exists a set $P' \subsetneq P$ with $P \cap S_s \subseteq P'$. In that case the problem P can be replaced by P' in L. Such a step is called a **reduction** step. Discarding P is equivalent to proving that it can be reduced to the empty set.

The various algorithms for global optimization differ mainly in the methods applied for computing reductions and for proving infeasibility. All of these proofs depend on the ability to prove global, or at least semilocal, results about function ranges, or zeros, or the like. In the following sections we will review these methods. Most of them either rely on *interval analysis*, an algorithmic way of proving globally valid estimates, or on *convexity*.

3.4 Interval Analysis

A large class of complete global optimization algorithms uses in the course of the algorithms interval arithmetic, an extension of the real (or complex) arithmetic to intervals. The field was started by Moore [152, 151], see also Kahan [106]; a thor-

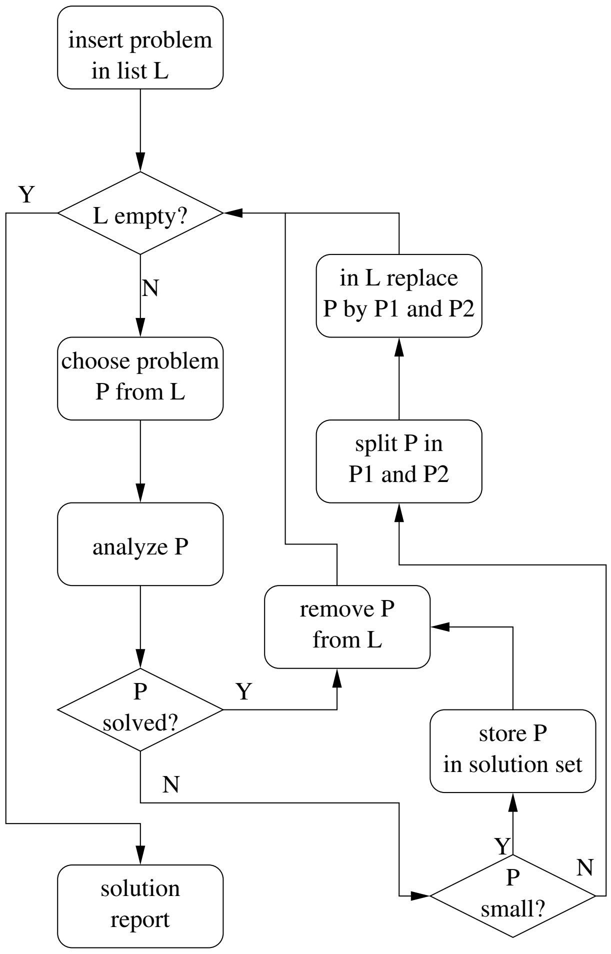


Figure 3.2. *The Branch-and-Bound scheme*

ough review of the field is given in NEUMAIER [159]. Its strength for solving global optimization problems was noticed soon, and HANSEN [75, 76], and KEARFOTT [114] have applied and implemented the methods.

Interval analysis is important for global optimization in many aspects. It provides cheaply algorithmically computable globally valid bounds for all functions, for which an explicit mathematical expression can be given. The bounds are very wide in many cases, but by additional effort (splitting of the area into subparts) the bounds can be tightened. Furthermore, interval analysis extends the classical analysis by providing *semilocal* existence and uniqueness results. While real analysis usually only proves the existence of a neighborhood U around a point x, in which x is the only point having some important property, interval analysis provides to tools for computing U explicitly.

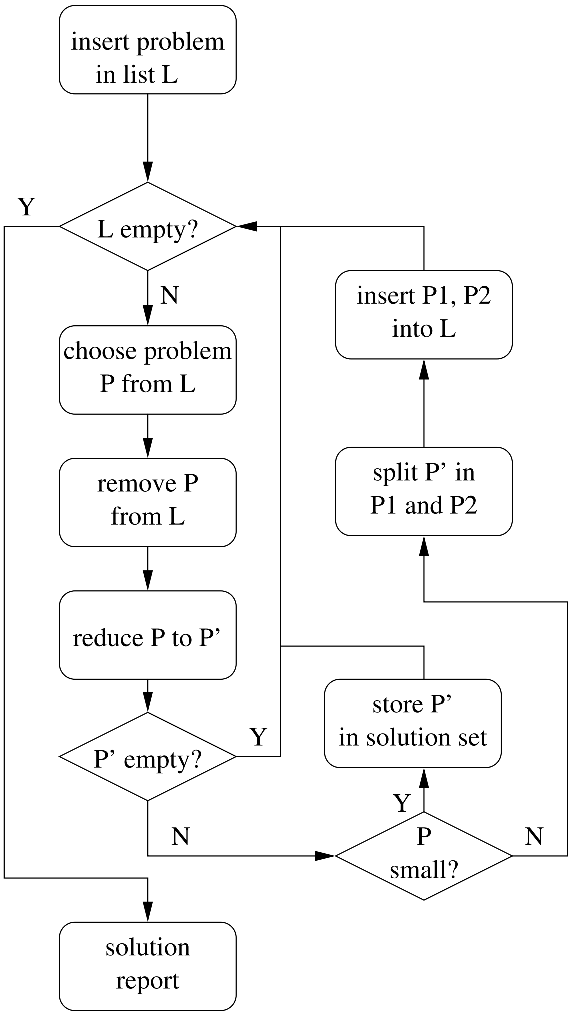


Figure 3.3. *Branch-and-Reduce scheme*

3.4.1 Interval Arithmetic

The basics of interval analysis is an extension of real arithmetic to intervals. Let $S \subseteq \mathbb{R}$ be any set, and define $\Box S \in \mathbb{IR}$ as

$$\Box S := [\inf S, \sup S].$$

We identify a real number $t \in \mathbb{R}$ with the point interval $[t, t]$, and so $\mathbb{R} \subseteq \mathbb{IR}$. In the following we will write t for the point interval $[t, t]$.

We will also need some additional notation. Let $\boldsymbol{x} \in \mathbb{IR}$ be an interval. We will denote the lower bound of $\boldsymbol{x}$ as $\underline{x} := \inf \boldsymbol{x}$, the upper bound of $\boldsymbol{x}$ as $\overline{x} := \sup \boldsymbol{x}$, and the **midpoint** of $\boldsymbol{x}$ as $\check{x} = \mid \boldsymbol{x} := \frac{1}{2}(\overline{x} + \underline{x})$ whenever both bounds are finite. We will never need the midpoint in case of infinite bounds. The **radius** of the interval is defined as

$$\operatorname{rad} \boldsymbol{x} := \begin{cases} \frac{1}{2}(\overline{x} - \underline{x}) & \text{if } \overline{x} \neq \underline{x} \\ 0 & \text{if } \overline{x} = \underline{x}, \end{cases}$$

the **magnitude** or **absolute value** of $\boldsymbol{x}$ is

$$|x| := \max\{|\underline{x}|, |\overline{x}|\},$$

and the **mignitude** shall be

$$\langle x \rangle := \begin{cases} \min\{|\underline{x}, \overline{x}\} & \text{for } 0 \notin \boldsymbol{x}, \\ 0 & \text{otherwise.} \end{cases}$$

Using that definition, we can extend the arithmetic operations and functions $f : \mathbb{R} \supseteq D_f \to \mathbb{R}$ to $\mathbb{IR}$, see WALSTER ET AL. [34, 48]. For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$ define

$$\begin{aligned} \boldsymbol{x} \circ \boldsymbol{y} &:= \square\{x \circ y \mid x \in \boldsymbol{x},\ y \in \boldsymbol{y}\}, \\ f(\boldsymbol{x}) &:= \square(\{f(x) \mid x \in \boldsymbol{x} \cap D_f\} \cup \{\lim_{y \to x} f(y) \mid y \in D_f,\ x \in \boldsymbol{x}\}), \end{aligned}$$

where $\circ$ is any of the arithmetic operators in $\{+, -, *, /, \hat{}\}$.

An easy argument implies for the arithmetic operators

$$\boldsymbol{x} + \boldsymbol{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \quad \boldsymbol{x} - \boldsymbol{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}], \quad \boldsymbol{x} * \boldsymbol{y} = \square\{\underline{xy}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{xy}\},$$

and whenever any arithmetic operation would result in an undefined expression, the interval result is $[-\infty, \infty]$. Integer division is a bit more complicated:

$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{x}/\boldsymbol{y}$
$0 \in \boldsymbol{x}$	$0 \in \boldsymbol{y}$	$[-\infty, \infty]$
$\overline{x} < 0$	$\underline{y} < \overline{y} = 0$	$[\overline{x}/\underline{y}, \infty]$
$\overline{x} < 0$	$\underline{y} < 0 < \overline{y}$	$[-\infty, \infty]$
$\overline{x} < 0$	$0 = \underline{y} < \overline{y}$	$[-\infty, \overline{x}/\overline{y}]$
$\underline{x} > 0$	$\underline{y} < \overline{y} = 0$	$[-\infty, \underline{x}/\underline{y}]$
$\underline{x} > 0$	$\underline{y} < 0 < \overline{y}$	$[-\infty, \infty]$
$\underline{x} > 0$	$0 = \underline{y} < \overline{y}$	$[\underline{x}/\overline{y}, \infty]$
	$0 \notin \boldsymbol{y}$	$\square\{\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}\}$

These definitions extend to higher dimensions in a straightforward way. Note that many algebraic properties of the real arithmetic operations do not carry over to interval arithmetic. We have, e.g., that $\boldsymbol{x} - \boldsymbol{x} \neq 0$ but only $\boldsymbol{x} - \boldsymbol{x} \ni 0$. Furthermore, the **subdistributivity** law

$$(\boldsymbol{x} + \boldsymbol{y})\boldsymbol{z} \subseteq \boldsymbol{xz} + \boldsymbol{yz}$$

holds instead of distributivity.

One strength of interval arithmetic is that computing globally valid bounds on nonlinear functions is straightforward. Take a function $f : \mathbb{R} \to \mathbb{R}$, and let $\mathfrak{f}$ be a mathematical expression for evaluating f containing arithmetic operators and elementary functions ($e^x, \log, \sqrt{\ }, \sin, \cos, \dots$) — note that this imposes strong restrictions on f. We define $\mathfrak{f}(\boldsymbol{x})$ as the interval, which results by replacing all real valued operations in $\mathfrak{f}$ by their interval equivalents. The properties of interval arithmetic imply

$$\mathfrak{f}(\boldsymbol{x}) \supseteq f(\boldsymbol{x}),$$

so any mathematical expression $\mathfrak{f}$ for f can be used to enclose the range of the function f. It is important to note that mathematically equivalent expressions do not yield the same range estimate. We have, e.g., $f(x) = x^2 = (x-2)(x+2)+4$ and $[-1,1]^2 = [0,1]$ but $([-1,1]-2)([-1,1]+2)+4 = [-5,3]$. In the second expression the variable appears more than once, and so it suffers from **inner dependency**, resulting in an **overestimation** of the range. In general, this effect cannot be prevented but at least under very mild conditions (see Neumaier [159, 1.4]) the overestimation satisfies the **linear approximation property**

$$\mathfrak{f}(\boldsymbol{x}) \subseteq f(\boldsymbol{x}) + O(\operatorname{rad}\boldsymbol{x}).$$

3.4.2 Centered Forms and Slopes

For many applications the linear approximation property of interval arithmetic is not good enough. However, for differentiable functions f we can obtain tighter enclosures by using a **centered form**. The simplest centered, the **mean value form** is based on the mean value theorem, which states

$$f(x) = f(z) + f'(\xi)(x-z), \quad \text{with } \xi \in [x,z].$$

Now let $z \in \boldsymbol{x}$ be fixed. In this case $\xi \in \boldsymbol{x}$ whenever $x \in \boldsymbol{x}$, and so

$$f(\boldsymbol{x}) \subseteq f(z) + f'(\boldsymbol{x})(\boldsymbol{x}-z).$$

The mean value form, as all other centered forms, has the **quadratic approximation property**

$$f(z) + f'(\boldsymbol{x})(\boldsymbol{x}-z) \subseteq f(\boldsymbol{x}) + O\big((\operatorname{rad}\boldsymbol{x})^2\big).$$

However, the mean value form, which uses the **interval derivative** $f'(\boldsymbol{x})$ is not the most efficient centered form. A better one is based on **slopes**. A slope is a linear approximation of the form

$$f(x) = f(z) + f[z,x](x-z), \tag{3.1}$$

see [208, 122]. In one dimension the slope is unique, if it is continuous, and we have

$$f[z,x] = \begin{cases} \dfrac{f(x)-f(z)}{x-z} & x \neq z \\ f'(z) & x = z. \end{cases}$$

In higher dimensions the slope is nonunique (see, e.g., Section 4.1.7), but it exists always if f is locally Lipschitz.

Using (3.1) we find an enclosure of the range of f over the box $\boldsymbol{x}$ by

$$f(x) \in f(z) + f[z, \boldsymbol{x}](\boldsymbol{x} - z), \quad \text{for all } x \in \boldsymbol{x}.$$

This is a centered form and also has the quadratic approximation property. The most general slope definition is the one with interval center

$$f(\boldsymbol{x}) \subseteq f(\boldsymbol{z}) + f[\boldsymbol{z}, \boldsymbol{x}](\boldsymbol{x} - \boldsymbol{z}),$$

and the special case $\boldsymbol{x} = \boldsymbol{z}$ gives $f[\boldsymbol{z}, \boldsymbol{z}] = f'(\boldsymbol{z})$ the interval derivative and hence the mean value form.

Slopes can be calculated automatically like derivatives, a chain rule holds:

$$(f \circ g)[\boldsymbol{z}, \boldsymbol{x}] = f[g(\boldsymbol{z}), g(\boldsymbol{x})] \cdot g[\boldsymbol{z}, \boldsymbol{x}]. \tag{3.2}$$

Thus, it is necessary to determine the slopes for the basic arithmetic operations. For addition and subtraction this is straightforward, and for multiplication and division we discuss the variants in Section 4.1.7. For elementary functions we can use the result by Kolev [122] that for convex and concave functions the optimal slope is given by

$$\varphi[z, \boldsymbol{x}] = \square\{\varphi[z, \underline{x}], \varphi[z, \overline{x}]\},$$

and case distinctions. For the other functions, the case is more difficult, but we can always use

$$\varphi[z, \boldsymbol{x}] \subseteq \varphi'(\boldsymbol{x}).$$

Centered forms based on higher order Taylor expansions are interesting, as well. The second order slopes are discussed in Section 3.4.4, and for even higher order methods see, e.g. Makino & Berz [140], Carrizosa et al. [31], and the survey by Neumaier [165]. Higher order expansions in principle lead to higher order approximation properties. However, there are no known methods which do not need computational effort exponential in the dimension to achieve that, at least in certain cases. Despite of that, higher order expansions might still be useful, if interval evaluation suffers from very strong inner dependency.

3.4.3 Interval Newton Method

For verifying (approximate) solutions of nonlinear systems of equations we will need another important tool, the interval Newton methods. Krawczyk [125] invented the following principle for verifying solutions of a system of equations $F(x) = 0$ with $x \in \boldsymbol{x}$. Given any regular preconditioning matrix C we define the **Krawczyk operator** $K(\boldsymbol{x}, z)$ (see Krawczyk [125], Kahan [106])

$$K(\boldsymbol{x}, z) := z - CF(z) + (I - CF'(\boldsymbol{x}))(\boldsymbol{x} - z).$$

Because of the mean value theorem, we have $x \in K(\boldsymbol{x}, z)$ if $x, z \in \boldsymbol{x}$, and the following central result holds:

Theorem 3.1 (Krawczyk operator).

(i) *Every zero* $x \in \boldsymbol{x}$ *of* F *lies in* $\boldsymbol{x} \cap K(\boldsymbol{x}, z)$.
(ii) *If* $\boldsymbol{x} \cap K(\boldsymbol{x}, z) = \emptyset$ *then* $\boldsymbol{x}$ *contains no zero of* F,
(iii) *If* $K(\boldsymbol{x}, z) \subseteq \int \boldsymbol{x}$ *then* $\boldsymbol{x}$ *contains a* unique *zero of* F.

Shen & Neumaier [208] have shown that for a variant of the Krawzcyk operator $K'(\boldsymbol{x}, z) := z - CF(z) + (I - CF[z, \boldsymbol{x}])(\boldsymbol{x} - z)$ with slopes all but the uniqueness result is true, as well. Since slopes usually are tighter than interval derivatives, K' has better contraction properties than K.

The results of Theorem 3.1 are important, because due to Theorem 2.33 the operator can be applied to the Karush-John first order optimality conditions. The uniqueness result allows the elimination of large regions around local optima (get rid of the cluster effect, see Section 3.4.6), and it is useful for reducing boxes in a branch-and-reduce scheme. This method is implemented in `GlobSol` [116] and `Numerica` [85]. The theorem is also the basis for the exclusion box results by Schichl & Neumaier [200] presented in Section 4.2.

3.4.4 Second order slopes

Since F is twice continuously differentiable, we can always (e.g., using the mean value theorem) write

$$F(x) - F(z) = F[z, x](x - z) \tag{3.3}$$

for any two points x and z with a suitable matrix $F[z, x] \in \mathbb{R}^{n \times n}$, continuously differentable in x and z; any such $F[z, x]$ is called a **slope matrix** for F. While (in dimension $n > 1$), $F[z, x]$ is not uniquely determined, we always have (by continuity)

$$F[z, z] = F'(z). \tag{3.4}$$

Thus $F[z, x]$ is a slope version of the Jacobian. There are recursive procedures to calculate a slope $F[z, x]$ given x and z, see Krawczyk & Neumaier [126], Rump [189] and Kolev [122]; a Matlab implementation is in Intlab [190].

Since the slope matrix $F[z, x]$ is continuously differentiable, we can write similarly

$$F[z, x] = F[z, z'] + \sum (x_k - z'_k) F_k[z, z', x] \tag{3.5}$$

with **second order slope matrices** $F_k[z, z', x]$, continuous in z, z', x. Here, as throughout this paper, the summation extends over $k = 1, \dots, n$. Second order slope matrices can also be computed recursively; see Kolev [122]. Moreover, if F is quadratic, the slope is linear in x and z, and the coefficients of x determine constant second order slope matrices without any work.

If $z = z'$ the formula above simplifies somewhat, because of (3.4), to

$$F[z, x] = F'(z) + \sum (x_k - z_k) F_k[z, z, x]. \tag{3.6}$$

3.4.5 Moore–Skelboe

The first algorithm for solving global optimization problems was designed by MOORE [152, 151] and later improved by SKELBOE [212]. The **Moore-Skelboe algorithm** was designed for solving global bound constrained optimization problems, i.e., for finding the exact range (up to tolerances) of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ within an initial box $\boldsymbol{x}_{\text{init}}$. It is a branch-and-bound method, which essentially follows the algorithm depicted in Figure 3.2.

For every box $\boldsymbol{x}$ in the list L of problems, an enclosure of the range of f is computed by interval arithmetic using a mathematical expression $\mathfrak{f}$. Initialize U to the upper bounds of $\mathfrak{f}(\boldsymbol{x}_{\text{init}})$.

From L always that $\boldsymbol{x}$ is selected, for which the lower bound ℓ of $\mathfrak{f}(\boldsymbol{x})$ is minimal. The problem is considered solved, if ℓ is bigger than U. If the upper bound u of $\mathfrak{f}(\boldsymbol{x})$ is smaller than U, we set $U := u$. If the width of the function range $\mathfrak{f}(\boldsymbol{x})$ is smaller than a prespecified tolerance ε (or the radius of the box is smaller than another given tolerance δ), the box $\boldsymbol{x}$ is put into the list of possible solutions. If neither is true, the box is split in two along the biggest edge at the middle point of the range.

The Moore-Skelboe algorithm as described above is not very efficient, and numerous enhancments exist, which will not be described here. However, it can be observed that close to the global minimum the number of boxes in the list grows exponentially with the dimension. This effect will be more closely investigated in the next section in the context of nonlinear equation solving.

3.4.6 The Cluster Effect

As explained by KEARFOTT & DU [117], many branch and bound methods used for global optimization suffer from the so called **cluster effect**. As is apparent from the discussion below, this effect is also present for branch and bound methods using constraint propagation methods to find and verify *all* solutions of nonlinear systems of equations. (See, e.g., VAN HENTENRYCK et al. [85] for constraint propagation methods.) Because of Theorem 2.33 finding local optima and solving nonlinear equations are strongly related. Hence, everything that applies to equation solving is true for optimization. For analyzing the cluster effects for optimization and pure constraint satisfaction, we will closely follow the presentations in NEUMAIER [168, Section 15] and SCHICHL & NEUMAIER [200].

The cluster effect is the excessive splitting of boxes close to a solution and failure to remove many boxes not containing the solution. As a consequence, these methods slow down considerably once they reach regions close to the solutions. The mathematical reason for the cluster effect and how to avoid it will be investigated in this section.

Let us first consider the unconstrained optimization problem

$$\min f(x),$$

let $\hat{x}$ be a global solution, and G be the Hessian of f at $\hat{x}$. In a neighborhood around the global minimizer, Taylor's theorem tells us, if the function is smooth enough, that

$$f(x) = f(\hat{x}) + \tfrac{1}{2}(x-\hat{x})^T G(x-\hat{x}) + O(\|x-\hat{x}\|^3),$$

because of the first order optimality conditions (Theorem 2.22). Suppose that we can compute bounds $\boldsymbol{f}$ on $f(\boldsymbol{x})$ for a box $\boldsymbol{x}$ which satisfy

$$\operatorname{rad}\boldsymbol{f} - \operatorname{rad} f(\boldsymbol{x}) \le \Delta = K\varepsilon^{s+1},$$

where $s \le 2$ if the radius of the box $\boldsymbol{x}$ is ε. In that case we can eliminate all boxes of diameter ε with the exception of those, which contain a point x with

$$\tfrac{1}{2}(x-\hat{x})G(x-\hat{x}) + O(\|x-\hat{x}\|^3) \le \Delta.$$

If the accuracy Δ is small enough the region $\mathcal{C}$ of those points is approximatively ellipsoidal, and its volume is

$$\operatorname{vol}(\mathcal{C}) \approx K\sqrt{(2\Delta)^n/\det G}.$$

Covering $\mathcal{C}$ with boxes of radius ε will, therefore needs *const* $\sqrt{(2\Delta)^n/(\varepsilon^n \det G)}$ of them, at least. If we insert the estimate on the accuracy Δ depending on the order s we find that the number of not eliminated boxes will be

$$\begin{aligned} M\varepsilon^{-n/2} &\quad \text{if } s = 0,\\ M &\quad \text{if } s = 1,\\ M\varepsilon^{n/2} &\quad \text{if } s = 2, \end{aligned}$$

where $M = \mathit{const}/\sqrt{\det G}$. We observe that the number of boxes will grow exponentially with falling ε for approximation order $s = 0$, it is in principle independent of ε for $s = 1$, but it still grows exponentially and may especially be very high for ill-conditioned problems. For $s = 2$, on the other hand, the number of boxes is small for small ε. So in order to avoid the cluster effect, second-order information is essential.

For pure constraint satisfaction problems a similar effect is present, if the solution set is discrete. We consider the nonlinear equation

$$F(x) = 0$$

with $F : \mathbb{R}^n \to \mathbb{R}^n$. Let us assume that for arbitrary boxes $\boldsymbol{x}$ of maximal width ε the computed expression $F(\boldsymbol{x})$ overestimates the range of F over $\boldsymbol{x}$ by $O(\varepsilon^k)$

$$F(\boldsymbol{x}) \in (1 + \boldsymbol{C}\varepsilon^k)\square\{F(x) \mid x \in \boldsymbol{x}\} \tag{3.7}$$

for $k \le 2$ and ε sufficiently small. The exponent k depends on the method used for the computation of $F(\boldsymbol{x})$.

Let $\hat{x}$ be a regular solution of (4.5) (so that $F'(\hat{x})$ is nonsingular), and assume (3.7). Then any box of diameter ε that contains a point x with

$$\|F'(\hat{x})(x-\hat{x})\|_\infty \le \Delta = \boldsymbol{C}\varepsilon^k \tag{3.8}$$

might contain a solution. Therefore, independent of the pruning scheme used in a branch and bound method, no box of diameter ε can be eliminated. The inequality (3.8) describes a parallelepiped of volume

$$V = \frac{\Delta^n}{\det F'(\hat{x})}.$$

Thus, any covering of this region by boxes of diameter ε contains at least V/ε^n boxes.

The number of boxes of diameter ε which cannot be eliminated is therefore proportional to at least

$$\begin{array}{ll} \dfrac{\boldsymbol{C}^n}{\det F'(\hat{x})} & \text{if } k = 1, \\ \dfrac{(\boldsymbol{C}\varepsilon)^n}{\det F'(\hat{x})} & \text{if } k = 2. \end{array}$$

For $k = 1$ this number grows exponentially with the dimension, with a growth rate determined by the relative overestimation C and a proportionality factor related to the condition of the Jacobian.

In contrast, for $k = 2$ the number is guaranteed to be small for sufficiently small ε. The size of ε, the diameter of the boxes most efficient for covering the solution, is essentially determined by the nth root of the determinant, which, for a well-scaled problem, reflects the condition of the zero. However, for ill-conditioned zeros (with a tiny determinant in naturally scaled coordinates), one already needs quite narrow boxes before the cluster effect subsides.

So to avoid the cluster effect, we need at least the quadratic approximation property $k = 2$. Hence, Jacobian information is essential, as well as techniques to discover the shape of the uncertainty region.

A comparison of the typical techniques used for box elimination shows that constraint propagation techniques lead to overestimation of order $k = 1$, hence they suffer from the cluster effect. Centered forms using first order information (Jacobians) as in Krawczyk's method provide estimates with $k = 2$ and are therefore sufficient to avoid the cluster effect, except near ill-conditioned or singular zeros. Second order information as used, e.g., in the theorem of Kantorovich still provides only $k = 2$ in estimate (3.8); the cluster effect is avoided under the same conditions.

For singular (and hence for sufficiently ill-conditioned) zeros, the argument does not apply, and no technique is known to remove the cluster effect in this case. A heuristic that limits the work in this case by retaining a single but *larger* box around an ill-conditioned approximate zero is described in Algorithm 7 (Step 4(c)) of Kearfott [112].

For the general global constrained optimization problem (GO), a mixture of these cases is apparent. In view of the second order optimality conditions, information on the reduced Hessian of the Lagrangian function and the Jacobian information of the constraints are needed to eliminate the cluster effect.

3.4.7 Semilocal existence theorems for zeros of equations

The oldest semilocal existence theorem for zeros of systems of equations is due to Kantorovich [110], who obtained as a byproduct of a convergence guarantee for Newton's method (which is not of interest in our context) the following result:

Theorem 3.2 (Kantorovich). *Let z be a vector such that $F'(z)$ is invertible, and let α and β be constants with*

$$\|F'(z)^{-1}\|_\infty \le \alpha, \quad \|F'(z)^{-1}F(z)\|_\infty \le \beta. \tag{3.9}$$

Suppose further that $z \in \boldsymbol{x}$ and that there exists a constant $\gamma > 0$ such that for all $x \in \boldsymbol{x}$

$$\max_i \sum_{j,k} \left| \frac{\partial^2 F_i(x)}{\partial x_j \partial x_k} \right| \le \gamma. \tag{3.10}$$

If $2\alpha\beta\gamma < 1$ then $\Delta := \sqrt{1 - 2\alpha\beta\gamma}$ is real and

(i) *There is no zero $x \in \boldsymbol{x}$ with*

$$\underline{r} < \|x - z\|_\infty < \overline{r},$$

where

$$\underline{r} = \frac{2\beta}{1+\Delta}, \qquad \overline{r} = \frac{1+\Delta}{\alpha\gamma}.$$

(ii) *At most one zero x is contained in $\boldsymbol{x}$ with*

$$\|x - z\|_\infty < \frac{2}{\alpha\gamma}.$$

(iii) *If*

$$\max_{x \in \mathbf{x}} \|x - z\|_\infty < \overline{r}$$

then there is a unique zero $x \in \boldsymbol{x}$, and this zero satisfies

$$\|x - z\|_\infty \le \underline{r}.$$

The affine invariant version of the Kantorovich theorem given in Deuflhard & Heindl [44] essentially amounts to applying the theorem to $F'(z)^{-1}F(x)$ in place of $F(x)$. In practice, rounding errors in computing $F'(z)^{-1}$ are made, which requires the use of a preconditioning matrix $C \approx F'(z)^{-1}$ and $CF(x)$ in place of $F(x)$ to get the benefits of affine invariance in floating point computations.

Kahan [106] used the Krawczyk operator, which only needs first order slopes (see Section 3.4.2), to make existence statements. Together with later improvements using slopes, his result is contained in the following statement:

Theorem 3.3 (Kahan). *Let $z \in \boldsymbol{z} \subseteq \boldsymbol{x}$. If there is a matrix $C \in \mathbb{R}^{n\times n}$ such that the Krawczyk operator*

$$K(\boldsymbol{z}, \boldsymbol{x}) := z - CF(z) - (CF[\boldsymbol{z}, \boldsymbol{x}] - I)(\boldsymbol{x} - z) \tag{3.11}$$

satisfies $K(\boldsymbol{z},\boldsymbol{x}) \subseteq \boldsymbol{x}$*, then* $\boldsymbol{x}$ *contains a zero of* (4.5)*. Moreover, if* $K(\boldsymbol{x},\boldsymbol{x}) \subseteq \operatorname{int}(\boldsymbol{x})$*, then* $\boldsymbol{x}$ *contains a unique zero of* (4.5)*.*

Shen & Neumaier [208] proved that the Krawczyk operator with slopes always provides existence regions which are at least as large as those computed by Kantorovich's theorem, and since the Krawczyk operator is affine invariant, this also covers the affine invariant Kantorovich theorem.

Recent work by Hansen [76] shows that there is scope for improvement in Krawczyk's method by better preconditioning; but he gives only heuristic recipes for how to proceed. For quadratic problems, where the slope is linear in x, his recipe suggests to evaluate $CF[z,x]$ term by term before substituting intervals. Indeed, by subdistributivity, we always have

$$CA_0 + \sum CA_k(\boldsymbol{x}_k - \boldsymbol{z}_k) \subseteq C\Big(A_0 + \sum A_k(\boldsymbol{x}_k - \boldsymbol{z}_k)\Big),$$

so that for quadratic functions, Hansen's recipe is never worse than the traditional recipe. We will adapt it in Section 4.2 to general functions, using second order slopes, as explained below; in the general case, the preconditioned slope takes the form

$$CF[z,x] = CF[z,z'] + \sum (x_k - z'_k)CF_k[z,z',x], \tag{3.12}$$

or with $z = z'$, as we use it most of the time,

$$CF[z,x] = CF'(z) + \sum (x_k - z_k)CF_k[z,z,x]. \tag{3.13}$$

In Section 4.2, the consequences of this formulation, combined with ideas from Shen & Neumaier [208], are investigated in detail.

Recent work on Taylor models by Berz & Hoefkens [19] (see also Neumaier [165]) uses expansions to even higher than second order, although at a significantly higher cost. This may be of interest for systems suffering a lot from cancellation, where using low order methods may incur much overestimation, leading to tiny inclusion regions.

Combining interval evaluation of functions and their gradient, as well as using the first order optimality conditions, backboxing (i.e., exclusion boxes), and interval Newton techniques in a branch-and-bound scheme are the basis for most interval based complete global optimization algorithms like `GlobSol` [116].

3.5 Constraint Propagation

Purely interval based global optimization algorithms suffer from the problem that the work invested into analyzing a subproblem P from the list L is either fully successful (i.e. P is discarded) or completely in vain (if P has to be split). Most purely interval analytic methods do not provide the possibility for *reducing* P as needed in the branch-and-reduce algorithm (Figure 3.3). The only exception is the interval Newton method, which usually works only near a local minimizer.

In constraint logic programming CLEARY [35] and OLDER & VELLINO invented a method, which used the constraints for reducing the domain of all variables by removing values, which lead to contradictions. This method became the base of the advanced search algorithms in the logic programming languages `Prolog V` and `ECLiPSe`.

It was realized that the methods from the discrete constraint satisfaction case can be carried over to continuous constraint programming and global optimization by using interval analysis, see, e.g., BENHAMOU ET AL. [14, 16], CHEN & VANEMDEN [33], VAN HENTENRYCK [83, 84, 85], or DALLWIG ET AL. [39].

The basic principle is that a constraint of the form

$$f(x, y) \in \boldsymbol{f}$$

poses a restriction on the possible values the variables x and y can take. Consider, e.g., the simple constraint

$$x + y \in [-1, 1],$$

where $x \in [0, 4]$, and $y \in [-1, 3]$. Certain combinations of values for x and y from their specific domains do not satisfy the constraint, and more than that. Setting $x = 4$ there is no possible y such that $x + y \in [-1, 1]$. Hence the value 4 can be removed from the domain of x. More general, the constraint $x + y \in [-1, 1]$ implies that $x \in [-1, 1] - [-1, 3] = [-4, 2]$, and hence the range for x can be reduced to $[0, 4] \cap [-4, 2] = [0, 2]$. In the next step we analyze the constraint again and find that $y \in [-1, 1] - [0, 2] = [-3, 1]$. Thus, we can also reduce the domain of y to $[-1, 3] \cap [-3, 1] = [-1, 1]$. Now we have reached a state where for every value in the domain of either variable there exists a value in the domain of the other variable such that their sum is in $[-1, 1]$. At this point the constraint propagation would have to switch to the next constraint.

The constraint propagation scheme can be generalized to constraints involving elementary functions, and even to general directed acyclic computational graphs, as presented later in Section 4.1.4.

Combining constraint propagation with branch-and-bound is efficient, since splitting the domain of one variable further reduces the domains of other variables.

In the example above, we can split y into $[0, 1]$ and $[-1, 0]$. Constraint propagation on the first interval does not change anything, but with the second interval the domain of x further reduces to $[0, 1]$.

However, since constraint propagation also is a zero-order technique just using interval evaluation, the constraint propagation algorithms also suffer from the cluster effect.

3.6 Linearization Techniques

Another idea for attacking global optimization problems is by solving a sequence of easier problems. We say that an optimization problem

$$\begin{aligned} &\min q(x) \\ &\text{s.t. } R(x) \in \boldsymbol{R} \\ &\quad\;\; x \in \boldsymbol{x} \end{aligned} \tag{RP}$$

is a relaxation of the optimization problem (GO), if the feasible set of (RP) contains the feasible set of (GO) and the objective function q satisfies

$$q(x) \le f(x), \quad \text{for all } x \in \mathcal{F},$$

where $\mathcal{F}$ is the feasible set of (GO). The following important properties of relaxations can then be used:

1. The global minimum q^* of problem (RP) is a lower bound for the global minimum of (GO).
2. If x^* is the global minimizer of problem (RP), $F(x^*) \in \boldsymbol{F}$, and $q(x^*) = f(x^*)$ then x^* is the global minimum of problem (GO).

Usually, the relaxed problem has a much simpler structure than the original one and can be solved with comparably small effort by standard algorithms to global optimality. If relaxations are used in a branch-and-bound scheme, property 1. can be used to discard the subproblem, if the value q^* is higher than the objective function value f_{best} of the best feasible point found so far. Property 2. is a method for proving global optimality of a feasible point.

Because linear programming is the most developed field of optimization, *linear relaxations* are a good choice, and they have been considered since McCormick [143].

After he had given the linear bounds on the constraint $z = xy$, see below, the so-called **reformulation linearizations** were invented by a number of researchers. For a factorable global optimization problem one introduces intermediate variables until all constraints are of one of the following types

$z = \varphi(x)$	unary
$z = x \circ y$	binary
$z \in \boldsymbol{z}$	bound,

and the objective function is linear. The next step is to replace all constraints by linear constraints enlarging the feasible set. For unary functions this can be done by constructing analytically like in Figure 3.4. Constraints involving powers can be rewritten in the form

$$z = x^y \quad \Longleftrightarrow \quad \begin{aligned} z &= \exp z_1 \\ z_1 &= y z_2 \\ z_2 &= \log x. \end{aligned}$$

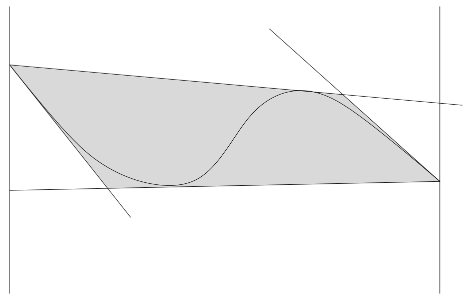


Figure 3.4. *Linear Relaxation of a nonlinear equality constraint*

Therefore, the only operations to be analyzed are products and quotients. For products McCormick provided the linear relaxations

$$\begin{aligned} \underline{y}x + \underline{x}y - \underline{x}\underline{y} &\le z \le \underline{y}x + \overline{x}y - \overline{x}\underline{y} \\ \overline{y}x + \overline{x}y - \overline{x}\overline{y} &\le z \le \overline{y}x + \underline{x}y - \underline{x}\overline{y}, \end{aligned}$$

if bounds $x \in \boldsymbol{x}$ and $y \in \boldsymbol{y}$ are known for the variables. The inequalities are a consequence of positivity relations like $(x - \underline{x})(y - \underline{y}) \ge 0$, and they are the **convex and concave envelope** of the product constraint, as was shown by Al-Khayyal & Falk [1]. For quotients a similar relaxation is valid, which is not the convex and concave envelope. This was computed by Tawarmalani & Sahinidis [220], but the formulas are rather complicated.

The disadvantage of the reformulation method is that for complex expressions the number of intermediate variables is high, and so the dimension of the linear programming problem is far higher than that of the original problem, requiring significant effort for solving it. Since the resulting linear programs are very sparse, a good LP solver is required. Nevertheless, if this prerequisite is met, the reformulation linearization is a very effective method, and `BARON` [220] and `LINGO` [62] mainly rely on that principle.

A different approach, which does not increase the problem size is taken in Section 4.1.6. There, using slopes and constraint propagation, for every nonlinear constraint a pair of linear inequalities is constructed.

3.7 Convex Underestimation

Linear proramming is comparably easy and fast. However, linear relaxations are somewhat stiff, since they cannot grasp the curvature of the functions involved (enclosing a circle properly by linear inequalities needs a large number of linear inequalities depending on the required accuracy).

As we have seen in Theorem 2.14 for convex objective functions on convex feasible sets all local minima are already global, so local optimization of convex relaxations can be used for obtaining valid bounds.

This fact can be exploited by constructing a nonlinear convex relaxation of a problem instead of a linear relaxation. In order to do that, several approaches are possible. The problem can be rewritten in sparse form (as above) and convex and concave envelopes can be constructed for the operations. This can improve the enclosures, e.g. in the case of a circle.

However, a different approach was taken by Adjiman, Floudas, et al. in αBB. Every inequality constraint $F_i(x) \leq 0$ is separated into linear, convex, concave parts and a general nonlinear remainder term. Linear and convex parts are kept, the concave part is overestimated by a secant hyperplane, and the general nonlinear remainder is convexified by adding a nonpositive separable quadratic function: If F_i is a C^2 function on a finite box $\boldsymbol{x}$ we have

$$q(x) := F_i(x) + \tfrac{1}{2}(x - \underline{x})^T D(x - \overline{x})$$

for a nonnegative diagonal matrix D. This q underestimates F_i, and the amount of underestimation satisfies

$$|q(x) - F_i(x)| \leq \tfrac{1}{8}\mathrm{rad}\,\boldsymbol{x}^T D \mathrm{rad}\,\boldsymbol{x}.$$

If for the interval Hessian $G(\boldsymbol{x}) \subseteq \boldsymbol{G}$ for all $x \in \boldsymbol{x}$ and all symmetric matrices in $\boldsymbol{G} + D$ are positive definite (this is especially true, if all $G \in \boldsymbol{G} + D$ are regular and $\check{G} + D$ the midpoint matrix of $\boldsymbol{G} + D$ is positive definite), the function q is convex. The sum of linear and convex parts, secant hyperplane, and q is then guaranteed to be convex, as well. Applying that procedure to all constraints results in a convex relaxation which can be solved by local optimization.

More general techniques involving differences of convex functions (DC techniques) are treated in Horst & Tuy [89] and an overview is given in Tuy [223].

For more information on convexity and convexity detection see Neumaier [168].

Chapter 4
New Techniques

In this chapter I want to present a few techniques which are uncommon for global optimization algorithms or altogether new. The chapter is essentially split in two parts. Section 4.1 describes a new representation for factorable global optimization problems, which provides high flexibility and improved slopes and interval derivatives by automatic differentiation methods. In Section 4.2 we construct exclusion boxes around approximate solutions of systems of equations containing no other solution of the equation. This results also provides exclusion boxes around approximate Karush–John points of a global optimization problem, if applied to the Karush–John first order optimality conditions (see Section 2.2.4).

4.1 Directed Acyclic Graphs

This section, taken in part from SCHICHL & NEUMAIER [201], discusses a new representation technique for global optimization problems using **directed acyclic graphs** (DAGs). Traditionally, DAGs have been used in automatic differentiation (see GRIEWANK [24, 71]) and in the theory of parallel computing (see CHEKURI [32]). We will show that the DAG representation of a global optimization problem serves many purposes. In some global optimization algorithms (see KEARFOTT [112]) and constraint propagation engines (e.g., `ILOG solver`), the computational trees provided by the parsers of high-level programming language compilers (`FORTRAN 90`, `C++`) are used, in others the parsers of modeling languages like `AMPL` [58] or `GAMS` [29] provide the graph representation of the mathematical problem.

One of the strengths of the DAG concept is that it is suitable both for efficient evaluation and for performing constraint propagation (CP). The basics of constraint propagation on DAGs are outlined in Section 4.1.4.

The results of constraint propagation, especially the ranges of the inner nodes, can be used to improve the ranges of the standard evaluation methods for interval derivatives, and slopes. The principles are outlined in Section 4.1.5. For global optimization algorithms not only range estimates are relevant but also relaxations by models which are easier to solve. Section 4.1.6 describes methods for generating linear relaxations using the DAG representation.

4.1.1 Directed acyclic graphs

This section is devoted to the definition of the graphs used to represent the global optimization problems. Although we will use the term directed acyclic graph (DAG) throughout this book to reference the graph structure of the problem representation, the mathematical structure used is actually a bit more specialized. Here we will describe the basic properties of the graphs.

Definition 4.1. *A **directed multigraph** $\Gamma = (V, E, f)$ consists of a finite set of vertices (nodes) V, a finite set of edges E, and a mapping $f : E \to V \times V$. For every edge $e \in E$ we define the **source of** e as $s(e) := \mathrm{Pr}_1 \circ f(e)$ and the **target of** e as $t(e) := \mathrm{Pr}_2 \circ f(e)$. An edge e with $s(e) = t(e)$ is called a **loop**. Edges $e, e' \in E$ are called **multiple**, if $f(e) = f(e')$.*

*For every vertex $v \in V$ we define the set of **in-edges***

$$E_i(v) := \{e \in E \mid t(e) = v\}$$

*as the set of all edges, which have v as their target, and the set of **out-edges** analogously as the set*

$$E_o(v) := \{e \in E \mid s(e) = v\}$$

*of all edges with source v. The **indegree** of a vertex $v \in V$ is defined as the number of in-edges $\mathrm{indeg}(v) = |E_i(v)|$, and the the **outdegree** of v as the number of out-edges $\mathrm{outdeg}(v) = |E_o(v)|$.*

A vertex $v \in V$ with $\operatorname{indeg}(v) = 0$ *is called a* ***(local) source*** *of the graph, and a vertex $v \in V$ with* $\operatorname{outdeg}(v) = 0$ *is called a* ***(local) sink*** *of the graph.*

Remark. 4.2. *A directed tree is a special case of a directed multigraph. In a computational tree context, the local sources are usually called* leafs*, and the local sinks are denoted* roots*.*

Definition 4.3. *Let $\Gamma = (V, E, f)$ be a directed multigraph. A* ***directed path*** *from $v \in V$ to $v' \in V$ is a sequence $\{e_1, \dots, e_n\}$ of edges with $t(e_i) = s(e_{i+1})$ for $i = 1, \dots, n-1$, $v = s(e_1)$, and $v' = t(e_n)$. A directed path is called* ***closed*** *or a* ***cycle****, if $v = v'$. The multigraph Γ is called* ***acyclic*** *if it does not contain a cycle.*

Proposition 4.4. *A non-empty acyclic graph contains at least one source and at least one sink.*

Proof. Assume that the acyclic graph Γ contains no sink. Then every vertex has at least one out-edge. The graph is non-empty, so it contains a vertex v_0. The set $E_o(v_0)$ is non-empty, let $e_0 \in E_o(v_0)$, and set $v_1 = t(e_0)$. Since Γ is acyclic, $e_1 \neq e_0$. By induction, we construct a infinite sequence of distinct vertices of Γ, a contradiction to the finiteness of V. Hence, the graph contains a sink, and by symmetry also a source. $\square$

Definition 4.5. *A* ***directed multigraph with ordered edges (DMGoe)*** $\Gamma = (V, E, f, \leq)$ *is a quadruple such that (V, E, f) is a directed multigraph and $(E, \leq)$ is a linearly ordered set. As subsets of E, the in-edges $E_i(v)$ and out-edges $E_o(v)$ for every vertex become linearly ordered as well.*

We will represent the global optimization problems as directed acyclic computational multigraphs with ordered edges (in short DAG), where every vertex corresponds to an elementary operation and every edge represents the computational flow. For later use, we define the relationship between different vertices.

The reasons that we need multigraphs is the fact that expressions (e.g. x^x) can take the same input more than once. The ordering of the edges is primarily needed for non-commutative operators like division and power. However, we will see in Section 4.1.7 that this also has a consequence for certain commutative operations.

Definition 4.6. *Consider the directed acyclic multigraph $\Gamma = (V, E, f)$. For two vertices $v, v' \in V$ we say that v is a* ***parent*** *of v' if there exists an edge $e \in E$ with $s(e) = v'$ and $t(e) = v$, and then we call v' a* ***child*** *of v. Furthermore, v will be named an* ***ancestor*** *of v' if there is a directed path from v' to v, and v' is then a* ***descendant*** *of v.*

Now we have all the notions at hand that we will use to represent the optimization problems.

Proposition 4.7. *For every directed acyclic multigraph $\Gamma = (V, E, f)$ there is a linear order $\preceq$ on V such that for every vertex v and every ancestor v' of v we have $v \preceq v'$.*

Proof. The proof of this theorem is constructive. It proceeds by constructing a bijection $\varphi : V \to [0, |V|] \subseteq \mathbb{N}$.

```
k = 0

func rec_walk(vertex $v$)
  foreach $e \in E_o(v)$ do
    rec_walk($t(e)$)
  end
  if($\varphi(v)$ is undefined)
    $\varphi(v)$ := k
    k = k + 1
  end
end

foreach source $v$ do
  rec_walk($v$)
end
```

Now we define $v \preceq v' :\Longleftrightarrow \varphi(v) \le \varphi(v')$. □

4.1.2 Representing global optimization problems

In this section we will see how we represent a global optimization problem as a DAG. In Section 4.1.2 we will talk about simplifying the representation without changing the mathematical model. Later, in Section 4.1.2 we will show that DAGs can be used to transfer the mathematical problem to various different structures which are needed by specialized optimization and constraint satisfaction algorithms like ternary structure, semi-separable form, and the like. Also sparsity-issues can be tackled by the reinterpretation method described there.

Consider the factorable optimization problem

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) \in \boldsymbol{F}. \end{aligned} \tag{4.1}$$

Since it is factorable, the functions f and F can be expressed by sequences of arithmetic expressions and elementary functions. For every arithmetic operation $\circ$ or elementary function involved we introduce a vertex in the graph. Every constant and variable becomes a local source. If $f \circ g$ is part of one function, we introduce an edge from g to f. The results of f and F become local sinks nodes, of which the result of f is distinguished as the result of the objective function. So with every vertex we associate an arithmetic operation $\{+, *, /, \hat{}\}$ or elementary function $\{1/, \exp, \log, \sin, \cos, \dots\}$. For every edge $e \in E$ we call the vertex $t(e)$ the **result node** and the vertex $s(e)$ the **argument node**.

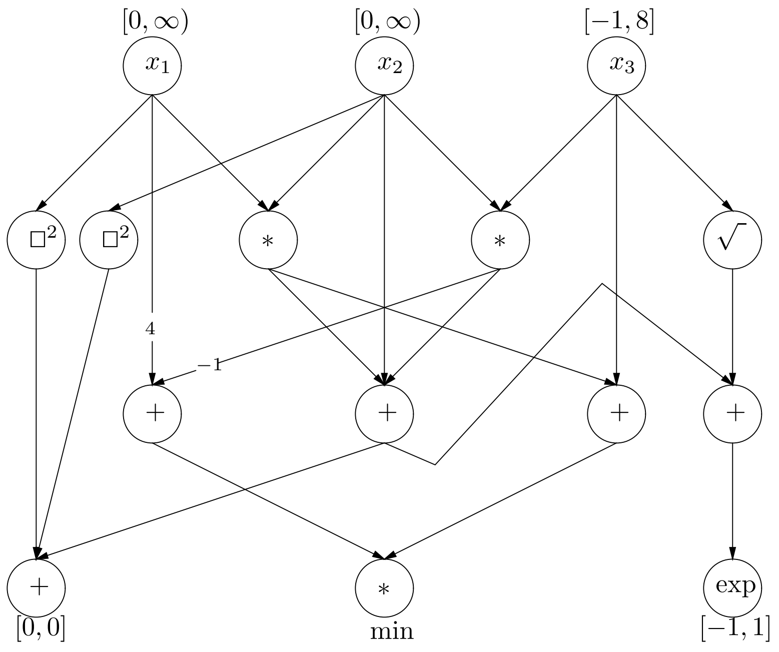


Figure 4.1. *DAG representation of problem 4.2*

When we draw DAG pictures, we write the operation in the interior of the circle representing the node, and mathematically we introduce a map $op : V \to \mathbb{O}$ to the set $\mathbb{O}$ of elementary operations. We also introduce a mapping $rg : V \to \mathbb{IR}$, the range map, which defines the feasible range of every vertex. In the pictures representing the graphs in this paper, we will write the result of the range map next to every vertex (and leave it out if $r(v) = (-\infty, \infty)$).

Consider for example the optimization problem

$$\begin{aligned} &\min\ (4x_1 - x_2x_3)(x_1x_2 + x_3) \\ &\text{s.t.}\ \ x_1^2 + x_2^2 + x_1x_2 + x_2x_3 + x_2 = 0 \\ &\qquad \exp(x_1x_2 + x_2x_3 + x_2 + \sqrt{x_3}) \in [-1, 1]. \end{aligned} \tag{4.2}$$

This defines the DAG depicted in Figure 4.1. Here, we have introduced further notation, the coefficient map $cf : E \to \mathbb{R}$. It multiplies the value of the source of e with $cf(e)$ before feeding it to the operation (or elementary function) $t(e)$. If the coefficient $cf(e)$ is different from 1, we write it over the edge in the picture. In some sense, the DAG in Figure 4.1 is optimally small, because it contains every subexpression of the functions f and F only once.

DAG Transformations - Simplification

If we start translating a function to a DAG, we introduce for every variable, every constant, and every operation involved a vertex and connect them by the necessary edges. The resulting DAG, however, usually is too big. Every subexpression of f which appears more than once will be represented by more than one node (e.g. v_1 and v_2). So, the subexpression will be recomputed too often in the evaluation routines, and during constraint propagation (see Section 4.1.4) the algorithms will not make use of the implicit equation $v_1 = v_2$.

Of course, variables usually appear more than once, and many algorithms for constraint propagation [97, 15, 195] use the principle that the variable nodes of identical variables can be identified, hereby reducing the size of the graph. However, this principle can be generalized.

Definition 4.8. *Two vertices v_1 and v_2 of the DAG $\Gamma = (V, E, f, \leq)$ are called **simply equivalent** if they represent the same operation or elementary function (i.e. $op(v_1) = op(v_2)$), and there is a monotone increasing bijective map $g : E_i(v_1) \to E_i(v_2)$ with the property $s(e) = s(g(e))$ for all $e \in E_i(v_1)$. If there are no distinct simply equivalent vertices in the DAG Γ, we call Γ a **reduced** DAG.*

The existence of the map g means nothing else than the fact that v_1 and v_2 represent the same expression. They are the same operation taking the same arguments in the same order. Therefore, any two simply equivalent vertices can be identified without changing the functions represented by Γ.

In particular, every DAG Γ can be transformed to an equivalent reduced DAG. We can start by identifying the equivalent leafs and continue to identify distinct simply equivalent nodes of Γ until all nodes are pairwise simply inequivalent. The resulting DAG Γ' is reduced. Note that this does **not** mean that the graph does not contain any *mathematically equivalent* subexpressions. This **only** implies that no *computationally equivalent* subexpressions exist.

These simple graph theoretic transformations can be complemented by additional mathematical transformations. These come in three categories:

Constant Evaluation/Propagation: If all children $v_1, \dots, v_k$ of a vertex v are leafs representing constants, it can be replaced by a leaf representing the constant which is the result of evaluating the operation $op(v)$ on the children: $v' := \mathrm{const}(op(v)(v_1, \dots, v_k))$. In a validated computation context, however, it has to made sure that no roundoff errors are introduced in this step.

Mathematical Equivalences: Typically, properties of elementary functions are used to change the DAG layout. E.g., the rule

$$\log(v_1 \dots v_k) = \log(v_1) + \cdots + \log(v_k)$$

replaces one log–node and one $*$–node by a $+$–node and a number of log–nodes (or vice versa).

Substitution: Equations of the form

$$-v_0 + v_1 + \cdots + v_k = 0$$

can be used to replace the node v_0 by $v_1 + \cdots + v_k$.

DAG Interpretation

One strength of the DAG representation is that the mathematical formulation of a problem can be transformed to an equivalent mathematical description which serves the specific needs of some optimization algorithms without having to change the DAG itself; just its **interpretation** is changed.

Consider again problem (4.2). The following problem is an equivalent formulation

$$\begin{aligned}
&\min\ x_{10} \\
&\text{s.t. } x_1^2 + x_2^2 + x_7 = 0 \\
&\quad\ \exp(x_7 + \sqrt{x_3}) \in [-1, 1] \\
&\quad\ x_2 x_3 - x_4 = 0 \\
&\quad\ x_6 + x_3 - x_5 = 0 \\
&\quad\ x_1 x_2 - x_6 = 0 \\
&\quad\ x_8 + x_2 - x_7 = 0 \\
&\quad\ x_4 + x_6 - x_8 = 0 \\
&\quad\ 4x_1 - x_4 - x_9 = 0 \\
&\quad\ x_9 x_5 - x_{10} = 0
\end{aligned} \tag{4.3}$$

of much higher dimension but with the property that the objective function is linear and that all constraints are **ternary**, i.e., involve at most three variables. This is the required problem formulation for a variety of CP algorithms.

Without changing the DAG we can get this representation just by changing the interpretation of the nodes. All intermediate nodes with more than one child and the objective function node are just regarded as variables, and an equation is added which connects the value of the variable with the value of the node as it is computed from its children. No change of the data structure is necessary.

Adding equations and changing the interpretation of intermediate nodes to variable nodes increases the dimension of the problem but also increases the sparsity. By carefully balancing the number of variables this method can be used, e.g., to optimize the sparsity structure of the Hessian of the Lagrangian.

4.1.3 Evaluation

There are several pieces of information which have to be computed for the functions involved in the definition of an optimization problem:

- function values at points,

- function ranges over boxes,
- gradients at points,
- interval gradients over boxes,
- slopes over boxes with fixed center,
- linear enclosures.

To illustrate the techniques, we will use throughout this and the following sections the simple example

$$\begin{aligned}&\min f(x_1, x_2, x_3) = (4x_1 - x_2x_3)(x_1x_2 + x_3)\\&\text{s.t. } \ x_1 \in [1, 2], \quad x_2 \in [3, 4], \quad x_3 \in [3, 4],\end{aligned} \tag{4.4}$$

whose DAG representation can be found in Figure 4.2.

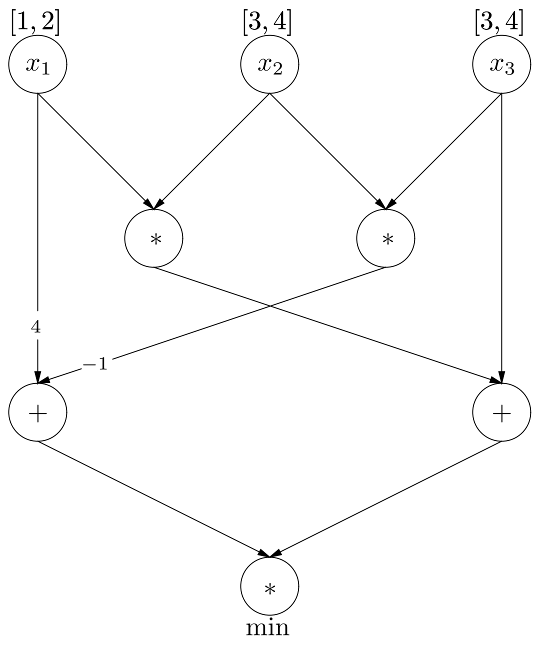


Figure 4.2. *Directed Acyclic Graph representation of* (4.4)

Forward Evaluation Scheme

The standard method of evaluating expressions works by feeding values to the local sources and propagating these values through the DAG in direction of the edges. This is the reason why this evaluation method is called **forward mode**.

Computing the function value $f(2, 4, 4)$ proceeds as depicted in Figure 4.3. Here, we have written the results for all nodes to the right of the circle representing them.

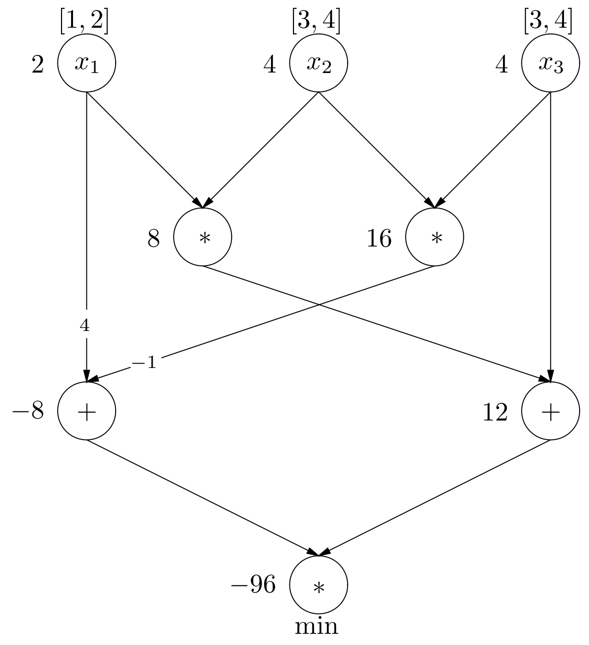


Figure 4.3. *Function evaluation for* (4.4)

In a completely analogous way we can compute a range estimate of f on the initial box $[1, 2] \times [3, 4] \times [3, 4]$. Instead of using real numbers we plug intervals into the sources and use interval arithmetic instead of real arithmetic and interval extensions of elementary functions instead of their real versions. Again, we show the process in Figure 4.4 by placing the ranges computed for the nodes next to them

Backward Evaluation Scheme

Calculating derivatives or slopes could be done by the forward mode as well but then we would need to propagate vectors through the graph, and at every node we would have to perform at least one full vector addition, so the effort to calculate a gradient would be proportional to the number of variables times the effort of calculating a function value.

However, it is well known from automatic differentiation that the number of operations can be reduced to be of the order of one function evaluation by reversing the direction of evaluation.

First, we recall the chain rule

$$\frac{\partial}{\partial x_i}(f \circ g)(x) = \sum_k \frac{\partial}{\partial x_k} f(g(x)) \cdot \frac{\partial}{\partial x_i} g(x).$$

So in a first step, during the computation of the function value, we construct a map $dm : E \to \mathbb{R}$ which associates with every edge the value of the partial derivative of the result node with respect to the corresponding argument node. Then we

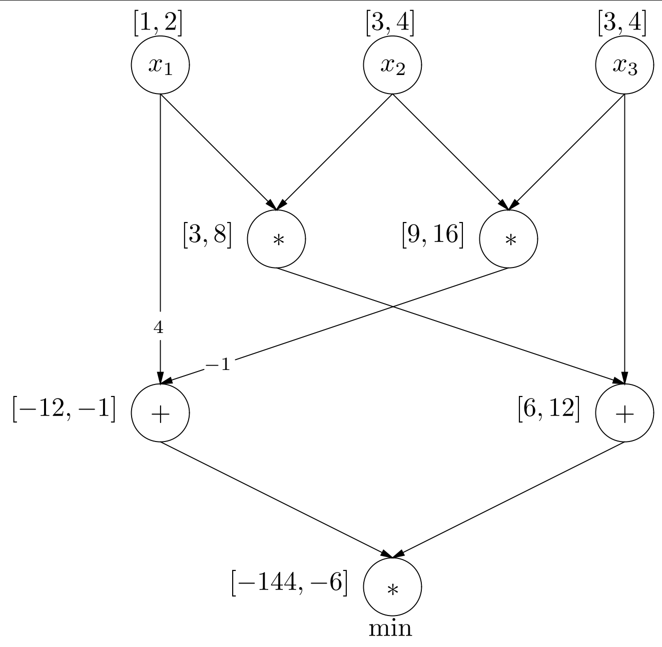


Figure 4.4. *Interval evaluation for* (4.4)

start at the local sinks and walk towards the sources in the opposite direction of the graph edges, multiplying by $dm(e)$ as we traverse e. When we reach the leaf representing variable x_i, we add the resulting product to the ith component of the gradient vector. The gradient at $(2, 4, 4)$ is calculated as in Figure 4.5. Here the values of dm are written next to the edges, and the results are next to the nodes. The components of the gradient can be found next to the sources of the graph. We have $\nabla f(2, 4, 4) = (16, -64, -56)$.

There is hardly any difference in computing the interval gradient of f over a given box $\boldsymbol{x}$. Since the chain rule looks exactly the same as for real gradients, the evaluation scheme is the same, as well. We only have to replace real arithmetic by interval arithmetic, and the map $idm : E \to \mathbb{IR}$ becomes interval valued. In Figure 4.6 we compute $\nabla f(\boldsymbol{x})$ for $\boldsymbol{x} = [1, 2] \times [3, 4] \times [3, 4]$.

Calculating slopes (see Section 3.4.2) works the same way, as was noticed by Bliek [24] for computational trees. Like there, we can use the backward mode to compute the slopes on the DAG. The arithmetic operations and the elementary functions look like depicted in Figure 4.7. There z_f denotes the center of f, and s_f the slope of f.

We see from the pictures that for the elementary functions, the slopes $\varphi[z, \boldsymbol{x}]$ have to be computed. We know from Section 3.4.2 that for convex and concave functions the optimal slope can be easily computed, and for the other functions, the case is

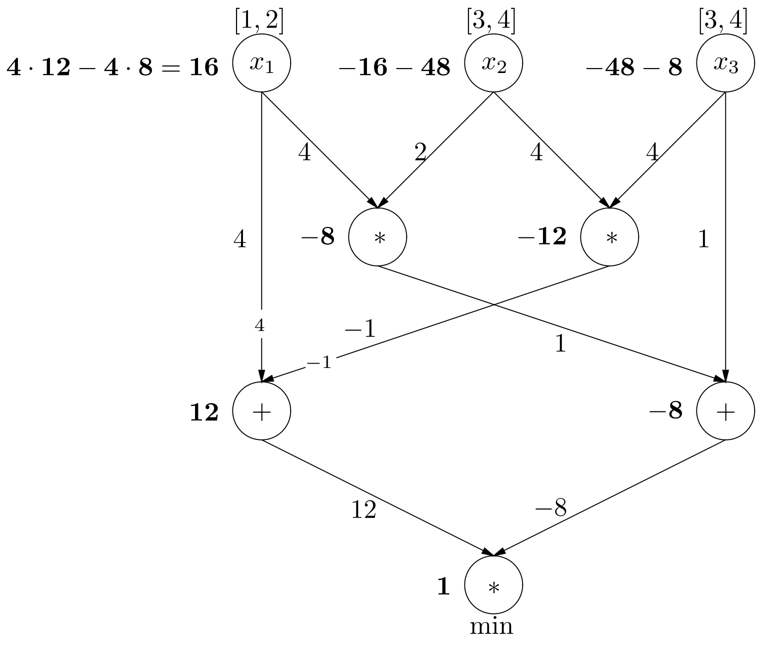


Figure 4.5. *Gradient evaluation for* (4.4)

more difficult, but we always have

$$\varphi[z, \boldsymbol{x}] \subseteq \varphi'(\boldsymbol{x}).$$

To compute general slopes, we first compute the values of the centers in forward mode, which is an ordinary (interval) function evaluation. Then we change the map *dm* to *slm* : $E \to \mathbb{IR}$ storing the slope of the result node with respect to the argument node during the forward pass, and then we use interval arithmetic to compute the slope in backward mode. This can be seen in Figure 4.8, where we keep at each node the centers and the slopes separated by a comma.

The result $f[z, \boldsymbol{x}] = ([-8, 24], [-64, -34], [-56, -32])$ is clearly slimmer than the interval derivative $f'(\boldsymbol{x}) = ([-24, 45], [-72, -19], [-60, -19])$ as it was expected, since slopes provide better enclosures than interval derivatives.

4.1.4 Constraint Propagation on DAGs

As already mentioned, one strength of the DAG concept for global optimization is that knowledge of feasible points and the constraints can be used to narrow the possible ranges of the variables, cf. [15, 85, 195].

If we have a feasible point x_{best} with function value f_{best} we can introduce the new constraint $f(x) \le f_{\text{best}}$ without changing the solution of the optimization problem (4.1). Then the ranges of the nodes can be propagated through the DAG, refining the range map $rg : V \to \mathbb{IR}$ in every step of the constraint propagation (i.e.

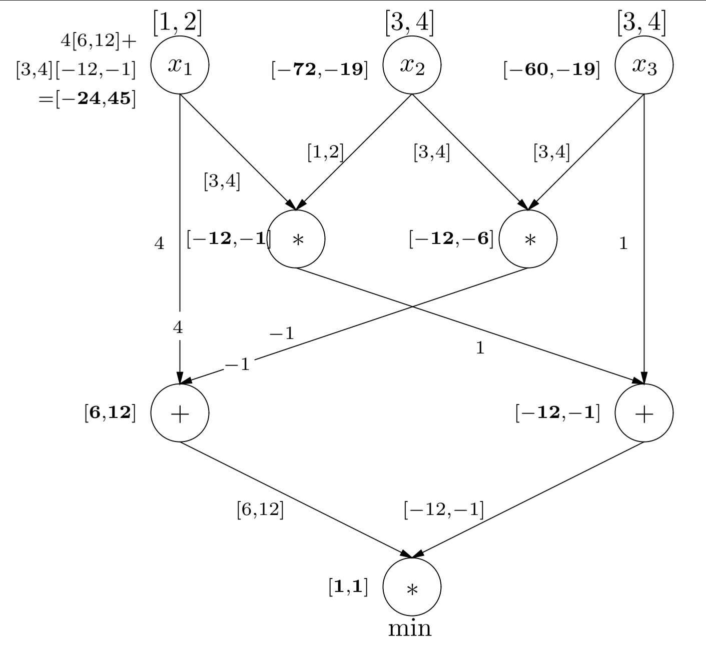
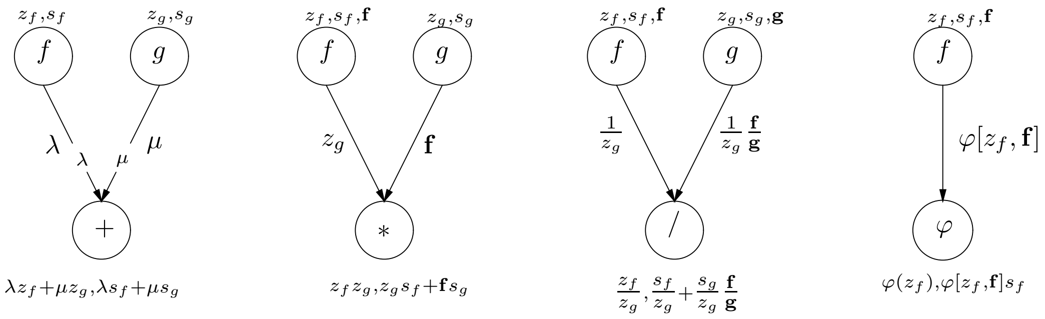


Figure 4.6. *Interval gradient evaluation for* (4.4)



z_f, s_f
z_g, s_g
λ
μ
$\lambda z_f + \mu z_g, \lambda s_f + \mu s_g$
$z_f, s_f, \mathbf{f}$
z_g, s_g
z_g
$\mathbf{f}$
$z_f z_g, z_g s_f + \mathbf{f} s_g$
$z_f, s_f, \mathbf{f}$
$z_g, s_g, \mathbf{g}$
$\frac{1}{z_g}$
$\frac{1}{z_g}\frac{\mathbf{f}}{\mathbf{g}}$
$\frac{z_f}{z_g}, \frac{s_f}{z_g} + \frac{s_g}{z_g}\frac{\mathbf{f}}{\mathbf{g}}$
$z_f, s_f, \mathbf{f}$
$\varphi[z_f, \mathbf{f}]$
$\varphi(z_f), \varphi[z_f, \mathbf{f}] s_f$

Figure 4.7. *Slopes for elementary operations*

$rg^{(n+1)}(v) \subseteq rg^{(n)}(v)$ for all v, if $rg^{(n)}$ denotes the range map at step n). We stop when the reductions become too small.

Constraint propagation has two directions, forward and backward. For the elementary functions the propagation steps are as follows.

$\mathbf{h} = \lambda\mathbf{f} + \mu\mathbf{g}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (\lambda \boldsymbol{f}^{(n+1)} + \mu \boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := \tfrac{1}{\lambda}(\boldsymbol{h}^{(n+1)} - \mu \boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := \tfrac{1}{\mu}(\boldsymbol{h}^{(n+1)} - \lambda \boldsymbol{f}^{(n)}) \cap \boldsymbol{g}^{(n)}.$$

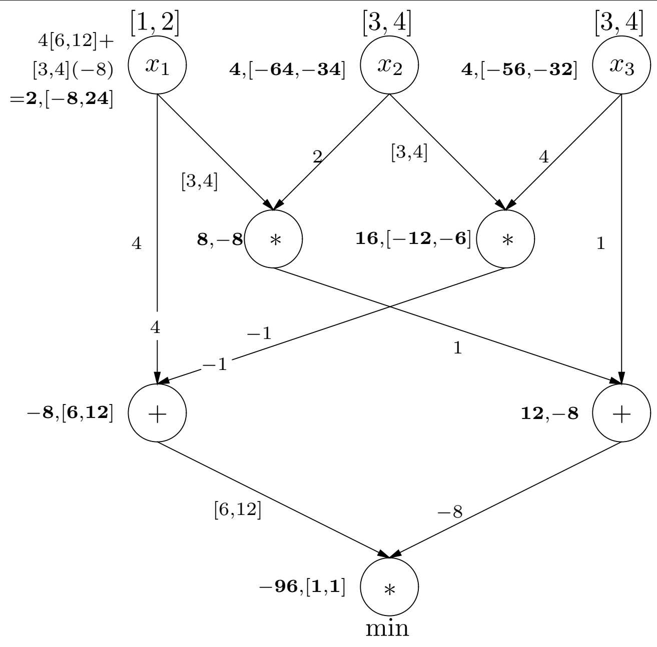


Figure 4.8. *Slope evaluation for* (4.4)

$\mathbf{h} = \mathbf{fg}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (\boldsymbol{f}^{(n+1)}\boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := (\boldsymbol{h}^{(n+1)}/\boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := (\boldsymbol{h}^{(n+1)}/\boldsymbol{f}^{(n)}) \cap \boldsymbol{g}^{(n)}.$$

$\mathbf{h} = \mathbf{f}/\mathbf{g}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (\boldsymbol{f}^{(n+1)}/\boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := (\boldsymbol{h}^{(n+1)}\boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := (\boldsymbol{f}^{(n)}/\boldsymbol{h}^{(n+1)}) \cap \boldsymbol{g}^{(n)}.$$

$\mathbf{h} = \varphi(\mathbf{f})$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := \varphi(\boldsymbol{f}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := \varphi^{-1}(\boldsymbol{h}^{(n+1)}) \cap \boldsymbol{f}^{(n)}.$$

Note that for the DAG representation we refine the range map for **all** nodes not only for the leaf nodes. This is an important step because that will help us in Section 4.1.5 to improve the ranges of interval derivatives, slopes, interval Hessians, and second order slopes.

In Figure 4.9 we show the result of constraint propagation to our example, if we use the function value -96 of the feasible point $(2, 4, 4)$ to introduce the constraint $f(x) \leq -96$. Note that the ranges of the variable nodes do not change, so the traditional method of calculating interval related results is not improved. The new ranges are printed in the picture in bold face.

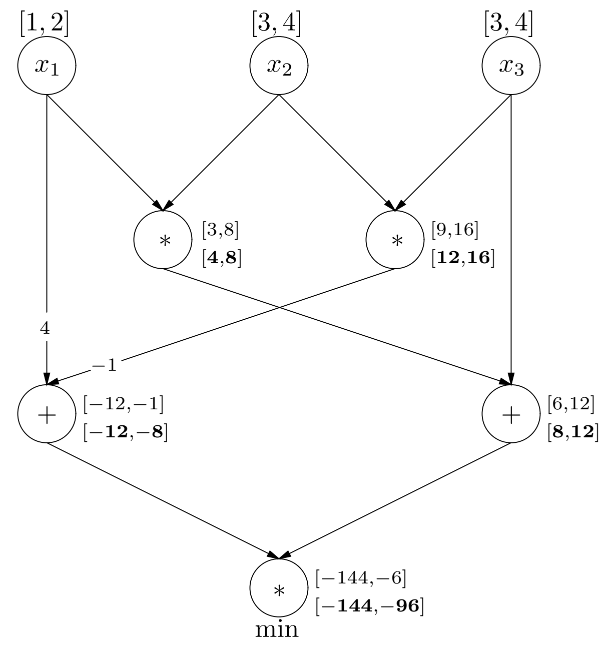


Figure 4.9. *Constraint propagation for* (4.4)

4.1.5 Combining CP and Evaluation

In this section we will use the range map $rg : V \to \mathbb{IR}$ improved by constraint propagation to recompute the interval derivative, the slope, and the interval Hessians. This improves the ranges, in some examples tested the improvement was several orders of magnitude.

Figure 4.10 contains the result of the interval gradient after constraint propagation, and in Figure 4.11 we recompute the slope.

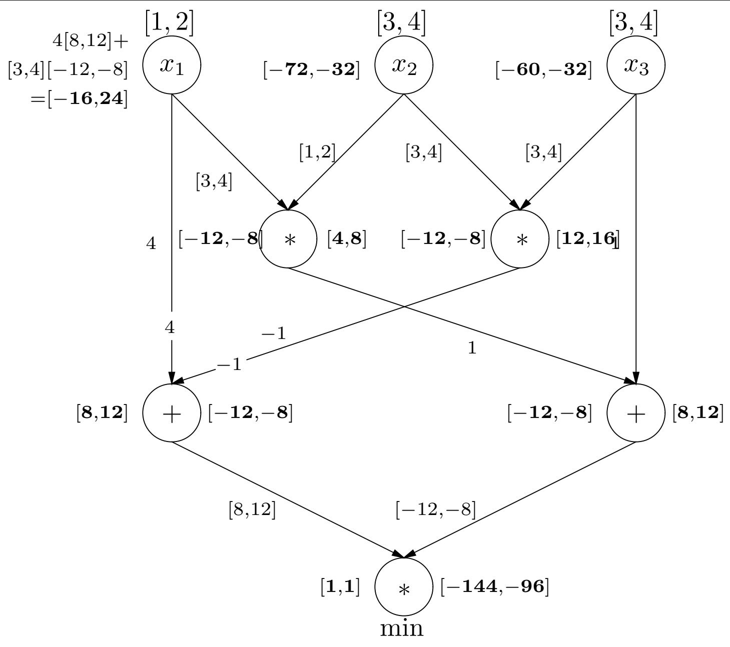


Figure 4.10. *Interval gradient evaluation for* (4.4) *after constraint propagation*

Both results are clearly an improvement over what we had before:

$$f'(\boldsymbol{x}) \subseteq \begin{pmatrix} [-16,24] \\ [-72,-32] \\ [-60,-32] \end{pmatrix} \subsetneq \begin{pmatrix} [-24,45] \\ [-72,-19] \\ [-60,-19] \end{pmatrix}, \quad f[z,\boldsymbol{x}] \subseteq \begin{pmatrix} [0,24] \\ [-64,-48] \\ [-56,-32] \end{pmatrix} \subsetneq \begin{pmatrix} [-8,24] \\ [-64,-34] \\ [-56,-32] \end{pmatrix}.$$

4.1.6 Slopes and linear enclosures

The linear approximation (3.3) of a function f provided by slopes can be used to construct an enclosure of f by linear functions. This in turn can be used to construct a linear relaxation of the original problem.

Proposition 4.9. *Let $s := f[z,\boldsymbol{x}]$ be a slope of the function $f : \mathbb{R}^n \to \mathbb{R}$. If $z \in \boldsymbol{x}$ then the function*

$$\underline{f}(x) = \underline{f} + \sum \overline{s}_i(\underline{x}_i - z_i) + \frac{\underline{s}_i(\overline{x}_i - z_i) - \overline{s}_i(\underline{x}_i - z_i)}{\overline{x}_i - \underline{x}_i}(x_i - \underline{x}_i)$$

is a linear function which underestimates f on $\boldsymbol{x}$, i.e.,

$$\underline{f}(x) \le f(x) \quad \textit{for all } x \in \boldsymbol{x},$$

and the function

$$\overline{f}(x) = \overline{f} + \sum \underline{s}_i(\underline{x}_i - z_i) + \frac{\overline{s}_i(\overline{x}_i - z_i) - \underline{s}_i(\underline{x}_i - z_i)}{\overline{x}_i - \underline{x}_i}(x_i - \underline{x}_i)$$

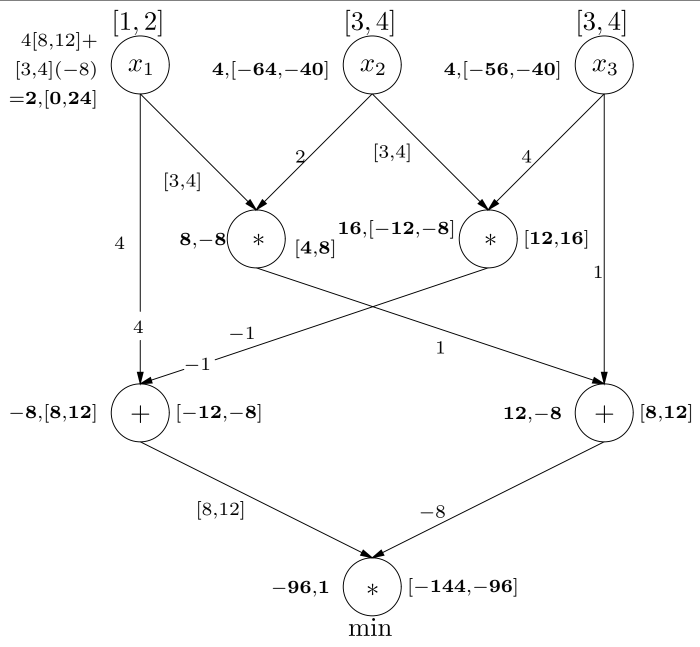


Figure 4.11. *Slope evaluation for* (4.4) *after constraint propagation*

is a linear overestimating function for f over $\boldsymbol{x}$.

Proof. A linear function is separable, so the over- and underestimation can be performed componentwise. So, the estimates can be reduced to a series of one dimensional problems, and for those the proof is easy. □

For problem (4.1) we have to consider the constraints componentwise. For every component $F_j(x) \in \boldsymbol{F}_j$ the constraints $\underline{F}_j(x) \le \overline{F}_j$ and $\overline{F}_j(x) \ge \underline{F}_j$ are valid linear constraints. They can be added as redundant constraints to the problem without affecting the solution.

Alternatively, one could also compute the underestimating function $\underline{l}$ for the objective function f. Then the linear program

$$\begin{aligned} &\min\ \underline{f}(x) \\ &\text{s.t.}\ \ \underline{F}(x) \le \overline{F} \\ &\qquad \overline{f}(x) \ge \underline{F} \\ &\qquad x \in \boldsymbol{x}, \end{aligned}$$

where $\underline{F}(x)$ denotes the vector of all underestimating functions $\underline{F}_j$ for all components F_j, is a linear relaxation of (4.1).

For the example given by (4.2), we already computed the slope for center $(2, 4, 4)$ in Section 4.1.5. Calculating a linear underestimating function for the objective, as above, leads to the constraint

$$-24(x_1 - 2) - 48(x_2 - 4) - 32(x_3 - 4) \le 0.$$

Performing constraint propagation again on the problem with this additional redundant constraint leads to the domain reduction $x_{2,3} \in [3.4, 4]$. With previously known techniques but without (expensive) higher order consistency, such a reduction would have required a split of the box.

Alternatively, it is possible to construct linear enclosures of the form

$$f(x) \in \boldsymbol{f} + s(x - z), \quad \text{for } x \in \boldsymbol{x},$$

with thin slope $s \in \mathbb{R}^n$ and thick constant term. This approach corresponds to first order Taylor arithmetic as, e.g., presented in [20, 21, 165]. Since linear Taylor expression also obey a chain rule similar to slopes, these enclosures can be computed by backward evaluation with little effort quite similar to “thick” slopes. Kolev [123] showed that propagating them in forward mode leads to better enclosures; however, the effort for computing in forward mode is n times higher.

4.1.7 Implementation Issues

Multiplication and Division

As has been mentioned in Section 3.4.2, slopes in dimensions greater than one are usually not unique. Two elementary operations, multiplication and division, therefore provide us with a choice for implementation.

All possible slopes for multiplication are

$$x_1 x_2 \in z_1 z_2 + \begin{pmatrix} \lambda \boldsymbol{x}_2 + (1-\lambda) z_2 \\ \lambda z_1 + (1-\lambda)\boldsymbol{x}_1 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x}_1 - z_1 \\ \boldsymbol{x}_2 - z_2 \end{pmatrix}$$

for some $\lambda \in \mathbb{R}$ (possibly dependent on the arguments), and for division they are

$$\frac{x_1}{x_2} \in \frac{z_1}{z_2} + \begin{pmatrix} \dfrac{\lambda}{z_2} + \dfrac{1-\lambda}{\boldsymbol{x}_2} \\ -\dfrac{\lambda}{z_2}\dfrac{\boldsymbol{x}_1}{\boldsymbol{x}_2} - \dfrac{1-\lambda}{\boldsymbol{x}_2}\dfrac{z_1}{z_2} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x}_1 - z_1 \\ \boldsymbol{x}_2 - z_2 \end{pmatrix}.$$

The best choice for division is $\lambda = 1$, because we can use the term $\frac{\boldsymbol{x}_1}{\boldsymbol{x}_2}$ after constraint propagation, which is the range enclosure of the result node, for which the slope is being computed, and in addition there is no subdistributivity problem during slope backward evaluation. So the proper choice for division is

$$s_{/} = \frac{1}{z_2} \begin{pmatrix} 1 \\ -\dfrac{\boldsymbol{x}_1}{\boldsymbol{x}_2} \end{pmatrix}.$$

For multiplication we can choose λ such that it minimizes the width of the resulting range. A short computation shows that the minimal width is produced for

$$\lambda = \begin{cases} 0 & \text{if } \operatorname{rad}(\boldsymbol{x}_1)|z_2| \le \operatorname{rad}(\boldsymbol{x}_2)|z_1|, \\ 1 & \text{otherwise.} \end{cases}$$

To avoid a case distinction in computing products, it is advisable to find a good heuristics. Considering the Horner scheme for polynomial evaluation gives the following hint: Sort the product by ascending complexity of the factors (i.e., roughly, by increasing overestimation). Then set $\lambda = 0$, hence choose the slope

$$s_* = \begin{pmatrix} z_2 \\ \boldsymbol{x}_1 \end{pmatrix}.$$

Rounding errors

Since enclosures of the form

$$f(x) \in f(z) + f[z, \boldsymbol{x}](\boldsymbol{x} - z),$$

are computed numerically, the direct evaluation of the thin term $f(z)$ generally does not produce guaranteed enclosures. Hence, it is important to take care for the rounding errors, in order to avoid the occasional loss of solutions in a branch and bound scheme. There are two possible approaches.

The first possibility is to change all calculations involving the center into interval operations, providing a linear interval enclosure

$$f(x) \in \boldsymbol{f}(\boldsymbol{z}) + f[\boldsymbol{z}, \boldsymbol{x}](\boldsymbol{x} - \boldsymbol{z})$$

with generally thick center $\boldsymbol{z}$. This needs slopes of the form $f[\boldsymbol{z}, \boldsymbol{x}]$ with $\boldsymbol{z} \subseteq \boldsymbol{x}$ for all elementary operations.

The second possibility is to allow approximate point evaluations at the centers and elementary slopes with point centers $f[z, \boldsymbol{x}]$, but to take care of the rounding errors in computing $f(z)$ during propagation, by adapting the chain rule appropriately. If

$$\begin{aligned} f(x) &\in \boldsymbol{f} + f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f), \quad f(z_f) \in \boldsymbol{f},\ x \in \boldsymbol{x} \\ g(y) &\in \boldsymbol{g} + g[z_g, \boldsymbol{y}](\boldsymbol{y} - z_g), \quad g(z_g) \in \boldsymbol{g},\ y \in \boldsymbol{y}, \end{aligned}$$

then, for arbitrary $z_g \approx f(z_f)$,

$$\begin{aligned} g(f(x)) &\in \boldsymbol{g} + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} + f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f) - z_g) \\ &\subseteq \boldsymbol{g} + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g) + g[z_g, f(\boldsymbol{x})]f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f). \end{aligned}$$

The remaining decision is what to compute in forward, and what in backward mode. Taking a third component provides the important hint:

$$h(t) \in \boldsymbol{h} + h[z_h, \boldsymbol{t}](\boldsymbol{t} - z_h),$$

and we find

$$\begin{aligned} h(g(f(x))) \in \boldsymbol{h} &+ h[z_h, g(f(\boldsymbol{x}))](\boldsymbol{g} - z_h + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g)) \\ &+ h[z_h, g(f(\boldsymbol{x}))]g[z_g, f(\boldsymbol{x})]f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f) \end{aligned}$$

if the center term is computed in forward mode. If it is computed backward, the term is

$$h[z_h, g(f(\boldsymbol{x}))](g(f(\boldsymbol{x})) - z_h) + h[z_h, g(f(\boldsymbol{x}))]g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g).$$

Because of subdistributivity, this is a worse (or identical) enclosure of the center. Therefore, computing the center in forward mode gives generally tighter results.

4.2 Exclusion Boxes

Branch and bound methods for finding all zeros of a nonlinear system of equations in a box (see Kearfott [112] or vanHentenryck et al. [85]) frequently have the difficulty that subboxes containing no solution cannot be easily eliminated if there is a nearby zero outside the box. This has the effect that near each zero, many small boxes are created by repeated splitting, whose processing may dominate the total work spent on the global search. This section primarily contains material from Schichl & Neumaier [200].

The cluster effect, see Section 3.4.6 in the reasons for the occurrence of this so-called cluster effect, and how to reduce the cluster effect by defining exclusion regions around each zero found, that are guaranteed to contain no other zero and hence can safely be discarded. Such exclusion boxes (possibly first used by Jansson [101]) are the basis for the backboxing strategy by van Iwaarden [99] (see also Kearfott [113, 115]) that eliminates the cluster effect near well-conditioned zeros.

Exclusion regions are traditionally constructed using uniqueness tests based on the Krawczyk operator (see, e.g., Neumaier [160, Chapter 5]) or the Kantorovich theorem (see, e.g., Ortega & Rheinboldt [176, Theorem 12.6.1]); both provide existence and uniqueness regions for zeros of systems of equations. Shen & Neumaier [208] proved that the Krawczyk operator with slopes always provides an existence region which is at least as large as that computed by Kantorovich's theorem. Deuflhard & Heindl [44] proved an affine invariant version of the Kantorovich theorem.

In Section 3.4.7, these results are reviewed, together with recent work on improved preconditioning by Hansen [74] and on Taylor models by Berz & Hoefkens [19] that is related to our present work. In Sections 4.2.1–4.2.3, we discuss componentwise and affine invariant existence, uniqueness, and non-existence regions given a zero or any other point of the search region. They arise from a more detailed analysis of the properties of the Krawczyk operator with slopes used in Shen & Neumaier [208].

Numerical examples given in Section 4.2.4 show that the refinements introduced in this paper significantly enlarge the sizes of the exclusion regions.

We consider the nonlinear system of equations

$$F(x) = 0, \tag{4.5}$$

where $F : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is twice continuously differentiable in a convex domain D. (For some results, weaker conditions suffice; it will be clear from the arguments used that continuity and the existence of the quantities in the hypothesis of the theorems are sufficient.)

As in Section 3.4.4, we construct the second order slope matrices $F_k[z, z', x]$ for F

$$F(x) = F(z) + F[z, z'](x - z) + \sum (x_k - z'_k) F_k[z, z', x](x - z). \tag{4.6}$$

and for $z = z'$

$$F(x) = F(z) + F'(z)(x - z) + \sum (x_k - z_k) F_k[z, z, x](x - z). \tag{4.7}$$

Throughout this Section we shall make the following assumption, without mentioning it explicitly.

Assumption A. The point z and the convex subset X lie in the domain of definition of F. The center $z \in X$, and the second order slope (4.7) are fixed. Moreover, for a fixed preconditioning matrix $C \in \mathbb{R}^{m\times n}$, the componentwise bounds

$$\begin{aligned} \overline{b} &\geq |CF(z)| \geq \underline{b}, \\ B_0 &\geq |CF'(z) - I|, \\ B_0' &\geq |CF'(z)|, \\ B_k(x) &\geq |CF_k[z,z,x]| \quad (k = 1, \dots, n) \end{aligned} \tag{4.8}$$

are valid for all $x \in X$.

Example. 4.10. *We consider the system of equations*

$$\begin{aligned} x_1^2 + x_2^2 &= 25, \\ x_1 x_2 &= 12. \end{aligned} \tag{4.9}$$

The system has the form (4.5) *with*

$$F(x) = \begin{pmatrix} x_1^2 + x_2^2 - 25 \\ x_1 x_2 - 12 \end{pmatrix}. \tag{4.10}$$

With respect to the center $z = \binom{3}{4}$, *we have*

$$F(x) - F(z) = \begin{pmatrix} x_1^2 - 3^2 + x_2^2 - 4^2 \\ x_1 x_2 - 3 \cdot 4 \end{pmatrix} = \begin{pmatrix} (x_1+3)(x_1-3) + (x_2+4)(x_2-4) \\ x_2(x_1-3) + 3(x_2-4) \end{pmatrix},$$

so that we can take

$$F[z,x] = \begin{pmatrix} x_1 + 3 & x_2 + 4 \\ x_2 & 3 \end{pmatrix}$$

as a slope. (Note that other choices would be possible.) The interval slope $F[z,\boldsymbol{x}]$ *in the box* $\boldsymbol{x} = [2,4] \times [3,5]$ *is then*

$$F[z,x] = \begin{pmatrix} [5,7] & [7,9] \\ [3,5] & 3 \end{pmatrix}.$$

The slope can be put in form (4.7) *with*

$$F'(z) = \begin{pmatrix} 6 & 8 \\ 4 & 3 \end{pmatrix}, \quad F_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad F_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

and we obtain

$$B_1 = \frac{1}{14}\begin{pmatrix} 3 & 0 \\ 4 & 0 \end{pmatrix}, \quad B_2 = \frac{1}{14}\begin{pmatrix} 8 & 3 \\ 6 & 4 \end{pmatrix}.$$

Since we calculated without rounding errors and z happens to be a zero of F, both B_0 and $\overline{b}$ vanish.

4.2.1 Componentwise exclusion regions close to a zero

Suppose that x^* is a solution of the nonlinear system of equations (4.5). We want to find an **exclusion region** around x^* with the property that in the interior of this region x^* is the only solution of (4.5). Such an exclusion region need not be further explored in a branch and bound method for finding all solutions of (4.5); hence the name.

In this section we take an approximate zero z of F, and we choose C to be an approximation of $F'(z)^{-1}$. Suitable candidates for z can easily be found within a branch and bound algorithm by trying Newton steps from the midpoint of each box, iterating while x^ℓ remains in a somewhat enlarged box and either $\|x^{\ell+1} - x^\ell\|$ or $\|F(x^\ell)\|$ decreases by a factor of say 1.5 below the best previous value in the iteration. This works locally well even at nearly singular zeros and gives a convenient stop in case no nearby solution exists.

Proposition 4.11. *For every solution $x \in X$ of* (4.5)*, the deviation*

$$s := |x - z|$$

satisfies

$$0 \le s \le \Big(B_0 + \sum s_k B_k(x)\Big)s + \overline{b}. \tag{4.11}$$

Proof. By (3.3) we have $F[z,x](x-z) = F(x) - F(z) = -F(z)$, because x is a zero. Hence, using (4.7), we compute

$$\begin{aligned} -(x-z) &= -(x-z) + C(F[z,x](x-z) + F(z) + F'(z)(x-z) - F'(z)(x-z)) \\ &= C(F[z,x] - F'(z))(x-z) + (CF'(z) - I)(x-z) + CF(z) \\ &= \Big(CF'(z) - I + \sum (x_k - z_k)CF_k[z,z,x]\Big)(x-z) + CF(z). \end{aligned}$$

Now we take absolute values, use (4.8), and get

$$\begin{aligned} s = |x-z| &\le \Big(|CF'(z) - I| + \sum |x_k - z_k|\,|CF_k[z,z,x]|\Big)|x-z| + |CF(z)| \\ &\le \Big(B_0 + \sum s_k B_k(x)\Big)s + \overline{b}. \end{aligned}$$

□

Using this result we can give a first criterion for existence regions.

Theorem 4.12. *Let $0 < u \in \mathbb{R}^n$ be such that*

$$\Big(B_0 + \sum u_k \overline{B}_k\Big)u + \overline{b} \le u \tag{4.12}$$

with $B_k(x) \le \overline{B}_k$ for all $x \in M_u$, where

$$M_u := \{x \mid |x - z| \le u\} \subseteq X. \tag{4.13}$$

Then (4.5) *has a solution* $x \in M_u$.

Proof. For arbitrary x in the domain of definition of F we define

$$K(x) := x - CF(x).$$

Now take any $x \in M_u$. We get

$$\begin{aligned} K(x) &= x - CF(x) = z - CF(z) - (CF[z,x] - I)(x-z) = \\ &= z - CF(z) - \Big(C\Big(F'(z) + \sum F_k[z,z,x](x_k - z_k)\Big) - I\Big)(x-z), \end{aligned}$$

hence

$$K(x) = z - CF(z) - \Big(CF'(z) - I + \sum CF_k[z,z,x](x_k - z_k)\Big)(x-z). \tag{4.14}$$

Taking absolute values we find

$$\begin{aligned} |K(x) - z| &= \Big|-CF(z) - \Big(CF'(z) - I + \sum CF_k[z,z,x](x_k - z_k)\Big)(x-z)\Big| \le \\ &\le |CF(z)| + \Big(|CF'(z) - I| + \sum |CF_k[z,z,x]|\,|x_k - z_k|\Big)|x-z| \le \\ &\le \overline{b} + \Big(B_0 + \sum u_k \overline{B}_k\Big)u. \end{aligned} \tag{4.15}$$

Now assume (4.12). Then (4.15) gives

$$|K(x) - z| \le u,$$

which implies by Theorem 3.3 that there exists a solution of (4.5) which lies in M_u. ☐

Note that (4.12) implies $B_0 u \le u$, thus that the spectral radius $\rho(B_0) \le 1$. In the applications, we can make both B_0 and $\overline{b}$ very small by choosing z as an approximate zero, and C as an approximate inverse of $F'(z)$.

Now the only thing that remains is the construction of a suitable vector u for Theorem 4.12.

Theorem 4.13. *Let* $S \subseteq X$ *be any set containing* z, *and take*

$$\overline{B}_k \ge B_k(x) \quad \textit{for all } x \in S. \tag{4.16}$$

For $0 < v \in \mathbb{R}^n$, *set*

$$w := (I - B_0)v, \quad a := \sum v_k \overline{B}_k v. \tag{4.17}$$

We suppose that

$$D_j = w_j^2 - 4a_j \overline{b}_j > 0 \tag{4.18}$$

for all $j = 1, \ldots, n$, *and define*

$$\lambda_j^e := \frac{w_j + \sqrt{D_j}}{2a_j}, \quad \lambda_j^i := \frac{\overline{b}_j}{a_j \lambda_j^e}, \tag{4.19}$$

$$\lambda^e := \min_{j=1,\dots,n} \lambda_j^e, \quad \lambda^i := \max_{j=1,\dots,n} \lambda_j^i. \tag{4.20}$$

If $\lambda^e > \lambda^i$ then there is at least one zero x^* of (4.5) in the (inclusion) region

$$R^i := [z - \lambda^i v, z + \lambda^i v] \cap S. \tag{4.21}$$

The zeros in this region are the only zeros of F in the interior *of the (exclusion) region*

$$R^e := [z - \lambda^e v, z + \lambda^e v] \cap S. \tag{4.22}$$

Proof. Let $0 < v \in \mathbb{R}^n$ be arbitrary, and set $u = \lambda v$. We check for which λ the vector u satisfies property (4.12) of Theorem 4.12. The requirement

$$\begin{aligned} \lambda v &\geq \Big(B_0 + \sum u_k \overline{B}_k\Big) u + \overline{b} = \Big(B_0 + \sum \lambda v_k \overline{B}_k\Big)\lambda v + \overline{b} \\ &= \overline{b} + \lambda B_0 v + \lambda^2 \sum v_k \overline{B}_k v = \overline{b} + \lambda(v - w) + \lambda^2 a \end{aligned}$$

leads to the sufficient condition $\lambda^2 a - \lambda w + \overline{b} \leq 0$. The jth component of this inequality requires that λ lies between the solutions of the quadratic equation $\lambda^2 a_j - \lambda w_j + \overline{b}_j = 0$, which are λ_j^i and λ_j^e. Hence, for every $\lambda \in [\lambda^i, \lambda^e]$ (this interval is nonempty by assumption), the vector u satisfies (4.12).

Now assume that x is a solution of (4.5) in $\operatorname{int}(R^e) \setminus R^i$. Let λ be minimal with $|x - z| \leq \lambda v$. By construction, $\lambda^i < \lambda < \lambda^e$. By the properties of the Krawczyk operator, we know that $x = K(z, x)$, hence

$$\begin{aligned} |x - z| &\leq |CF(z)| + \Big(|CF'(z) - I| + \sum |CF_k[z, z, x]|\,|x_k - z_k|\Big)|x - z| \\ &\leq \overline{b} + \lambda B_0 v + \lambda^2 \sum v_k \overline{B}_k v < \lambda v, \end{aligned} \tag{4.23}$$

since $\lambda > \lambda^i$. But this contradicts the minimality of λ. So there are indeed no solutions of (4.5) in $\operatorname{int}(R^e) \setminus R^i$. ☐

This is a componentwise analogue of the Kantorovich theorem. We show in Example 4.19 that it is best possible in some cases.

We observe that the inclusion region from Theorem 4.13 can usually be further improved by noting that $x^* = K(z, x^*)$ and (4.14) imply

$$x^* \in K(z, \boldsymbol{x}^i) = z - CF(z) - \Big(CF'(z) - I + \sum CF_k[z, z, \boldsymbol{x}^i](\boldsymbol{x}_k^i - z_k)\Big)(\boldsymbol{x}^i - z) \subset \operatorname{int}(\boldsymbol{x}^i).$$

An important special case is when $F(x)$ is quadratic in x. For such a function $F[z, x]$ is linear in x, and therefore all $F_k[z, z, x]$ are constant in x. This, in turn, means that $B_k(x) = B_k$ is constant as well. So we can set $\overline{B}_k = B_k$, and the estimate (4.16) becomes valid everywhere.

Corollary 4.14. *Let F be a quadratic function. For arbitrary $0 < v \in \mathbb{R}^n$ define*

$$w := (I - B_0)v, \quad a := \sum v_k B_k v. \tag{4.24}$$

We suppose that

$$D_j = w_j^2 - 4a_j\overline{b}_j > 0 \tag{4.25}$$

for all $j = 1, \dots, n$, *and set*

$$\lambda_j^e := \frac{w_j + \sqrt{D_j}}{2a_j}, \quad \lambda_j^i := \frac{\overline{b}_j}{a_j\lambda_j^e}, \tag{4.26}$$

$$\lambda^e := \min_{j=1,\dots,n} \lambda_j^e, \quad \lambda^i := \max_{j=1,\dots,n} \lambda_j^i. \tag{4.27}$$

If $\lambda^e > \lambda^i$ *then there is at least one zero* x^* *of* (4.5) *in the (inclusion) box*

$$\boldsymbol{x}^i := [z - \lambda^i v, z + \lambda^i v]. \tag{4.28}$$

The zeros in this region are the only zeros of F *in the* interior *of the (exclusion) box*

$$\boldsymbol{x}^e := [z - \lambda^e v, z + \lambda^e v]. \tag{4.29}$$

The examples later will show that the choice of v greatly influences the quality of the inclusion and exclusion regions. The main difficulty for choosing v is the positivity requirement for every D_j. In principle, a vector v could be found by local optimization, if it exists. A method worth trying could be to choose v as a local optimizer of the problem

$$\begin{aligned} &\max n \log \lambda^e + \sum_{j=1}^{n} \log v_j \\ &\text{s.t.} \quad D_j \geq \eta \quad (j = 1, \dots, n) \end{aligned}$$

where η is the smallest positive machine number. This maximizes locally the volume of the excluded box. However, since λ^e is non-smooth, solving this needs a non-smooth optimizer (such as SolvOpt [129]).

The $\overline{B}_k$ can be constructed using interval arithmetic, for a given reference box $\boldsymbol{x}$ around z. Alternatively, they could be calculated once in a bigger reference box $\boldsymbol{x}_{\text{ref}}$ and later reused on all subboxes of $\boldsymbol{x}_{\text{ref}}$. Saving the $\overline{B}_k$ (which needs the storage of n^3 numbers per zero) provides a simple exclusion test for other boxes. This takes $O(n^3)$ operations, while recomputing the $\overline{B}_k$ costs $O(n^4)$ operations.

It is possible to generalize the exclusion boxes to polytopes, as shown in Shen & Neumaier [200].

4.2.2 Uniqueness regions

Regions in which there is a unique zero can be found most efficiently as follows. First one verifies as in the previous sections an exclusion box $\boldsymbol{x}^e$ which contains no zero except in a much smaller inclusion box $\boldsymbol{x}^i$. The inclusion box can be usually refined further by some iterations with Krawczyk's method, which generally converges quickly if the initial inclusion box is already verified. Thus we may assume that $\boldsymbol{x}^i$ is really tiny, with width determined by rounding errors only.

Clearly, int $(\boldsymbol{x}^e)$ contains a unique zero iff $\boldsymbol{x}^i$ contains at most one zero. Thus it suffices to have a condition under which a tiny box contains at most one zero. This can be done even in fairly ill-conditioned cases by the following test.

Theorem 4.15. *Take an approximate solution $z \in X$ of* (4.5)*, and let B be a matrix such that*

$$|CF[z,\boldsymbol{x}] - I| + \sum |\boldsymbol{x}_k - z_k|\,|CF_k[\boldsymbol{x},z,\boldsymbol{x}]| \le B. \tag{4.30}$$

If $\|B\| < 1$ for some monotone norm then $\boldsymbol{x}$ contains at most one solution x of (4.5).

Proof. Assume that x and x' are two solutions. Then we have

$$0 = F(x') - F(x) = F[x,x'](x'-x) = \Big(F[x,z] + \sum (x'_k - z_k)F_k[x,z,x']\Big)(x'-x). \tag{4.31}$$

Using an approximate inverse C of $F'(z)$ we further get

$$x - x' = \Big((CF[z,x] - I) + \sum (x'_k - z_k)CF_k[x,z,x']\Big)(x'-x). \tag{4.32}$$

Applying absolute values, and using (4.30), we find

$$|x'-x| \le \Big(|CF[z,x] - I| + \sum |CF_k[x,z,x']|\,|x'_k - z_k|\Big)|x'-x| \le B|x'-x|. \tag{4.33}$$

This, in turn, implies $\|x'-x\| \le \|B\|\,\|x'-x\|$. If $\|B\| < 1$ we immediately conclude $\|x'-x\| \le 0$, hence $x = x'$. $\square$

Since B is nonnegative, $\|B\| < 1$ holds for some norm iff the spectral radius of B is less than one (see, e.g., Neumaier [160, Corollary 3.2.3]); a necessary condition for this is that $\max B_{kk} < 1$, and a sufficient condition is that $|B|u < u$ for some vector $u > 0$.

So one first checks whether $\max B_{kk} < 1$. If this holds, one checks whether $\|B\|_\infty < 1$; if this fails, one computes an approximate solution u of $(I-B)u = e$, where e is the all-one vector, and checks whether $u > 0$ and $|B|u < u$. If this fails, the spectral radius of B is very close to 1 or larger. (Essentially, this amounts to testing $I - B$ for being an H-matrix; cf. Neumaier [160, Proposition 3.2.3].)

We can find a matrix B satisfying (4.30) by computing $\hat{B}_k \ge |CF_k[\boldsymbol{x},z,\boldsymbol{x}]|$, for example by interval evaluation, using (4.7), and observing

$$\begin{aligned}|CF[z,\boldsymbol{x}] - I| &\le |CF'(z) - I| + \sum |\boldsymbol{x}_k - z_k|\,|CF_k[z,z,\boldsymbol{x}]|\\ &\le |CF'(z) - I| + \sum |\boldsymbol{x}_k - z_k|\,|CF_k[\boldsymbol{x},z,\boldsymbol{x}]|.\end{aligned}$$

Then, using (4.8), we get

$$|CF[z,\boldsymbol{x}] - I| + \sum |\boldsymbol{x}_k - z_k|\,|CF_k[\boldsymbol{x},z,\boldsymbol{x}]| \le B_0 + 2\sum |\boldsymbol{x}_k - z_k|\,\hat{B}_k =: B, \tag{4.34}$$

where B can be computed using rounding towards $+\infty$.

If F is quadratic, the results simplify again. In this case all $F_k[x', z, x] =: F_k$ are constant, and we can replace $\hat{B}_k$ by $B_k := |CF_k|$. Hence (4.34) becomes

$$B = B_0 + 2\sum |\boldsymbol{x}_k - z_k| B_k.$$

4.2.3 Componentwise exclusion regions around arbitrary points

In a branch-and-bound based method for finding all solutions to (4.5), we not only need to exclude regions close to zeros but also boxes far away from all solutions. This is usually done by interval analysis on the range of F, by constraint propagation methods (see, e.g., van Hentenryck et al. [85]), or by Krawczyk's method or preconditioned Gauss-Seidel iteration (see, e.g., Neumaier [160]). An affine invariant, component-wise version of the latter is presented in this section.

Let z be an arbitrary point in the region of definition of F. Throughout this section, $C \in \mathbb{R}^{m\times n}$ denotes an arbitrary rectangular matrix. M_u is as in (4.13).

Theorem 4.16. *Let $0 < u \in \mathbb{R}^n$, and take $\overline{B}_k \geq B_k(x)$ for all $x \in M_u$. If there is an index $i \in \{1, \dots, n\}$ such that the inequality*

$$\underline{b}_i - (B_0'u)_i - \sum u_k(\overline{B}_k u)_i > 0 \tag{4.35}$$

is valid, then (4.5) has no solution $x \in M_u$.

Proof. We set $\boldsymbol{x} = [z-u, z+u]$. For a zero $x \in M_u$ of F,we calculate using (4.7), similar to the proof of Theorem 4.12,

$$\begin{aligned} 0 = |K(x) - x| &= \Big| -CF(z) - \Big(CF'(z) - \sum CF_k[z,z,x](x_k - z_k)\Big)(x-z)\Big| \\ &\geq |CF(z)| - \Big|(CF'(z) - I)(x-z) + \sum (x_k - z_k) CF_k[z,z,x](x-z)\Big|. \end{aligned} \tag{4.36}$$

Now we use (4.8) and (4.35) to compute

$$\begin{aligned} |CF(z)|_i \geq \underline{b}_i &> (B_0'u)_i + \sum (u_k \overline{B}_k u)_i \\ &\geq \Big(|CF'(z)|u\Big)_i + \sum \Big(u_k |CF_k[z,z,x]\,|u\Big)_i \\ &\geq \Big|CF'(z)(x-z)\Big|_i + \sum \Big|(x_k - z_k)CF_k[z,z,x](x-z)\Big|_i \\ &\geq \Big|(CF'(z) - I)(x-z) + \sum (x_k - z_k) CF_k[z,z,x](x-z)\Big|_i. \end{aligned}$$

This calculation and (4.35) imply

$$\begin{aligned} |CF(z)|_i &- \Big|CF'(z)(x-z) + \sum (x_k - z_k) CF_k[z,z,x](x-z)\Big|_i \\ &\geq \underline{b}_i - (B_0'u)_i - \sum (u_k \overline{B}_k u)_i > 0, \end{aligned}$$

contradicting (4.36). ∎

Again, we need a method to find good vectors u satisfying (4.35). The following theorem provides that.

Theorem 4.17. *Let $S \subseteq X$ be a set containing z, and take $\overline{B}_k \geq B_k(x)$ for all $x \in S$. If for any $0 < v \in \mathbb{R}^n$ we define*

$$\begin{aligned} w^\times &:= B_0' v \\ a^\times &:= \sum v_k \overline{B}_k v \\ D_i^\times &:= {w_i^\times}^2 + 4\underline{b}_i a_i^\times \\ \lambda_i^\times &:= \frac{\underline{b}_i}{w_i^\times + \sqrt{D_i^\times}} \\ \lambda^\times &:= \max_{i=1,\dots,n} \lambda_i^\times \end{aligned} \tag{4.37}$$

then F has no zero in the interior of the exclusion region

$$R^\times := [z - \lambda^\times v, z + \lambda^\times v] \cap S. \tag{4.38}$$

Proof. We set $u = \lambda v$ and check the result (4.35) of Theorem 4.16:

$$0 < \underline{b}_i - (B_0' u)_i - \sum (u_k \overline{B}_k u)_i = \underline{b}_i - \lambda (B_0' v)_i - \lambda^2 \sum (v_k \overline{B}_k v)_i.$$

This quadratic inequality has to be satisfied for some $i \in \{1, \dots, n\}$. The ith inequality is true for all $\lambda \in [0, \lambda_i^\times[$, so we can take the maximum of all these numbers and still have the inequality satisfied for at least one i. Bearing in mind that the estimates are only true in the set S, the result follows from Theorem 4.16. ∎

As in the last section, a vector v could be calculated by local optimization, e.g., as a local optimizer of the problem

$$\max n \log \lambda^\times + \sum_{j=1}^n \log v_j$$

This maximizes locally the volume of the excluded box. Solving this also needs a non-smooth optimizer since $\lambda^\times$ is non-smooth like λ^e. However, in contrast to the v needed in Theorem 4.13, there is no positivity requirement which has to be satisfied. In principle, every choice of v leads to some exclusion region.

Finding a good choice for C is a subtle problem and could be attacked by methods similar to Kearfott, Hu, & Novoa [118]. Example 4.21 below shows that a pseudo inverse of $F'(z)$ usually yields reasonable results. However, improving the choice of C sometimes widens the exclusion box by a considerable amount.

Again, for quadratic F the result can be made global, due to the fact that the $F_k[z,z,x]$ are independent of x.

Corollary 4.18. *Let F be quadratic and $0 < v \in \mathbb{R}^n$. Choose $\overline{B}_k \geq \left|CF_k\right|$, $w_i^\times$, $a_i^\times$, $D_i^\times$, $\lambda_i^\times$, and $\lambda^\times$ as in Theorem 4.17. Then F has no zero in the interior of the exclusion box*

$$\boldsymbol{x}^\times := [z - \lambda^\times v, z + \lambda^\times]. \tag{4.39}$$

Proof. This is a direct consequence of Theorem 4.17 and the fact that all $F_k[z,z,x]$ are constant in x. ☐

Results analogous to Theorems 4.13, 4.15, and 4.17 can be obtained for exclusion regions in global optimization problems by applying the above techniques to the first order optimality conditions. Nothing new happens mathematically, so giving details seems not necessary.

4.2.4 Examples

We illustrate the theory with a some low-dimensional examples, but note that the improvements over traditional results are usually even more pronounced in a branch and bound context for higher dimensional problems; cf. Schichl & Neumaier [200, Example 4].

Example. 4.19. *We continue Example 4.10, doing all calculations symbolically, hence free of rounding errors, assuming a known zero. (This idealizes the practically relevant case where a good approximation of a zero is available from a standard zero-finder.)*

We consider the system of equations (4.9), which has the four solutions $\pm\binom{3}{4}$ and $\pm\binom{4}{3}$; cf. Figure 4.12. The system has the form (4.5) with F given by (4.10). If we take the solution $x^* = \binom{3}{4}$ as center z, we can use the slope calculations from the introduction. From (4.24) we get

$$w_j = v_j, \quad D_j = v_j^2 \quad (j = 1, 2),$$

$$a_1 = \tfrac{1}{14}(3v_1^2 + 8v_1v_2 + 3v_2^2), \quad a_2 = \tfrac{1}{14}(4v_1^2 + 6v_1v_2 + 4v_2^2),$$

and for the particular choice $v = \binom{1}{1}$, we get from (4.26)

$$\lambda^i = 0, \qquad \lambda^e = 1. \tag{4.40}$$

Thus, Corollary 4.14 implies that the interior of the box

$$[x^* - v, x^* + v] = \begin{pmatrix} [2,4] \\ [3,5] \end{pmatrix}$$

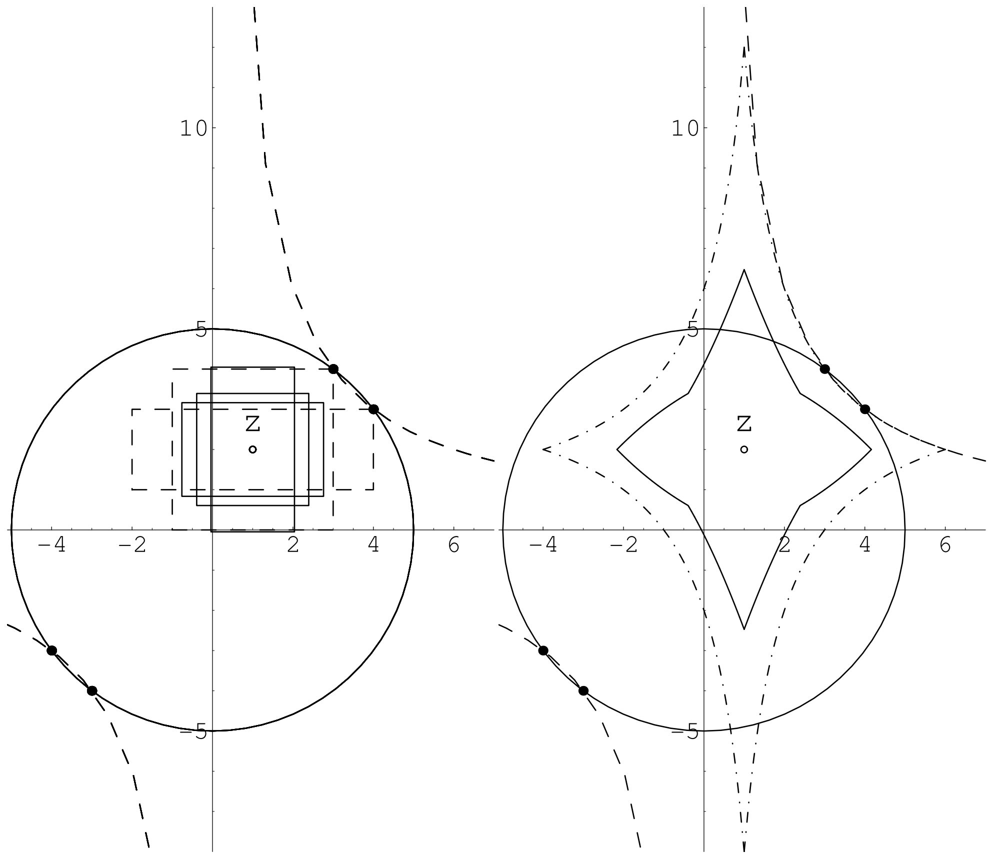


Figure 4.12. *Maximal exclusion boxes around* $\binom{1}{2}$ *and total excluded region for Example 4.19*

contains no solution apart form $\binom{3}{4}$. *This is best possible, since there is another solution* $\binom{4}{3}$ *at a vertex of this box. The choice* $v = \binom{1}{2}$, $\omega(v) = \frac{8}{7}$ *gives another exclusion box, neither contained in nor containing the other box.*

If we consider the point $z = \binom{1}{2}$, *we find*

$$F(z) = \begin{pmatrix} -20 \\ -10 \end{pmatrix}, \quad F'(z) = \begin{pmatrix} 2 & 4 \\ 2 & 1 \end{pmatrix}, \quad C = \frac{1}{6}\begin{pmatrix} -1 & 4 \\ 2 & -2 \end{pmatrix},$$

$$\underline{b} = \frac{10}{3}\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad B_0 = 0, \quad B_1 = \frac{1}{6}\begin{pmatrix} 1 & 0 \\ 2 & 0 \end{pmatrix}, \quad B_2 = \frac{1}{6}\begin{pmatrix} 4 & 1 \\ 2 & 2 \end{pmatrix},$$

$$w^\times = v, \quad a^\times = \frac{1}{6}\begin{pmatrix} v_1^2 + 4v_1v_2 + v_2^2 \\ 2v_1^2 + 2v_1v_2 + 2v_2^2 \end{pmatrix},$$

$$D_1^\times = \frac{1}{9}(29v_1^2 + 80v_1v_2 + 20v_2^2), \quad D_2^\times = \frac{1}{9}(40v_1^2 + 40v_1v_2 + 49v_2^2).$$

Since everything is affine invariant and $v > 0$, we can set $v = (1, v_2)$, and we compute

$$\lambda^\times = \begin{cases} \frac{20}{3v_2+\sqrt{40+40v_2+49v_2^2}} & \text{if } v_2 \le 1, \\ \frac{30}{3+\sqrt{29+80v_2+20v_2^2}} & \text{if } v_2 > 1. \end{cases}$$

Depending on the choice of v_2, the volume of the exclusion box varies. There are three locally best choices $v_2 \approx 1.97228$, $v_2 \approx 0.661045$, and $v_2 = 1$, the first providing the globally maximal exclusion box.

For any two different choices of v_2 the resulting boxes are never contained in one another. Selected maximal boxes are depicted in Figure 4.12 (left) in solid lines; the total region which can be excluded by Corollary 4.18 is shown in solid lines in the right part of the figure.

The optimal preconditioner for exclusion boxes, however, does not need to be an approximate inverse to $F'(z)$. In this case, it turns out that $C = (0\ 1)$ is optimal for every choice of v. Two clearly optimal boxes and the total excluded region for every possible choice of v with $C = (0\ 1)$ can be found in Figure 4.12 in dashed lines.

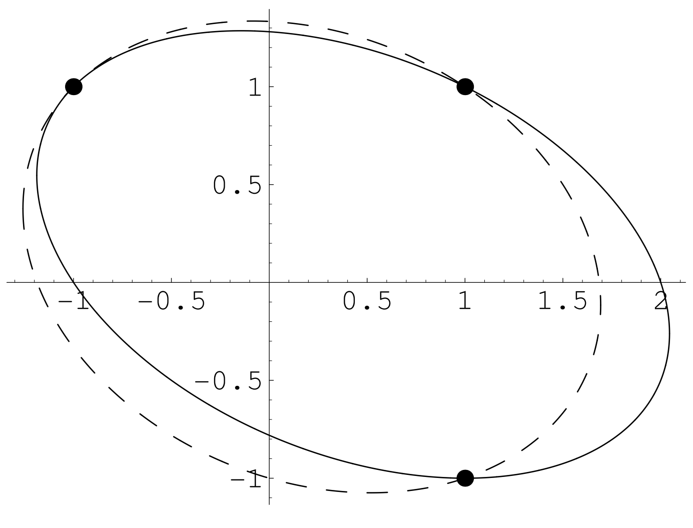


Figure 4.13. *Two quadratic equations in two variables, Example 4.20.*

Example. 4.20. *The system of equations* (4.5) *with*

$$F(x) = \begin{pmatrix} x_1^2 + x_1x_2 + 2x_2^2 - x_1 - x_2 - 2 \\ 2x_1^2 + x_1x_2 + 3x_2^2 - x_1 - x_2 - 4 \end{pmatrix} \tag{4.41}$$

has the solutions $\binom{1}{1}$, $\binom{1}{-1}$, $\binom{-1}{1}$, cf. Figure 4.13. It is easily checked that

$$F[z,x] = \begin{pmatrix} x_1 + x_2 + z_1 - 1 & 2x_2 + z_1 + 2z_2 - 1 \\ 2x_1 + x_2 + 2z_1 - 1 & 3x_2 + z_1 + 3z_2 - 1 \end{pmatrix}$$

satisfies (3.3). *Thus* (4.7) *holds with*

$$F'(z) = \begin{pmatrix} 2z_1+z_2-1 & z_1+4z_2-1 \\ 4z_1+z_2-1 & z_1+6z_2-1 \end{pmatrix}, \qquad F_1 = \begin{pmatrix} 1 & 0 \\ 2 & 0 \end{pmatrix}, \qquad F_2 = \begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix}.$$

We consider boxes centered at the solution $z = x^* = \binom{1}{1}$. *For*

$$\boldsymbol{x} = [x^* - \varepsilon u, x^* + \varepsilon u] = \begin{pmatrix} [1-\varepsilon, 1+\varepsilon] \\ [1-\varepsilon, 1+\varepsilon] \end{pmatrix},$$

we find

$$F'[x^*, \boldsymbol{x}] = \begin{pmatrix} [2-2\varepsilon, 2+2\varepsilon] & [4-2\varepsilon, 4+2\varepsilon] \\ [4-3\varepsilon, 4+3\varepsilon] & [6-3\varepsilon, 6+3\varepsilon] \end{pmatrix},$$
$$F'(\boldsymbol{x}) = \begin{pmatrix} [2-3\varepsilon, 2+3\varepsilon] & [4-5\varepsilon, 4+5\varepsilon] \\ [4-5\varepsilon, 4+5\varepsilon] & [6-7\varepsilon, 6+7\varepsilon] \end{pmatrix}.$$

The midpoint of $F'(\boldsymbol{x})$ *is here* $F'(z)$, *and the optimal preconditioner is*

$$C := F'(x^*)^{-1} = \begin{pmatrix} -1.5 & 1 \\ 1 & -0.5 \end{pmatrix};$$

from this, we obtain

$$B_1 = \begin{pmatrix} 0.5 & 0 \\ 0 & 0 \end{pmatrix}, \qquad B_2 = \begin{pmatrix} 0.5 & 0 \\ 0.5 & 0.5 \end{pmatrix}.$$

The standard uniqueness test checks for a given box $\boldsymbol{x}$ *whether the matrix* $F'(\boldsymbol{x})$ *is strongly regular (*NEUMAIER *[160]). But given the zero* x^* *(or in finite precision calculations, a tiny enclosure for it), it suffices to show strong regularity of* $F[x^*, \boldsymbol{x}]$. *We find*

$$|I - CF'(\boldsymbol{x})| = \frac{\varepsilon}{2}\begin{pmatrix} 19 & 29 \\ 11 & 17 \end{pmatrix},$$

with spectral radius $\varepsilon(9+4\sqrt{5}) \approx 17.944\varepsilon$. *Thus* $F'(\boldsymbol{x})$ *is strongly regular for* $\varepsilon < 1/17.944 = 0.0557$. *The exclusion box constructed from slopes is better, since*

$$|I - CF[x^*, \boldsymbol{x}]| = \varepsilon\begin{pmatrix} 6 & 6 \\ 3.5 & 3.5 \end{pmatrix},$$

has spectral radius 9.5ε. *Thus* $F[x^*, \boldsymbol{x}]$ *is strongly regular for* $\varepsilon < 1/9.5$, *and we get an exclusion box of radius* $1/9.5$.

The Kantorovich Theorem 3.2 yields the following results:

$$F'' = \left(\begin{pmatrix} 2 & 1 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 \\ 1 & 6 \end{pmatrix}\right),$$
$$\alpha = 2.5, \quad \beta = 0, \quad \gamma = 12, \quad \Delta = 1,$$
$$\underline{r} = 0, \quad \overline{r} = \frac{2}{2.5 \cdot 12} = \frac{1}{15},$$

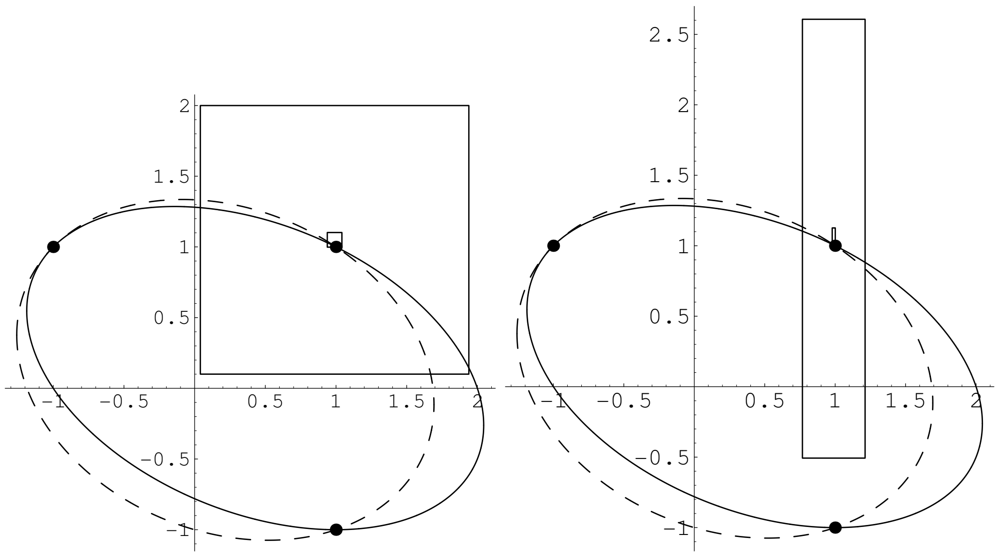


Figure 4.14. $\mathbf{x}^e$ *and* $\mathbf{x}^i$ *calculated for Example 4.20with 3 significant digits for* $v = (1, 1)$ *and* $v = (1, 7)$ *at* $z = (0.99, 1.05)$

hence it provides an even smaller (i.e., inferior) exclusion box of radius $\frac{1}{15}$.

If we apply Kahan's Theorem 3.3 with $F'(\boldsymbol{x})$, *we have to check that* $K(\boldsymbol{x}, \boldsymbol{x}) \subseteq \operatorname{int}(\boldsymbol{x})$. *Now*

$$K(\boldsymbol{x}, \boldsymbol{x}) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \frac{\varepsilon}{2} \begin{pmatrix} 19 & 29 \\ 11 & 17 \end{pmatrix} \begin{pmatrix} [-\varepsilon, \varepsilon] \\ [-\varepsilon, \varepsilon] \end{pmatrix}$$

is in $\operatorname{int}(\boldsymbol{x})$ *if*

$$\begin{pmatrix} [1 - 24\varepsilon^2, 1 + 24\varepsilon^2] \\ [1 - 14\varepsilon^2, 1 + 14\varepsilon^2] \end{pmatrix} \subseteq \begin{pmatrix} [1 - \varepsilon, 1 + \varepsilon] \\ [1 - \varepsilon, 1 + \varepsilon] \end{pmatrix},$$

which holds for $\varepsilon < 1/24$. *This result can be improved if we use slopes instead of interval derivatives. Indeed,*

$$K(z, \boldsymbol{x}) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \varepsilon \begin{pmatrix} 6 & 6 \\ 3.5 & 3.5 \end{pmatrix} \begin{pmatrix} [-\varepsilon, \varepsilon] \\ [-\varepsilon, \varepsilon] \end{pmatrix}$$

is in $\operatorname{int}(\boldsymbol{x})$ *if*

$$\begin{pmatrix} [1 - 12\varepsilon^2, 1 + 12\varepsilon^2] \\ [1 - 7\varepsilon^2, 1 + 7\varepsilon^2] \end{pmatrix} \subseteq \begin{pmatrix} [1 - \varepsilon, 1 + \varepsilon] \\ [1 - \varepsilon, 1 + \varepsilon] \end{pmatrix},$$

i.e., for $\varepsilon < 1/12$.

Now we consider the new results. From (4.26) *we get*

$$\lambda^e = \frac{2}{v_1 + v_2} \tag{4.42}$$

In exact arithmetic, we find $\lambda^e = 1$, *so that Corollary 4.14 implies that the interior*

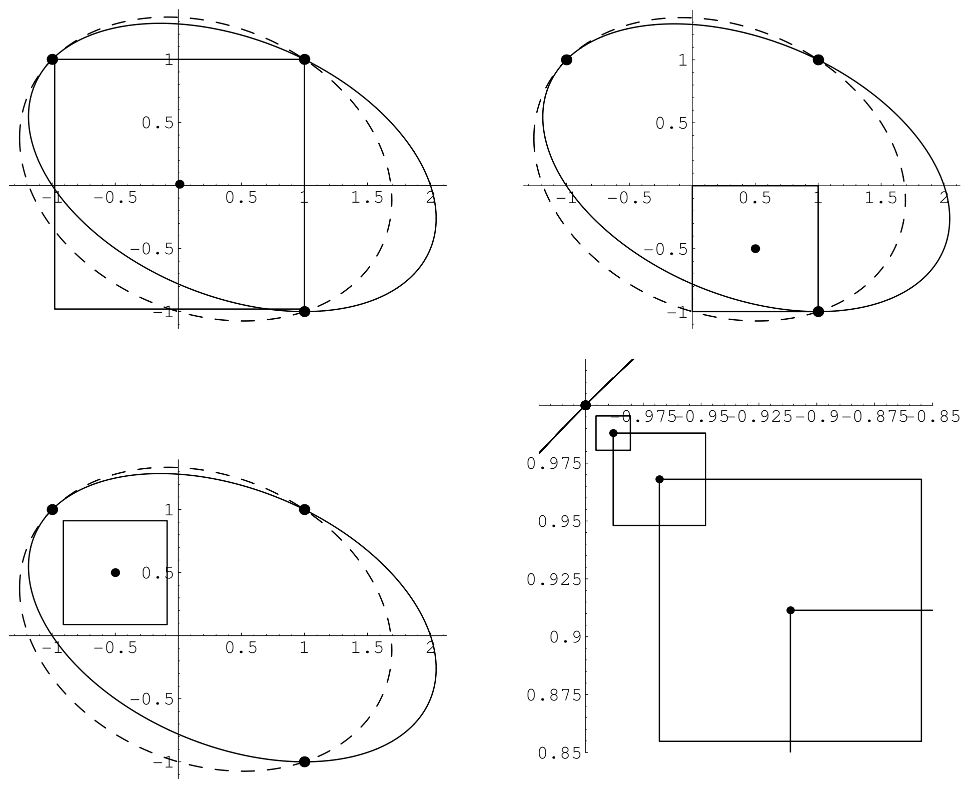


Figure 4.15. $\mathbf{x}^{\times}$ *for Example 4.20 and various choices of z and $v = (1,1)$.*

of the box

$$[x^* - v, x^* + v] = \begin{pmatrix} [0,2] \\ [0,2] \end{pmatrix} \tag{4.43}$$

contains no solution apart from z. In this example, the box is not as large as desirable, since in fact the larger box

$$[x^* - 2v, x^* + 2v] = \begin{pmatrix} [-1,3] \\ [-1,3] \end{pmatrix}$$

contains no other solution. However, the box (4.43) is still one order of magnitude larger than that obtained from the standard uniqueness tests or the Kantorovich theorem.

arithmetic (we used Mathematica with three significant digits, using this artificially low precision to make the inclusion regions visible in the pictures) and only approximative zeros, the results do not change too much, which can be seen in the pictures of Figure 4.14.

Corollary 4.18 also gives very promising results. The size of the exclusion boxes again depends on the center z and the vector v. The results for various choices can be found in Figure 4.15.

For a nonquadratic polynomial function, all calculations become more complex, and the exclusion sets found are usually far from optimal, though still much better than those from the traditional methods. The $F_k[z,z,x]$ are no longer independent of x, so Theorems 4.13 and 4.17 have to be applied. This involves the computation of a suitable upper bound $\overline{B}_k$ of $F_k[z,z,x]$ by interval arithmetic.

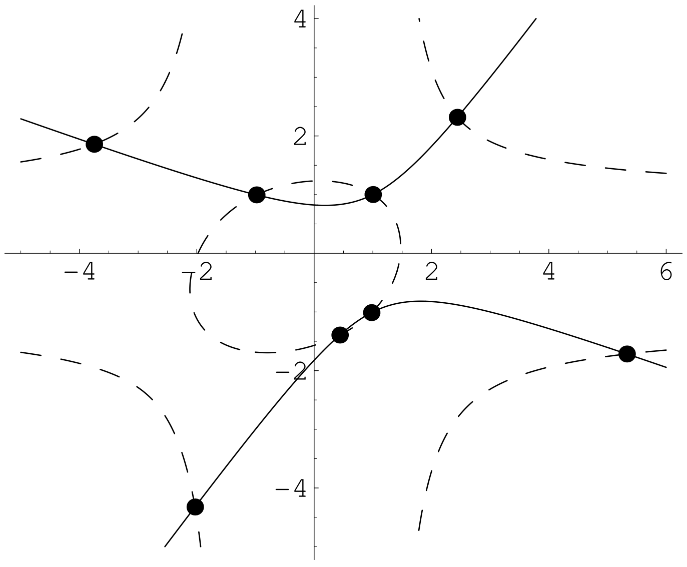


Figure 4.16. *Two polynomial equations in two variables, Example 4.21.*

Example. 4.21. *Figure 4.16 displays the following system of equations $F(x) = 0$ in two variables, with two polynomial equations of degree 2 and 8:*

$$\begin{aligned} F_1(x) &= x_1^2 + 2x_1x_2 - 2x_2^2 - 2x_1 - 2x_2 + 3, \\ F_2(x) &= x_1^4x_2^4 + x_1^3x_2^4 + x_1^4x_2^3 + 15x_1^2x_2^4 - 8x_1^3x_2^3 + 10x_1^4x_2^2 + 3x_1x_2^4 + 5x_1^2x_2^3 \\ &\quad + 7x_1^3x_2^2 + x_1^4x_2 - 39x_2^4 + 32x_1x_2^3 - 57x_1^2x_2^2 + 21x_1^3x_2 - 17x_1^4 - 27x_2^3 - 17x_1x_2^2 \\ &\quad - 8x_1^2x_2 - 18x_1^3 - 478x_2^2 + 149x_1x_2 - 320x_1^2 - 158x_2 - 158x_1 + 1062. \end{aligned} \tag{4.44}$$

The system (4.44) *has* 8 *solutions, at approximately*

$$\begin{pmatrix} 1.0023149901708083 \\ 1.0011595047756938 \end{pmatrix}, \quad \begin{pmatrix} 0.4378266929701329 \\ -1.3933047617799774 \end{pmatrix}, \quad \begin{pmatrix} 0.9772028387127761 \\ -1.0115934531170049 \end{pmatrix},$$
$$\begin{pmatrix} -0.9818234823156266 \\ 0.9954714636375825 \end{pmatrix}, \quad \begin{pmatrix} -3.7502535429488344 \\ 1.8585101451403585 \end{pmatrix}, \quad \begin{pmatrix} 2.4390986061035260 \\ 2.3174396617957018 \end{pmatrix},$$
$$\begin{pmatrix} 5.3305903297000243 \\ -1.7161362016394848 \end{pmatrix}, \quad \begin{pmatrix} -2.0307311621763933 \\ -4.3241016906293375 \end{pmatrix}.$$

We consider the approximate solution $z = \binom{0.99}{1.01}$. For the set S we choose the box

$[z-u, z+u]$ with $u = \binom{1}{1}$. In this case we have

$$F(z) \approx \begin{pmatrix} -0.0603 \\ -1.170 \end{pmatrix}, \quad F'(z) \approx \begin{pmatrix} 2 & -4.06 \\ -717.55 & -1147.7 \end{pmatrix},$$

$$F_1[z,z,x] = \begin{pmatrix} 1 & 0 \\ f_1 & 0 \end{pmatrix}, \quad F_2[z,z,x] = \begin{pmatrix} 2 & -2 \\ f_2 & f_3 \end{pmatrix},$$

where

$$\begin{aligned} f_1 \approx &-405.63 - 51.66x_1 - 17x_1^2 + 36.52x_2 + 23x_1x_2 + x_1^2x_2 - \\ &- 13.737x_2^2 + 26.8x_1x_2^2 + 10x_1^2x_2^2 - 7.9x_2^3 - 6.02x_1x_2^3 + x_1^2x_2^3 + \\ &+ 19.92x_2^4 + 2.98x_1x_2^4 + x_1^2x_2^4, \\ f_2 \approx &\quad 191.04 - 7.6687x_2 + 62.176x_2^2 + 39.521x_2^3, \\ f_3 \approx &-588.05 - 36.404x_2 - 19.398x_2^2. \end{aligned}$$

We further compute

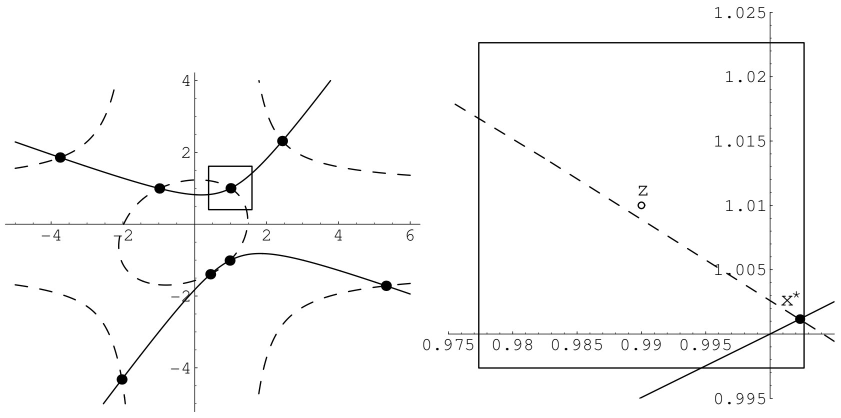


Figure 4.17. *Exclusion and inclusion boxes for Example 4.21 at* $z = (0.99, 1.01)$

$$C = \begin{pmatrix} 0.22035 & -0.00077947 \\ -0.13776 & -0.00038397 \end{pmatrix}, \quad B_0 = 10^{-5} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

$$\overline{B}_1 = \begin{pmatrix} 1.0636 & 0 \\ 0.5027 & 0 \end{pmatrix}, \quad \overline{B}_2 = \begin{pmatrix} 0.3038 & 0.1358 \\ 0.5686 & 0.5596 \end{pmatrix}, \quad \overline{b} = \begin{pmatrix} 0.0124 \\ 0.0088 \end{pmatrix}.$$

If we use Theorem 4.13 for $v = \binom{1}{1}$, we get

$$w = \begin{pmatrix} 0.99999 \\ 0.99998 \end{pmatrix}, \quad a = \begin{pmatrix} 1.5032 \\ 1.6309 \end{pmatrix}, \quad D = \begin{pmatrix} 0.925421 \\ 0.942575 \end{pmatrix},$$

$$\lambda^i = 0.0126403, \quad \lambda^e = 0.604222,$$

so we may conclude that there is exactly one zero in the box

$$\boldsymbol{x}^i = \begin{pmatrix} [0.97736, 1.00264] \\ [0.99736, 1.02264] \end{pmatrix},$$

and this zero is the only zero in the interior of the exclusion box

$$\boldsymbol{x}^e = \begin{pmatrix} [0.385778, 1.59422] \\ [0.405778, 1.61422] \end{pmatrix}.$$

In Figure 4.17 the two boxes are displayed. In Figure 4.18 exclusion boxes and some inclusion boxes for all of the zeros of F are provided.

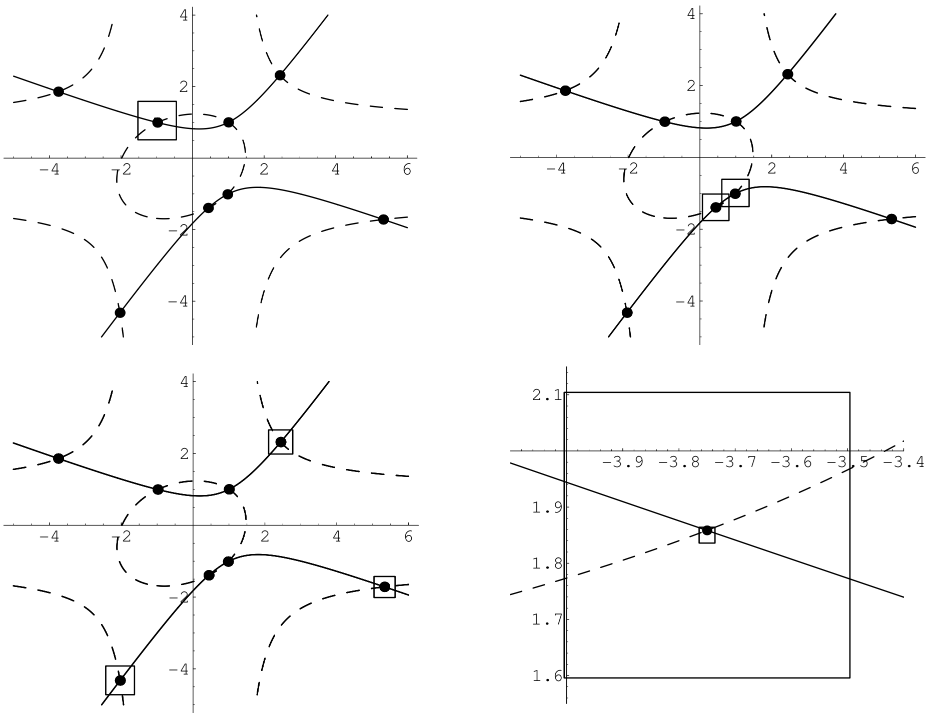


Figure 4.18. *Exclusion boxes for all zeros of F in Example 4.21.*

Next we consider the point $z = \binom{1.5}{-1.5}$ to test Theorem 4.17. We compute

$$F(z) \approx \begin{pmatrix} -3.75 \\ -1477.23 \end{pmatrix}, \quad F_1[z, z, x] \approx \begin{pmatrix} 1 & 0 \\ g_1 & 0 \end{pmatrix},$$

$$F'(z) \approx \begin{pmatrix} -2 & 7 \\ -1578.73 & 1761.77 \end{pmatrix}, \quad F_2[z, z, x] = \begin{pmatrix} 2 & -2 \\ g_2 & g_3 \end{pmatrix},$$

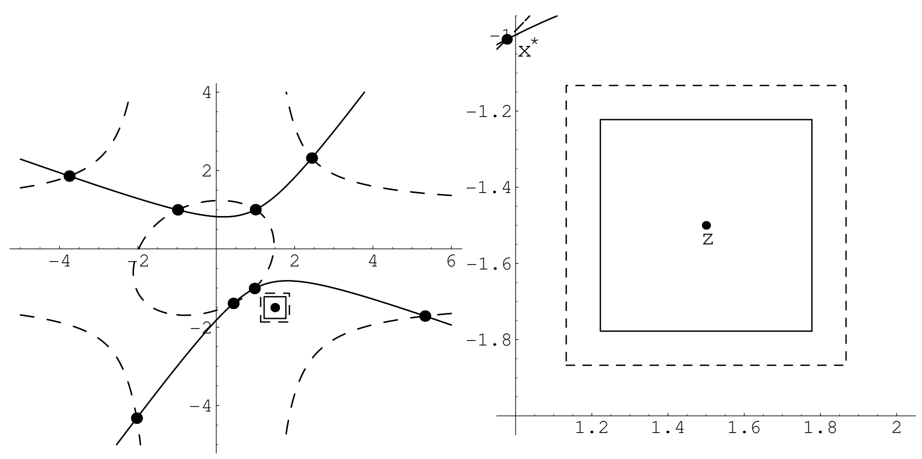


Figure 4.19. *Exclusion boxes for Example 4.21 at* $z = (1.5, -1.5)$.

with

$$\begin{aligned}
g_1 \approx & -488.75 - 69x_1 - 17x_1^2 + 61.75x_2 + 24x_1x_2 + x_1^2x_2 + \\
& + 31.5x_2^2 + 37x_1x_2^2 + 10x_1^2x_2^2 - 12.25x_2^3 - 5x_1x_2^3 + x_1^2x_2^3 + \\
& + 24.75x_2^4 + 4x_1x_2^4 + x_1^2x_2^4, \\
g_2 \approx & \quad 73.1563 + 138.063x_2 - 95.875x_2^2 + 68.25x_2^3, \\
g_3 \approx & -536.547 - 12.75x_2 + 7.6875x_2^2.
\end{aligned}$$

Performing the necessary computations, we find for $\boldsymbol{x} = [z-u, z+u]$ *with* $u = \frac{1}{2}\binom{1}{1}$.

$$F'(z)^{-1} \approx \begin{pmatrix} 0.234 & -0.00093 \\ 0.21 & -0.000266 \end{pmatrix}, \quad \underline{b} = \begin{pmatrix} 0.496 \\ 0.3939 \end{pmatrix},$$

$$\overline{B}_1 = \begin{pmatrix} 1.2895 & 0 \\ 0.5113 & 0 \end{pmatrix}, \quad B_0' = \begin{pmatrix} 1 & 10^{-5} \\ 10^{-5} & 1.00001 \end{pmatrix}, \quad \overline{B}_2 = \begin{pmatrix} 1.5212 & 0.0215 \\ 0.7204 & 0.2919 \end{pmatrix}.$$

Now we use Theorem 4.17 for $v = \binom{1}{1}$ *and* $C = F'(z)^{-1}$ *and get*

$$w^\times = \begin{pmatrix} 1.00001 \\ 1.00002 \end{pmatrix}, \quad a^\times = \begin{pmatrix} 2.8322 \\ 1.5236 \end{pmatrix}, \quad D^\times = \begin{pmatrix} 6.6191 \\ 3.4006 \end{pmatrix}, \quad \lambda^\times = 0.277656;$$

so we conclude that there are no zeros of F *in the interior of the exclusion box*

$$\boldsymbol{x}^\times = \begin{pmatrix} [1.22234, 1.77766] \\ [-1.77766, -1.22234] \end{pmatrix}.$$

However, the choice $C = F'(z)^{-1}$ *is not best possible in this situation. If we take*

$$C = \begin{pmatrix} 1 & 0.002937 \end{pmatrix},$$

we compute $\lambda^\times = 0.367223$ *and find the considerably larger exclusion box*

$$\boldsymbol{x}^\times = \begin{pmatrix} [1.13278, 1.86722] \\ [-1.86722, -1.13278] \end{pmatrix}.$$

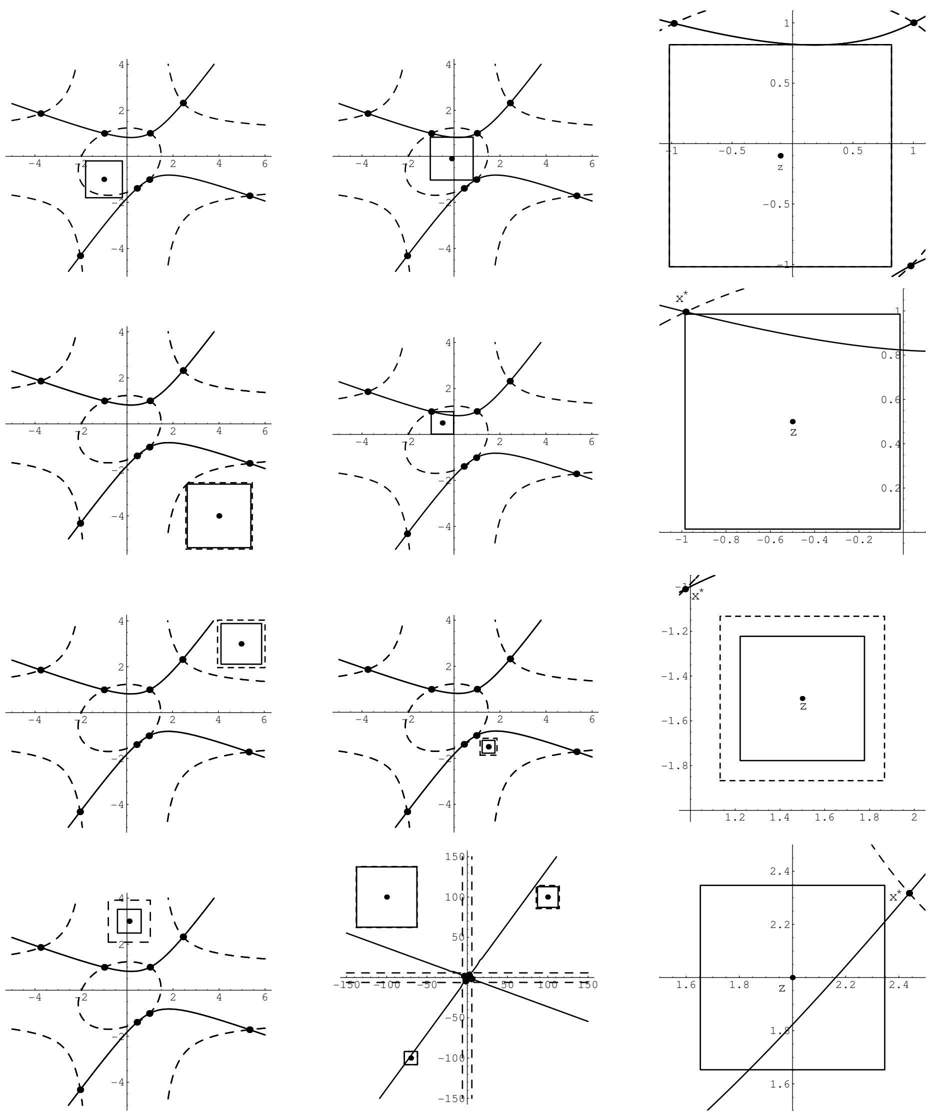


Figure 4.20. *Exclusion boxes for Example 4.21 in various regions of* $\mathbb{R}^2$

Figure 4.19 shows both boxes, the bigger one in dashed lines, and Figure 4.20 contains various exclusion boxes for nonzeros.

Part II

Open solver platform for global optimization

Chapter 5
The COCONUT environment

At the turn of the millennium six European academic institutions

- EPFL Lausanne, Switzerland,
- IRIN Nantes, France,
- Technische Hochschule Darmstadt, Germany,
- University of Coimbra, Portugal,
- University of Louvain-la-Neuve, Belgium,
- University of Vienna, Austria,

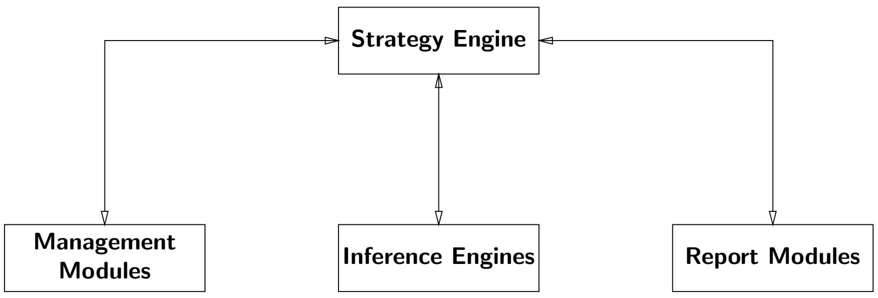


Figure 5.1. *Basic Scheme of the Algorithm*

joined forces with

- ILOG, France, the biggest company selling optimization software,

and applied for a joint project to bring together expertise from the different fields for boosting the development of the area. The COCONUT (**Co**ntinuous **Con**straints **U**pdating the **T**echnology [36] [36]) project was funded by the Future and Emerging Technologies (FET) arm of the IST programme FET-Open scheme of the European Community (IST-2000-26063) and started in November 2000. At the beginning it was planned to "integrate the currently available techniques from mathematical programming, constraint programming, and interval analysis into a single discipline, to get algorithms for global optimization and continuous constraint satisfaction problems that outperform the current generation of algorithms based on using only techniques from one or two of the traditions." First, a common theoretical basis was created, leading to the state of the art report [25], the to that date most thorough analysis of the field. Unfortunately, soon it became clear that the technological platform to fulfill that intention was missing. This required a restructuring of the project, shifting it from a theoretical cooperation towards software development. The effort led to the design and implementation of the *COCONUT Environment*, a modular solver environment for nonlinear global optimization problems with an open-source kernel, which can be expanded by commercial and open-source solver components, bridging the gap between academic demands and commercial interest.

The API (application programmer's interface) is designed to make the development of the various module types independent of each other and independent of the internal model representation. It is a collection of open-source `C++` [219] classes protected by the `LGPL` [69] license model, so it could be used as part of commercial software. Support for dynamic linking relieves the user from recompilation when modules are added or removed and makes it possible for software vendors to provide precompiled libraries. In addition, it is designed for distributed computing, and will probably support parallel computing, as well, in the years after the end of the COCONUT project.

The API kernel implementation consists of more than 50.000 lines of `C++` code and a few `perl` scripts, organized into about 150 files, occupying 1.5 MB of disk space.

The algorithmic design follows the scheme depicted in Figure 5.1; its various parts are described in more detail in the following sections.

This chapter is a description of the internal structure of the API and of the internal structure of API and the basic libraries VGTL and VDBL. A C++ reference of the most important classes and methods can be found in the technical reports [197, 199, 198].

The sections in this chapter are devoted to the description of the basic building blocks of the open solver platform.

In Sections 5.1 and 5.2 the basic libraries on which the system is built, the VGTL and the VDBL are reviewed. In the following sections the environment itself is described.

The C++ classes of the API encapsulate the internal data structure of the environment in such a way that all modular solver components can be written (almost) without knowledge about the internal structuring of the data.

The API consists of two main parts. The first part is responsible for building the internal representation of the optimization problem, the search graph, the search database, all which is needed for the solution process. The structure starts with its small parts, the *expression nodes*. These are the basic operators admissible for constructing the mathematical expressions in an optimization problem. A directed acyclic graph (see Section 4.1) of expression nodes with additional information on objective function and constraints is called a *model*. It represents one global optimization problem, be it the original one or a problem which was derived during the solution process. Since the basic algorithm is a branch-and-bound scheme, various subproblems are created during the solution process. These are organized into a directed acyclic graph of models, the *search graph*. Because storing complete models in the search graph would be very memory consuming, the search graph only stores changes to previous models, i.e., if a model is split in two smaller ones, only the variable bounds which have changed are kept in the search graph. Such a change is called a *delta*, and for every type of change a special class of deltas is available. The classes and methods for these structures is described in Section 5.3.

The second part of the API was designed to make evaluation of expressions, as used by the solver modules, independent of the internal expression DAG structure. For this purpose a number of *evaluators* were designed, see Section 5.6. This should make it possible to later change the internal representation of the DAG without having to change most of the solver components.

Section 5.8 describes the *management modules*, modules for initializing and manipulating the search graph and the models.

The *report modules*, small parts used for producing output, human or machine readable, are explained in Section 5.9.

Most important for the solution process are the solver components, here called *inference engines* and *graph analyzers*, depending on whether they analyze a model or the search graph. The base class for those modules is described in Section 5.7.

The inference engines, which are available in the COCONUT environment, are described in Chapter 6.

The last section of this chapter is devoted to a short description of the *strategy engine*, the central part of the algorithm. It can be programmed using a *strategy language* based on Python [183]. This component, developed at IRIN, University of Nantes, will be briefly explained in Section 5.10.

5.1 Vienna Graph Template Library (VGTL)

All graphs used in the COCONUT API are based on this standalone library.

The Vienna Graph Template Library (VGTL) is a generic graph library with generic programming [155] structure. Its design is similar to the standard template library STL [217], and it uses STL containers like `map`, `list`, and `vector` to organize the internal structure of the graphs.

A collection of walking algorithms for analyzing and working with the graphs are implemented as generic algorithms. Similar to STL iterators, which are used to handle data in containers independently of the container implementation, for graphs the *walker* concept (see Section 5.1.2) is introduced.

5.1.1 Core components

This section describes the few extensions to the STL needed for using the VGTL.

C array to vector adaptor

The library contains an adaptor, which allows ordinary arrays to be accessed like STL vectors, as long as no `insert` and `erase` methods and their relatives are performed.

```
template <class _T>
class array_vector : public std::vector<_T>
{
public:
  array_vector();

  // constructor building an array_vector from pointer __a with size n
  array_vector(_T* __a, int n);

  ~array_vector();

  // assign an array __a of length n to this array_vector.
  void assignvector(_T* __a, int n);
};
```

Reverse search

New search routines `rfind`, and `rfind_if` have been defined, which work like the STL algorithms `find` and `find_if`, respectively, except that they work for bidirectional iterators and search from the back.

`find` (`find_if`) returns the last iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) == true`. If no such iterator is found, `last` is returned.

```
template <class BidirectionalIterator, class T>
BidirectionalIterator rfind(BidirectionalIterator first,
                            BidirectionalIterator last,
                            const T& val);

template <class BidirectionalIterator, class Predicate>
BidirectionalIterator rfind_if(BidirectionalIterator first,
                               BidirectionalIterator last,
                               Predicate pred);
```

5.1.2 Walker

A **walker** is, like an STL iterator, a generalization of a pointer. It dereferences to the data a graph node stores.

There are two different kinds of walkers: **recursive** walker and **iterative** walker.

Recursive Walker

A recursive walker is a pointer to graph nodes, which can be moved around on the graph by changing the node it points to. Walkers can move along the edges of the graph to new nodes. The operators reserved for that are $<<$ for moving along in-edges and $>>$ for moving along out-edges. A recursive walker does not have an internal status, so the walking has to be done recursively. The following methods and operators are defined for walkers:

```
unsigned int n_children() const;
unsigned int n_parents() const;
bool is_root() const;
bool is_leaf() const;
bool is_ground() const;
bool is_sky() const;

children_iterator child_begin();
children_iterator child_end();

parents_iterator parent_begin();
parents_iterator parent_end();

bool operator==(const walker&) const;
bool operator!=(const walker&) const;

walker operator<<(parents_iterator);
walker operator>>(children_iterator);
walker& operator<<=(parents_iterator);
walker& operator>>=(parents_iterator);

walker& operator=(const walker&);
```

The methods `n_children` and `n_parents` return the number of in-edges and out-edges, respectively, of the node the walker points to. For checking whether a walker points to a local sink or a local source, the methods `is_root` and `is_leaf` are provided. The implementation of the VGTL directed graph types uses virtual leafs for accessing the local sources and local sinks of the graph. We add one node (the *ground node*), which is the parent of all local sinks, and one node (the *sky node*), which is child of all local sources, to the graph. After this augmentation, the graph contains exactly one source and one sink. The methods for checking, whether a walker points to one of the two virtual nodes are `is_ground` and `is_sky`, respectively.

For iterating through all in-edges, the `children_iterator`s are used. The first child of the node, the walker points to, can be accessed by `child_begin`. This returns a children iterator, which can be used like an STL iterator. `child_end` points past the end of the last child. Moving the walker to a new node along an in-edge works by using the operator `>>`. If `c` is a `children_iterator` pointing to a valid in-edge of the node pointed at by walker `w`, the expression `w >> c` returns a walker to the child reachable via the edge `c`.

The methods `parent_begin` and `parent_end` and the operator `<<` work the same way for out-edges.

Iterative Walker

An iterative walker (automatic walker) can walk through a graph without guidance. Simply using the operators `++` and `--`, the walker itself searches for the next node in the walk. Iterative walkers are not defined for directed acyclic graphs, so I will not describe them here in detail.

5.1.3 Container

Several types of graph containers are implemented in the VGTL. I will not describe all of them in detail, since they are not important for the COCONUT environment.

Trees and Forests

The first few of the collection of graph containers are the n-ary trees and forests. These trees come in various flavors: standard trees, labeled trees, with and without data hooks. Trees provide iterative walkers and recursive walkers.

Generic Graphs

Generic graphs don't have directed edges. They are the most general class of graphs, and special walking algorithms are provided for them. Generic graphs only have recursive walkers.

Directed Graphs and DAGs

The most important graphs for the COCONUT environment are **directed graphs**. There are two classes implemented. Standard directed graphs and directed acyclic graphs (DAGs). Directed graphs provide recursive walkers only.

The definition of the original VGTL class is quite complicated, since apart from the type every node contains, the allocator class and the sequence container, which is used for managing in- and out-edges can be chosen. In this section, I will explain how the class looks like, if the standard allocators and the default container `vector` are chosen. For using directed graphs the header file `dag.h` has to be included. The class `dag` is a template depending on the content type of the node, i.e., `dag<int>` would define a dag of integer nodes.

There are two methods working on the whole graph. `empty` returns `true` if the graph is empty, and `clear` removes all nodes but the virtual ones from the graph.

There are two different ways of inserting nodes into a directed graph. The `between` methods insert a new node N between one parent (or some parents) and one child (or some children), such that N's parent is the given parent (are the given parents) and N's child is the specified child (are the specified children). The return value is always a walker pointing to the newly inserted node. The various `between` methods differ in where the edges to the new node are inserted in the parent and the child nodes, respectively.

The `split` methods also insert a new node N between one parent (some parents) and one child (some children), such that N's parent is the given parent (are the given parents) and N's child is the specified child (are the specified children). The return value is always a walker pointing to the newly inserted node. In contrast to the `between` methods, the `split` methods remove existing edges between the parent and the child nodes, effectively disconnecting them before inserting the new node. The various `split` methods differ in where the edges to the new node are inserted in the parent and the child nodes, respectively.

The `insert_subgraph` method works like `between`, but instead of adding just one node, a whole directed graph is inserted between the parents and the children.

There are a bunch of methods for handling single edges. The `add_edge` methods insert edges between given nodes `parent` and `child`, making `parent` a new parent of `child`. The various `add_edge` methods differ in at what position the edge is added in parents and children. Removal of edges can be done by calling `remove_edge`. This method deletes *one* edge from the given child to the specified parent. If there are more edges between the nodes, only the first one is removed. Another remove method is `remove_edge_and_deattach`. It works like `remove_edge` except that it does *not* make sure that the resulting graph is well formed. Whenever a node becomes a local source or local sink it is not connected to the corresponding virtual node. Sometimes, edges have to be moved from one child (parent) to another. While this can be done by removing edges and later adding them, there are faster methods performing the move in a single step, `replace_edge_to_parent` and `replace_edge_to_child`.

Removing a node from the graph is performed by the `erase` method. Special, faster methods `erase_child` and `erase_parent` for removing local sources and local sinks, respectively, are available, too.

The `merge` method merges two nodes `first` and `second` in the graph. The node `second` is removed and all its parents and children are connected to the node `first`. If the predicate `merge_child_edges` is set to `true`, edges in both nodes coming from the same child are identified rather than duplicated, and analogous for `merge_parent_edges`. This method assumes that there is a `merge` method for the node data, as well, and it is called during the merge process, as a method for the data of node `first`, passing the data of node `second`.

Given one or a number of nodes N_i, the method `erase_minimal_subgraph` removes all those nodes N of the graph, which

- are descendants of any N_i,
- and for which there is no path from N to the `ground` not, which does not pass through any of the N_i.

In contrast to that, `erase_maximal_subgraph` removes all nodes, which are descendants of any N_i. The `erase_..._pregraph` methods work the same way for ancestors. All methods return the removed subgraph as a whole, and by assigning the `erased_part` object to an empty `dag` (or `dgraph`) it can be recovered as a directed graph.

The walkers pointing to the virtual nodes can be retrieved by the methods `ground` and `sky`.

5.1.4 Algorithms and Visitors

There is a set of generic algorithms for working with graphs and the data stored in them. They perform graph walks, visiting the vertices in a well-defined sequence. Every algorithm takes as argument a walker pointing to the starting vertex of the graph walk, a visitor responsible for performing operations on the node data during the graph walk, and sometimes additional predicates for directing the walk.

Various methods of the visitor a called during the graph walk, depending on the type of the walk. For each of the standard walks there is a base class for the corresponding visitors, making it visitor programming more comfortable.

All visitor base classes are templates depending on three template parameter. The first corresponds to the type of the node data, the second to the return value `Ret` of the `value` and `vvalue` methods visitor, and the third one is optional defining the type of the second input parameter of the `collect` method. By default, this is a constant reference to `Ret`. However, it can be changed to any type which can be constructed implicitly from a single `Ret`. In particular, it can be changed to `Ret`, or a to a (non-constant) reference to `Ret`.

For all visitor base classes the methods `vvalue` and `value` have to be defined, they are pure virtual in the base class. Both produce the return value of the visitor, where `vvalue` is called at virtual graph nodes and `value` at normal vertices. However, for a useful visitor, other methods should be defined, as well. The methods `collect` and `vcollect` are called for every finished child. The first parameter contains the current vertex data, and through the second parameter the return value for the finished child is passed. Again, `vcollect` is called for virtual nodes, while `collect` is used at ordinary graph vertices.

A `preorder_visitor` is a template especially designed for preorder graph walks, i.e., a walk in a directed graph, for which every vertex is visited *before* its descendants (or ancestors for upward walks) are visited. In addition to the standard methods this visitor also defines `preorder`, which is called before the graph walk proceeds towards the children.

The second visitor base class is the `postorder_visitor`. It is tailored for postorder graph walks, where each vertex is visited *after* all its descendants have been been visited. Additional methods defined for this visitor type are `init`, which can be used to initialize the internal data structure of the visitor, and `postorder`, which is called after all children have been processed.

The final visitor class is the `prepost_visitor` for the most complicated graph walks, where a node is visited *before* all its descendants are processed and again *after* the walk has visited all children. In addition to the methods defined for all visitors, the class definition contains `preorder`, which is called in the preorder visit of the node and `postorder`, which is called after the children have finished.

In the following, I will give a short list of those algorithms, which are typically used with walks on DAGs, as used in the COCONUT environment.

`recursive_preorder_walk`: It takes two arguments, a walker `w` and a visitor `f` and performs a recursive preorder walk starting at `w`, i.e., the walk visits all descendants of the node pointed to by `w`. At every node first the node is visited, and then the walk is continued recursively for all children of the node in succession. The methods of the visitor `f` are called as follows:

- `vinit` is called before walking for every virtual node,
- `vcollect` is called after a child of a virtual node has finished,
- `vvalue` is called to compute the return value of a virtual node,

- `preorder` is called before the children are visited,
- `collect` is called every time a child has finished,
- `value` is called to compute the return value for this node.

`recursive_postorder_walk`: This function takes two arguments, a walker `w` and a visitor `f`. It performs a recursive postorder walk starting at `w`, i.e., the walk visits all descendants of the node pointed to by `w`. At every node first the walk is continued towards all children recursively in the ordering of the out-edges. After the walks of all children have finished, the node is visited. The methods of the visitor `f` are called as follows:

- `vinit` is called before walking for every virtual node,
- `vcollect` is called after a child of a virtual node has finished,
- `vvalue` is called to compute the return value of a virtual node,
- `init` is called before the children are visited to initialize the visitors internal data (Note: This method does *not* have access to the vertex data),
- `collect` is called every time a child has finished,
- `postorder` is called after all children have finished,
- `value` is called to compute the return value for this vertex.

`recursive_walk`: This generic algorithm takes two arguments, a walker `w` and a visitor `f` and performs a recursive preorder and postorder walk starting at `w`, i.e., the walk visits all descendants of the node pointed to by `w`. At every node first the vertex is visited. Then the walk is continued towards all children recursively in the ordering of the out-edges. After the walks of all children have finished, the node is visited again. The methods of the visitor `f` are called as follows:

- `vinit` is called before walking for every virtual node,
- `vcollect` is called after a child of a virtual node has finished,
- `vvalue` is called to compute the return value of a virtual node,
- `preorder` is called *before* the children are visited,
- `collect` is called every time a child has finished,
- `postorder` is called *after* all children have been visited,
- `value` is called to compute the return value for this node.

All other algorithms are variants or mixtures of the walks described above.

`recursive_preorder_walk_if`: This works like `recursive_preorder_walk`, but it can be decided, whether the walk towards the children of a vertex is performed. There are two variants of this algorithm. The first one depends on the return value of the `preorder` method of the visitor. If it returns `true` the walk is continued. The second version depends on a predicate `p`, which decides by returning `true` or `false` when passed the vertex data, whether the walk is continued.

`recursive_postorder_walk_if`: This generic algorithm is a variant of `recursive_postorder_walk`. With the help of an additional predicate `p` the algorithm decides, whether the children of the current vertex should be visited or not. The predicate's input parameter is the vertex data of the current node.

`recursive_walk_if`: This variant of `recursive_walk` can be guided in two ways. First, the boolean return value of `preorder` decides, whether the children of this nodes are visited or ignored. Second, the return value of the `postorder` method decides, whether the current node should be visited again by immediately switching back to preorder mode. If `postorder` returns `true`, the walk is restarted, and if it returns `false`, the walk for this vertex is finished.

A second variant of this algorithm uses two predicates `p1` and `p2` instead of `preorder` and `postorder` to make the decision about continuing the graph walk.

`recursive_cached_walk`: This is still another variant of `recursive_walk`, which works like `recursive_walk_if`, except that the return value of `postorder` is always taken as `false`.

Another form of this walk uses a predicate `p` instead of `preorder` to decide, whether the children are visited or ignored.

`recursive_multi_walk`: This generic algorithm works like `recursive_walk_if`, except that the return value of `preorder` is taken to be `true` at every vertex.

Also this generic algorithm can be called with a predicate to replace the `postorder` method when making the decision about restarting the walk at the current node.

`recursive_..._walk_up`: These generic algorithms works exactly like the corresponding `recursive_..._walk` functions, except that they walk the graph along the in-edges of the vertices towards the predecessors.

The other generic walking algorithms for directed and general graphs can be found in the VGTL reference manual [199, 6.4].

5.2 Vienna Database Library (VDBL)

The Vienna Database Library (VDBL) is an in memory database developed with generic programming in mind. It uses STL containers like `map` and `vector` to organize its internal structure.

Databases in the VDBL consist of tables, which are constructed from columns and rows like ordinary relational databases.

Columns can take arbitrary types, and their values need not be constant. Using function objects, called *methods*, the column values can change according to an *evaluation context*.

It is possible to construct *views* onto the tables of a database.

5.2.1 Database

A VDBL database consists of a number of tables which can be dynamically constructed, changed and destroyed. Every table (see Section 5.2.2) has a unique **name** (a `std::string`) and a unique **table id**, which is used for organizing the internal structure.

There is a general table interface defined in `class table`, which defines the minimal functionality needed for implementing a VDBL table. The structure is defined in such a way that `SQL` interfaces could be written, as well as tables which keep all their data in memory.

In addition to tables, the database knows of *users*. There are access control lists (at the moment not fully implemented) for restricting the access of users to the tables on a global and a column-wise base. The users are defined in the `class user`.

Users can construct **views** onto tables (see Section 5.2.5). These views can restrict a table to a subset of columns and/or rows. Also, additional rows can be defined for a view, and it is even possible to *join* various tables into one view. All views onto tables are constructed within a prespecified *context* (see Section 5.2.7). Using this mechanism, columns can change their value automatically according to the evaluation context. This is, e.g., useful in the COCONUT project for organizing points, where some of the properties change from work node to work node, like whether the point is feasible or not.

The most important methods of the `database` are described below. Note that all of them take the `userid` of the database user performing the operation as its second argument. For the COCONUT environment this `userid` is constant, and can be retrieved from the `work_node` class (see Section 5.4.3) or alternatively from the `search_graph` (see Section 5.4.4) class.

`create_table`: This method creates a new table. It takes two arguments and an optional third. The first argument is the name of the table (either a `C` or a `C++` string). The third argument are the `tableflags`, which are not used in the COCONUT environment, and so this optional parameter can be kept at its default.

`get_table_id`: This returns the `tableid` of the given table. The first parameter is the table name.

`drop_table`: For removing a table call this method. The first parameter for this method is either the name of the table or its id.

`has_table`: This method returns `true`, if the table with the specified `tableid` or name exists.

`get_table`: This accessor for a specific table returns a pointer to it, provided either the name or the `tableid` is passed. It returns `NULL` if the table does not exist.

`create_view`: This method creates a new standard view. Apart from the standard second argument, it has to be provided with a name for the view, the name or id of a table, for which this view should be created, an evaluation context for column retrieval, and the type of the view (see Section 5.2.5).

`get_view_id`: This returns the `viewid` of the given view, where the first parameter is the view's name.

`drop_table`: Views are removed by a call to this method. Its first parameter is either the name of the view or its `viewid`.

`has_view`: This method returns `true`, if the view with the specified `viewid` or name exists.

`get_table`: This is the accessor for views. It returns a pointer to the specified, either by name or by `viewid`, view. The return value is `NULL` if the view does not exist.

5.2.2 Tables

A VDBL table consists of a number of columns and rows. The definition of a table is always done by specifying its columns (see Section 5.2.3). The type of the columns value, which can be any `C++` type, is fixed upon creating the column. This can be done dynamically, like modifying and removing. Optionally, for each column a *default value* can be given (this default value may also change w.r.t. the evaluation context). All columns within a table have a **name** (a `std::string`) and a **column id**, which is used for organizing the column structure of the table internally. A column of a table can be accessed by specifying its name or, equivalently, its column id.

In addition to the column structure, which determines the outline of the table, the table's data is organized in **rows** (see Section 5.2.4). Every row has a **row id**, which is used for internal organization. Rows themselves consist of columns. When creating a new row, strict type checking is done between the row's column entries and the column type stored in the table. Column entries of a row can be left out, if a default value for the column is specified in the table definition.

It is possible to implement differently organized tables, as long as they are subclasses of the `class table`.

Relevant for the COCONUT environment is the **standard table** as described below.

Standard Table

The standard table of the VDBL is a standard structure freshly created from columns and rows. After the table is created, either in a database or as a standalone table, columns (see also Section 5.2.3) have to be defined, such that in later stages rows (see Section 5.2.4) can be added. Columns can be added in later stages, as well, but they need to have a default value. Otherwise, updating the existing rows will fail, as will the addition of the new column.

The most important method of the `standardtable` class are:

`add_col`: The first parameter of this method contains the name of the new column, the second the value. This value is mandatory, whether the column has a default or not. This value is used to determine the column's value type. All rows, which are added later, have to conform to exactly that type in that column. If the column is defined to have a default value, the second parameter not only determines the type but also the default value. The third parameter contains the column flags, described below. The method returns `true` if adding the column was successful.

There are two templatized `add_col` methods in addition, which do not require a prebuilt `col` object containing the column's value. These construct the `col` object themselves calling the copy constructor of the value type for the parameter passed.

`modify_col`: There are three methods for modifying a column. All take the column's name as first parameter. Depending on what to change, the other parameters are a `col` parameter for changing the value, and a `colflags` parameter for altering the column flags, or both if appropriate. The method returns `true`, if the modification was successful.

`drop_col`: This method removes a column from the table. It also deletes all entries of this column in all stored rows. The return value is `true`, if erasing was successful.

`rename_col`: By use of this method, a column's name can be changed. The first parameter is the old name, the second is the new name.

`insert`: This method inserts a new row into the table. A row is described as a vector of column values, each a `pair` of pointers to a name (`string`) and a column value of type `col`. One variant returns in the last reference parameter the `rowid` of the newly created row. The method returns `true`, if inserting the row was successful.

`insert_row`: The second class of insertion methods are templated. They can use any sequential STL container for storing the column name–column value pairs. The biggest insertion variant is able to add a sequence of rows at the same method call. Again, an arbitrary STL sequence container may hold the list of row specifications. These methods return `true` if insertion was successful.

`remove`: This method deletes a row from the table. The only parameter is the `rowid` of the row. The method returns `true` if the row was successfully removed.

`has_col`: A call to the first variant of this method returns `true` if the table contains a column of the specified name.

The second variant checks, whether the row with specified `rowid` contains a column with given `colid`.

`get_row`: This method returns a (constant) reference to the row with the given `rowid`. If the row does not exist, the second parameter `error` is set to `true` and an empty `row` is returned.

`has_def`: The `has_def` method checks, whether the column with the given `colid` has a default value.

`get_def`: This method returns a (constant) reference to the default column entry for the specified `colid`. If no default value exists, it returns an empty `col` and sets the `error` parameter to `true`.

`get_colid`: Finding the `colid` for a given column name is the task of this method.

`retrieve`: This method retrieves the value of a column in a specific row. The first parameter is the `rowid`, the second the `colid`. The third parameter specifies the evaluation context for column evaluation. The value is returned, encapsulated into an `alltype` object. If there is no value defined for the given row, the default value of the column is retrieved. The method returns `true`, if retrieving the column was successful.

In addition to the methods described above, the class provides two iterators, a `col_const_iterator` for iterating through all columns, and a `row_const_iterator` for iteration over the rows. The column iterator dereferences to a name–column id pair, and the row iterator to the row id. The methods `row_begin`, `row_end`, `col_begin`, and `col_end` return iterators for the begin and beyond the end of the respective lists.

Column Flags

The `colflags` class contains two boolean entries, `master_index` specifies, whether all column entries have to be different. The `has_default` entry determines, if the column has a default value. The standard constructor sets both flags to `false`. For setting the `has_default` entry to `true`, use `colflags(true)`.

5.2.3 Columns

VDBL columns are built in a very complicated way using three classes on top of each other, making it possible that arbitrary `C++` types can be stored in a column.

There are two main column classes implemented:

`typed_col`: This column holds constant values of arbitrary types. Their values are independent of the evaluation context (`class vdbl::typed_col<_T>`).

`method_col`: A column of this type holds data, whose value is computed whenever it is retrieved and may depend on the evaluation context (`vdbl::method_col<_T>`). Instead of holding data, it contains a function object (method), which is a subclass of `class vdbl::method<_T>`. This function object is used to calculate the column value.

Within a table different column types can be mixed within different rows and the default value, as long as their content types (strict run-time type checking) coincide. It is, e.g., possible to use a `method_col` as default value and override the default with `typed_col` entries in some rows.

Column base class

The following methods are defined for all column classes.

`setcontext`: This method sets the evaluation context for all value retrieval methods. The first parameter is of class `context`, the real evaluation context. The second parameter is a const pointer to the row, in case the column's value depends on other columns, as well.

`get`: This method retrieves a copy of the value. This value is stored into the only parameter.

`def`: The `def` method works almost like the `get` method. It is called whenever this column is retrieved in the context of the default value. For `typed_col` columns this does not make a difference. However, for `method_col` columns it might be useful to distinguish the two retrieval modes.

`def_copy`, `get_copy`: Those two methods work similar to `def` and `get`, except that they allocate new copies of the value to be returned and pass the pointer back. It is important that the user later `delete`s the value to prevent the occurrence of a memory leak.

`get_ptr`: This method returns a constant pointer to the true value of the column, or `NULL` if the column does not have a true value (e.g., for a `method_col`). No copying is done, so this method is fast.

For columns an output operator `<<` is defined, which depends on an output operator for the column's value type. This is sometimes source of an undefined operator error.

Typed column

In addition to the methods described above, the `typed_col` class defines additional methods useful for this class holding constant column values.

`set`, `set_default`: These methods explicitly set the contents of the column to the value passed.

`get_val`: If a constant reference to the column value is sufficient, this method is very useful, since it does not perform any implicit copy operations.

explicit constructor: There is an explicit constructor for setting the value of the column during construction.

Method column

For the method column there is only the explicit constructor, which builds the column class from a column evaluation method.

Column Evaluation Methods

A column evaluation method, which can be stored inside a `method_col`, must be a function object and subclass of `method`. One operator and two methods have to be provided for the class. The `def` method returns the value when evaluated in a default value context, and the evaluation operator `operator()` returns the value when evaluated in a row context. Finally, the `set_context` method is used to store the evaluation context and the evaluation row.

5.2.4 Rows

The VDBL rows (`class vdbl::row`) are internally defined as STL maps of columns, organized with **column id** keys and column entries.

In principle, different types of rows could be defined, but at the moment only standard rows are implemented.

Every row contains a number of columns, whose values can be retrieved within an evaluation context (see Section 5.2.7). A column can contain an arbitrary type encapsulated in a `col` class, see Section 5.2.3. If you want to make sure, that type checking is used, you have to change the rows through the table methods.

The following methods are available in the `row` class.

`get_col`: These method return a (constant) reference to the column with the given `colid`. If the column does not exist, an empty column is returned, and the `error` parameter is set to `true`.

`has_col`: The check, whether the row contains a column for the provided column id, is performed by calling this method.

`insert`: This method inserts a new column with given id and value into this row. It returns `true` if the column was successfully inserted.

`drop`: Deleting a column from a row is done by a call to this method. The column's id is provided as the only argument. The return value is `true`, if erasing was successful.

`update`: This method replaces the value of the column, whose id is passed as first parameter, with a copy of the second parameter. If the column is not yet defined, `update` works like `insert`. If updating was successful, the method returns `true`.

5.2.5 Views

A **view** (`class vdbl::view_base`) onto a table is table-like construct built from table structures. They may be restricted to a subset of the rows and/or columns of the table.

The most important properties of a view is that it is always created within a given context (see Section 5.2.7). The contents of the view can vary depending on this context. Two different views to the same table can at the same time show different data in the same column of the same row.

Two different classes of views have been implemented:

Standard View: This view (of `class vdbl::view`) is constructed over **one** table, and it can only be restricted to subsets of rows and columns of this table.

Hierarchical View: An hierarchical view (in `class vdbl::hierarchical_view`) looks onto a **stack of tables**, the top ones "overlaying" the lower ones. This makes it possible to have, e.g., a globally valid table on bottom and a stack of locally valid tables on top of them.

Some views can hold an internal **cache** of table entries, which are used for fast access, reducing the number of calls to function objects within columns.

The method description is split into three parts. The first part explains those defined for all view types, and the other parts describe the special methods of the `view` class and of the `hierarchical_view` class, respectively.

Every view has associated one of the following view properties.

`V_window`: This view looks onto a table. It is possible to change the table contents through the view.

`V_transparent`: This view does not change the underlying table. It can be expanded, but changes are not committed to the table.

`V_frozen`: This view is a constant view to a table. It does not change and cannot be changed.

`V_materialized`: The view is the result of a select, and there is not an underlying table.

`V_independent`: The view is just a temporary collection of rows and columns without a table.

General Views

All views are subclasses of the base class `viewbase`. This class already defines a series of methods which I will describe below.

`insert`: This method inserts a new row into the view. Rows are built like for tables (see Section 5.2.2). The return value is `true`, if inserting was successful.

`remove`: The only parameter of `remove` specifies the row to be deleted. This parameter is a pair. The first component defines the table id from which the row has to be removed. The second component specifies the row. If the row was successfully erased, the return value is `true`.

`has_col`: This method returns whether the view contains a column of the name given.

`get_col_id`: The table id of the defining table of this column and the column id are returned by this method. The only input parameter is the name of the column. If the column does not exist, the pair `std::make_pair(tableid(),colid())` is returned, which is impossible for a real column.

`get_row`: This method returns a constant reference to the row specified by the input parameter, which is like in `remove`. If the row does not exist, an empty row is returned, and the `error` parameter is set to `true`.

`get_raw_col`: The first parameter of this method defines the row like in `remove`, and the second one the id of a column. The method returns a reference to the column defined by these coordinates, and in addition it returns a pointer to the row containing the column through the third parameter. In this method, the evaluation context for which the view was created is `not` passed to the column, so the column's raw data is returned. If the row does not exist, the `error` parameter is set to true, and the return value is a reference to an empty column.

`print_col`: This method prints a column. Its first parameter is the output stream, and the second and third parameters define a column like in `get_raw_col`. The last parameter is set to `true`, if something was printed.

`get_def`: This one returns the default value of a column. If successful, `error` is set to `false`, otherwise it is set to `true`.

`view_type`: A call to this method returns the view's type.

In addition to the methods described above, the class provides three iterators, a `col_const_iterator` for iterating through all columns, and a `row_const_iterator` for iteration over the rows, and a `default_const_iterator`, which in sequence visits all default values.

The default and column iterators dereference to a constant `col` reference for the default or column they are pointing at. The row iterator dereferences to a constant `row` reference.

All iterators have an `id` method returning the `colid` or `rowid` of their position, and the default iterator in addition has a `tid` method which returns the `tableid`.

The methods `row_begin`, `row_end`, `col_begin`, `col_end`, `default_begin`, and `default_end` return iterators for the begin and beyond the end of the respective lists.

Standard View

There are two constructors for the `view` class, the standard view of the VDBL. The first one takes the table id and a pointer to the table, onto which it looks. An evaluation context and the view type complete the input parameters. The second constructor in addition takes a list of `rowid`s which should be visible in the view. In that case, all rows *not* given in the list remain hidden.

There are two new methods, which come in a number of similar variants. The first one `get` retrieves the data of a column within the evaluation context. The `tableid` and `rowid` have to be specified and in addition either the name or the `colid` of the column. The methods return `true`, if the `get` was successful.

By `get_raw_ptr` a constant pointer to the data of a column is returned. This is only useful for columns with constant value. Since no copy constructor is called, the method is fast.

Hierarchical View

For a hierarchical view the same methods as for a standard view are defined. The constructor also looks the same, however its semantics is a bit different. It defines the *master table* of the view, which determines column name–column id mappings and always defines the bottom of the view. The master table cannot be changed, and it cannot be removed.

In addition two `push_table` methods are defined, which add a new table on top of the hierarchical stack. The first version takes a `tableid` and a pointer to the table, adding all rows, while the second one takes a third parameter, a list of `rowid` which should be visible in this table, all other rows of the table are hidden.

The last method `pop_table` removes the topmost table from the hierarchical stack and it returns its `tableid`.

5.2.6 View Database

A **view database** is a view onto a complete database, automatically constructing views (standard or hierarchical) for every table defined in the database. These views can be accessed under the same names as the defining tables, having a subset of their columns (also with identical names). The defining class is `vdbl::viewdbase`.

There are two non-standard constructors for a `viewdbase`. The first variant defines a view database from a `database` (see Section 5.2.1) and an evaluation `context` (see Section 5.2.7), which will be valid for all constructed views. In addition, the `userid` of the user constructing the view database has to be passed. The second variant of the constructor has a fourth parameter, an arbitrary STL sequence container of `tableid`–`rowid` pairs, which should be visible in the view database. All rows from all tables, which are not contained in the list remain hidden from view.

The `get_table_id` method returns the `tableid` associated to a given table name. If no such table exists, `tableid()` is returned, which is an invalid table id.

All variants of the `has_view` method check whether a view to a given `tableid` or name exists.

Finally, the `get_view` methods return a pointer to the view of the given id or name. If no view of that name or id exists, `NULL` is returned.

5.2.7 Contexts

Evaluation contexts are subclasses of the `vdbl::context` class. They may hold arbitrary data and are keeping a `const vdbl::table *` to their associated table.

This context is passed to every function object along with the row the column belongs to for constructing the columns value. The contexts have no influence on `typed_col` columns, whose values don't change within different contexts.

The only methods for this class are both called `table`. Without arguments it returns the constant pointer to the associated table, and if such a pointer is passed to the table method, the associated table is changed to the new one.

5.3 The API

This first part of the API is responsible for the structural framework of the program, for the data encapsulation, and for the communication between the different modules.

After reviewing the basic classes needed throughout most of the environment (Section 5.3.1), we start with the description of the expression class and the basic operator types, which are used for representing the functions of the optimization problems (Section 5.3.2).

5.3.1 Helper Classes

Three helper classes form the backbone of the communication. The basic data types which can be transferred between the various models of the environment are kept in a single union `basic_alltype`, a kind of "all-type".

The base class for all data transfer (`datamap`) is in principle a map of `string` to `basic_alltype`. Subclasses of this type are used for communication between the strategy engine and all the modules.

Database rows are kept in the class `dbt_row`, which is primarily used by the deltas (see Section 5.5) for changing the model annotations (see Section 5.4.2).

Basic Types

The class `basic_alltype` is a union of six scalar types, five vector types, and three matrix types. For compatibility reasons a second typename `additional_info_u` can be used for the basic alltype. However, the use of the older typename is deprecated, and support for it may be removed in a future version.

The following types can be stored. They are listed together with their retrieval operation and their type identifier.

/* *empty* */	/* *none* */	ALLTYPE_EMPTY
`bool`	`nb()`	ALLTYPE_BOOL
`int`	`nn()`	ALLTYPE_INT
`unsigned int`	`nu()`	ALLTYPE_UINT
`double`	`nd()`	ALLTYPE_DOUBLE
`interval`	`ni()`	ALLTYPE_INTERVAL
`string`	`s()`	ALLTYPE_ALLOCED_S
`vector<bool>`	`b()`	ALLTYPE_ALLOCED_B
`vector<int>`	`n()`	ALLTYPE_ALLOCED_N
`vector<unsigned int>`	`u()`	ALLTYPE_ALLOCED_U
`vector<double>`	`d()`	ALLTYPE_ALLOCED_D
`vector<interval>`	`i()`	ALLTYPE_ALLOCED_I
`matrix<int>`	`nm()`	ALLTYPE_ALLOCED_NM
`matrix<double>`	`m()`	ALLTYPE_ALLOCED_M
`matrix<interval>`	`im()`	ALLTYPE_ALLOCED_IM

Every type, which can be stored in a basic alltype, can be assigned to it, and a `basic_alltype` object can be (automatically) constructed from every storable type. The corresponding assignment operators and constructors are defined. However, for the assignment operators the vector and matrix types are passed as pointers to avoid unnecessary calls to the copy constructor of (probably big) complex types. Strings can be passed as `string` objects or as pointers to `C` strings (`const char *`).

Additional important methods are `clear`, which deletes the contents of a `basic_alltype` and leaves it empty, and the `empty` method can be used for testing, whether the basic alltype is empty. With `contents_type` the type of the stored data can be retrieved.

Finally, there are four methods for boolean queries about the stored data. `is_allocated()` returns true if the alltype contains any data, for which memory allocation is necessary, i.e. the vectors, matrices, and strings. The `is_vector`, `is_matrix`, `is_scalar` return, whether the data is a vector, a matrix, or a scalar, respectively.

Datamap

The basic structure of the classes for data communication between modules is the `datamap` class. It is based on a map from `C++` strings to the `basic_alltype` class. It is enhanced by a variety of methods making the data handling more convenient for the model developer.

The data map contains two different classes of entries. The *simple entry* consists of a key given by a string and its data entry, and the *parametrized entry* has in addition to the key string an integer *subkey* and a data entry for every defined key–subkey pair. Strings can be entered in all methods either as `C++` strings or as `C` strings.

The `sinsert` methods add a new data entry into the datamap. If the `replace` variable is set to `true`, new entries for the same key (key–subkey pair) replace older ones. If the variable is set to `false`, inserting a value for an already defined key (key–subkey pair) fails. The methods return `true` if they have successfully inserted the new values into the data map.

With `sfind` the value in the data map to a given key (key–subkey pair) can be retrieved. If the entry is undefined, an empty `basic_alltype` is returned.

The `remove` methods delete the entry for a given key (key–subkey pair) from the data map, and the `defd` methods check, whether the data map contains an entry for the key (key–subkey pair) passed.

For a key with integer subkeys the `which` methods can be used to retrieve the subkeys, for which an entry is defined. After the call the vector `idx` contains the defined subkeys, and the function returns `true` if the key passed is indeed defined as a key with subkeys.

In addition to the methods described above there is a huge number of `retrieve` (`retrieve_i`) methods. They come in two flavors, with or without default entry. The first (first two) argument(s) of the methods specify the key (key–subkey pair) for which the value should be retrieved. The next argument is a reference to a variable of one of the basic scalar types and a reference to a pointer to one of the complex basic types. This variable is set by the method to the data entry corresponding to the key (key–subkey pair). If no entry exists for the the key (key–subkey pair) presented, the method without default returns `false`, and the method with default sets the variable to the value of the last method argument.

Database Tools

The type `dbt_row` is used by all modules which need to generate new database entries. First an object of that class is generated, and then the database row is constructed column by column with calls to `add_to_dbt_row`, which takes a `dbt_row` as its first argument, a column name as its second, and the column value as its third argument. Column names can be given as `C++`- or as `C` strings.

Global Pointers

In order to prepare the API for distributed computing, in certain positions the local pointers have to be abstracted. This is done by the global pointer class `gptr`. It is a templated class defining an abstraction of a pointer to a prespecified type. The abstraction is performed in such a way that a `const gptr` correctly abstracts a const pointer. In addition to the operators `*` and `->`, which work like for ordinary pointers, there is one additional method `get_local_copy`, which returns an ordinary pointer to a local copy of the object referenced by the global pointer.

At the moment there is only one subclass of gptr defined, the global abstraction of local pointers. The ptr class emulates a global pointer by a local pointer, and using it poses no significant performance penalty compared to a normal pointer. For the ptr class there is no difference between the operator * and the get_local_copy method.

5.3.2 Expressions

As we have seen, every model in the search graph is of the form

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } F(x) \in \boldsymbol{F} \\ &\qquad x \in \boldsymbol{x}. \end{aligned}$$

When represented in the environment the functions $f : \mathbb{R}^n \to \mathbb{R}$ and $F : \mathbb{R}^n \to \mathbb{R}^m$ have to follow the following requirements:

1. The function F is a vector of functions $F_i : \mathbb{R}^n \to \mathbb{R}$.
2. All functions f and F_i are compositions of a finite number of elementary expressions.

The expressions representing the functions are built using a directed acyclic graph (DAG) where every node represents an operation and the result of an expression at the same time. The operator_types admissible are listed below. Every child of an expression is multiplied by its corresponding entry in the coeffs vector before it is used in the expression. So the value x_i used in the expression definitions below already contains the factor from the coeffs array. If the value y_i is used in the description, you can safely assume that the coeffs-entry is 1.

The methods and public variables of the expression_node class are

node_num: The node number of this expression node. This number is unique throughout a DAG and all DAG deltas, so it can be used for node referencing.

operator_type: This entry describes the operator represented by this node. The different possibilities for operators are described below.

coeffs: The coefficients array contains the multipliers for the children in the same sequence as they ares sorted as children.

params: This entry contains additional parameter information for the node. From a single integer to a matrix of intervals everything is possible. The parameters for the various operator types are describen below.

f_bounds: If this node belongs to a constraint, this entry holds the bounds. In this version of the API, the bounds are represented by an interval.

sem: This entry describes additional semantics information for a node. This information is described below in more detail.

n_parents, n_children: Those two members contain the number of parents and children, respectively, that this node has.

v_i: The variable indicator `v_i` describes in form of a bitmap the variables, this node depends on. Since the variable indicators are mostly used in an evaluator context, a more detailed description of this class is contained in Section 5.6.

In addition to the public class members above, there are a number of public methods for manipulating objects of the `expression_node` class.

var_indicator: This is the `const` accessor for the `v_i` entry, the variable indicator.

set_bounds: This method can be used for changing the bounds on the expression node. It is important to note that this does not implicitely make an expression node to a constraint (see the description of the `model` class in Section 5.3.3 for further information on that).

add_is_var: For reinterpreting nodes as variables in a DAG, the variable number has to be set with this method.

rm_is_var: This method removes a variable number from the expression node. If all the variable numbers are removed, the node is not regarded as a variable any longer.

is: This method tests the semantics of an expression node. The following enum entries can be used for testing. If more than one property should be tested in one call, the corresponding entries have to be combined with the `|` operator. If alternatives are to be tested, two calls to `is` have to be made, whose results are combined with `||`. The enum entries and their meaning are described in the list below.

ex_bound: the node is a simple bound constraint, i.e., a single variable,

ex_linear: it is linear but not a simple bound constraint,

ex_quadratic: the expression is nonlinear quadratic,

ex_polynomial: it is polynomial of higher than second degree,

ex_other: it is non-polynomial,

ex_atmlin: the node is at most linear,

ex_atmquad: it is at most quadratic, i.e., quadratic or linear,

ex_atmpoly: the expression is any polynomial,

ex_nonlin: it is nonlinear,

ex_nonbnd: the node represents an expression which is not a simple bound,

ex_any: this matches any expression,

ex_kj: the node was added because it is part of the Karush-John first order optimality conditions (see Section 2.2.3),

ex_org: the node is not only due to the Karush-John conditions,

ex_redundant: the constraint represented by this node is redundant (e.g., a cut or a relaxation), `ex_notredundant`: the constraint is not redundant,

`ex_active_lo`, `ex_active_hi`: the constraint has possibly active lower (upper) bound,

`ex_inactive_lo`, `ex_inactive_hi`: it has inactive lower (upper) bound,

`ex_active`, `ex_inactive`: the constraint is possibly active (definitely inactive) at any bound,

`ex_integer`: this node is integer valued — if this is set for a variable, the variable is an integer variable,

`ex_convex`: the function represented by this expression is convex and nonlinear with respect to the variables (this does not mean that a constraint represented by this node is convex, this depends in addition on the bounds of the constraint),

`ex_concave`: the function represented by this expression is concave and nonlinear with respect to the variables,

`ex_inequality`: this constraint is an inequality constraint,

`ex_equality`: the constraint is an equality constraint,

`ex_leftbound`: this constraint has only a lower bound,

`ex_rightbound`: it has only an upper bound,

`ex_bothbound`: both sides are bounded.

Semantics

An important part of every expression node is the semantics information, which can in part be questioned by the `is` method of the expression node class. There are three classes of flags and additional entries.

`property_flags`: The list of property flags describes information which typically changes during the algorithm, since it is in part dependent on the domain of the node's variables.

`c_info`: the convexity information of the node,

`act`: the activity information for the bounds of the node,

`separable`: whether the node is separable,

`is_at_either_bound`: is the value of this node either at is upper or at its lower bound but not in between.

`annotation_flags`: Annotation flags are set during the construction of the node. They specify properties, which are fixed by the structure of the expression.

`kj`: The node was added during the generation of the Karush-John first order optimality conditions.

`integer`: The node is integer valued.

`type`: This expression node belongs to one of the following four types, which will be used in the future algorithmic setup of the code: `v_exists`, `v_forall`, `v_free`, and `v_stochastic`. At the moment, only the `v_exists` specifier is used.

`hard`**:** The constraint represented by this node is a hard constraint and not a soft constraint.

`info_flags`**:** These flags describe algorithmic properties of the nodes. At the moment there is only one flag defined, `has_0chnbase` denotes whether the expression represented by this node is a one dimenensional function of one node, the 0-chain base.

`_0chnbase`**:** This is the node number of the 0-chain base, if it exists, and of the node's node number, otherwise.

`addinfo`**:** For Karush-John variables, i.e., Lagrange multipliers, this entry is the corresponding constraint number. If it is the multiplier of the objective function, the value of this member is -1.

`degree`**:** This is the polynomial degree of the expression with respect to the variables, or -1 if it is non-polynomial.

`dim`**:** The dimension of the expression, i.e, the number of variables it depends on is contained in this entry.

`stage`**:** This member denotes for multistage problems the stage to which this expression belongs.

There are numerous methods for retrieving and setting the various entries of the semantics entries. They can be found in Appendix A.3.2.

Expression Nodes

There is a number of expressions with real result:

`EXPRINFO_GHOST`**:** This is a virtual expression needed for `dag_delta`s. Ghost nodes are exceptions to the rule that node numbers have to be unique throughout all models in a model group. The node number of a ghost node is the same as the node number of the node, of which it is a ghost.

`EXPRINFO_CONSTANT`**:** The value of the constant is contained in `params.nd()`.

`EXPRINFO_VARIABLE`**:** The variable number is contained in `params.nn()`.

`EXPRINFO_SUM`**:** This is a general linear combination of the form

$$\sum_{i=1}^{n} x_i + c$$

where c is stored in `params.nd()`.

`EXPRINFO_MEAN`**:** This is a sum node where it is guaranteed that all coefficients are positive, and their sum is 1. So this is a convex linear combination without constant term:

$$\sum_{i=0}^{n} x_i.$$

EXPRINFO_PROD: The product node has the form

$$c\prod_{i=0}^{n} y_i$$

where c is in `params.nd()`.

EXPRINFO_MAX: The maximum node is defined as

$$\max(c, x_i \quad i = 0, \dots, n)$$

with c in `params.nd()`.

EXPRINFO_MIN: The minimum node is defined as

$$\min(c, x_i \quad i = 0, \dots, n)$$

with c in `params.nd()`.

EXPRINFO_MONOME: The general monomial node is defined as

$$\prod_{i=0}^{n-1} x_i^{n_i}$$

where the n_i are contained in the vector `params.n()`.

EXPRINFO_SCPROD: The scalar product is defined as

$$\sum_{i=0}^{k-1} x_{2i}x_{2i+1}$$

without parameters.

EXPRINFO_NORM: The norm node has the form

$$\|x\|_k$$

where k is stored in `params.nd()`, and x is the vector built from the values of all children.

EXPRINFO_INVERT: Inversion is defined as

$$a/y_0$$

with a in `params.nd()`.

EXPRINFO_SQUARE: The square node is

$$(x_0 + q)^2$$

where q is stored in `params.nd()`.

EXPRINFO_SQROOT: The square node is

$$\sqrt{x_0 + q}$$

where q is stored in `params.nd()`.

`EXPRINFO_ABS`: The absolute value node is

$$|x_0 + q|$$

where q is stored in `params.nd()`.

`EXPRINFO_INTPOWER`: The integer power node is

$$x_0^n$$

where the integer n is in `params.nn()`.

`EXPRINFO_EXP`: The exponential node is

$$e^{x_0+q}$$

where q is stored in `params.nd()`.

`EXPRINFO_LOG`: The logarithm node is

$$\log(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_SIN`: The sine node is

$$\sin(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_COS`: The cosine node is

$$\cos(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_GAUSS`: The gaussian function node is

$$e^{-(x_0-q_0)^2/q_1^2}$$

where the two dimensional vector q is stored in `params.d()`.

`EXPRINFO_POLY`: The polynomial node is

$$\sum_{i=0}^{n} \alpha_i x_0^i$$

where the coefficients α_i are stored in `params.d()`.

`EXPRINFO_POW`: The general power node is

$$(x_0 + p)^{x_1}$$

with p in `params.nd()`.

`EXPRINFO_DIV`: The general division node is

$$a * y_0/y_1$$

and a is stored in `params.nd()`.

`EXPRINFO_ATAN2`: The atan2 node has no parameters, and its definition is

$$\texttt{atan2}(x_0, x_1) = \text{atan}(x_0/x_1).$$

`EXPRINFO_LIN`: This node is a linear combination of **all variables**. In `params.nn()` the row number in the matrix of linear constraints is stored.

`EXPRINFO_QUAD`: This node is a quadratic form in **all variables**. The `params.m()` stores the enhanced matrix $(A|b)$ for the function

$$x^T Ax + b^T x$$

A is assumed symmetric and sparse.

`EXPRINFO_IF`: We get the interval $\boldsymbol{x}$ from `params.ni()` and define

$$n = \begin{cases} x_1 & \text{if } x_0 \in \boldsymbol{x} \\ x_2 & \text{otherwise.} \end{cases}$$

The next operators have discrete results:

`EXPRINFO_IN`: The point defined by $x = (x_i)$ all children is tested against the box $\boldsymbol{x}$ stored in `params.i()`. If $x \in \text{int}\,(\boldsymbol{x})$ the value is 1, if $x \in \partial\boldsymbol{x}$, the return value is 0, and if $x \notin \boldsymbol{x}$ the node's value is -1.

`EXPRINFO_AND`: The value is 1 if all $x_i \in \boldsymbol{x}_i$ where the $\boldsymbol{x}_i$ are stored in `params.i()`, and 0 otherwise.

`EXPRINFO_OR`: The value is 0 if all $x_i \notin \boldsymbol{x}_i$ where the $\boldsymbol{x}_i$ are stored in `params.i()`, and 1 otherwise.

`EXPRINFO_NOT`: The value is 0 if $x_0 \in \boldsymbol{x}$ and 1 otherwise. Here $\boldsymbol{x}$ is in `params.ni()`.

`EXPRINFO_IMPLIES`: The value is 1 if the following is true:

$$x_0 \in \boldsymbol{x}_0 \quad \Longrightarrow \quad x_1 \in \boldsymbol{x}_1$$

where the $\boldsymbol{x}_i$ are stored in `params.i()`. Otherwise, the value is 0.

`EXPRINFO_COUNT`: The count node returns for how many i we have $x_i \in \boldsymbol{x}_i$, where the intervals $\boldsymbol{x}_i$ are stored in `params.i()`.

`EXPRINFO_ALLDIFF`: This node returns 1 if all children are pairwise different. Here different means

$$|x_i - x_j| > \delta$$

with δ in `params.nd()`.

`EXPRINFO_HISTOGRAM`: The histogram node counts how many children x_i are in which interval $\boldsymbol{x}_{2k}$. The numbers n_k are collected, and if for all k the value $n_k \in \boldsymbol{x}_{2k+1}$, the result is 1. Otherwise, it is 0.

`EXPRINFO_LEVEL`: The matrix A in `params.mi()` contains a set of boxes $A_{i:}$ contained within another. The result is the smallest row index j such that the vector $x = (x_i)$ is contained in $A_{j:}$.

`EXPRINFO_NEIGHBOR`: This returns 1 if x_0 and x_1 are neighbors, i.e., if (x_0, x_1) is in the list of allowed value pairs (v_{2k}, v_{2k+1}) where the vector v is stored in `params.n()`. The x_i and v_i are integers.

`EXPRINFO_NOGOOD`: This returns 1 if $x = v$ with v in `params.n()`. Both $x = (x_i)$ and v are integer vectors.

There are two integration nodes:

`EXPRINFO_EXPECTATION`: integrates over all variables in the subgraph over all $\mathbb{R}$.

`EXPRINFO_INTEGRAL`: integrates over all variables in the subgraph, for which the integration area defined in `params.i()` is a non-empty interval. Variables with empty interval are considered free.

The following highly complex operations implicitly build matrices, and all use `params.nm()` to hold the parameters. Special is `params.nm[0]`, because it contains the noted info. If the matrix is sparse, the remaining rows contain column index information.

`EXPRINFO_DET`: The determinant of the matrix formed by the child nodes. `params.nm()(0,0)` specifies whether the matrix is dense (1) or sparse (0), `params.nm()(0,1)` specifies the dimension of the matrix.

`EXPRINFO_COND`: The condition number of a matrix formed by the child nodes. `params.nm()(0,0)` specifies whether the matrix is dense (1) or sparse (0), `params.nm()(0,1)` and `params.nm()(0,2)` specify the dimension of the matrix, and `params.nm()(0,3)` deterimines which condition number is calculated. Here 0 is for ∞.

`EXPRINFO_PSD`: This is 1 if the matrix is positive semidefinite and 0 otherwise. The `params` are as in `EXPRINFO_DET`.

`EXPRINFO_MPROD`: The value of this node is

$$\|Ax - b\|_2^2,$$

where the nodes define sequentially A, b, x, and the `params` define the matrix dimensions as in `EXPRINFO_COND`.

`EXPRINFO_FEM`: The value of this node is

$$\|A^T DAx - b\|_2^2,$$

where the nodes sequentially define A, D, b, x, and the `params` define the matrix dimensions as in `EXPRINFO_COND`.

Finally, there are two nodes containing constant matrices:

`EXPRINFO_CMPROD`: The node returns the value

$$\|Ax - b\|_2^2,$$

where the matrix $(A|b)$ is stored in `params.m()`.

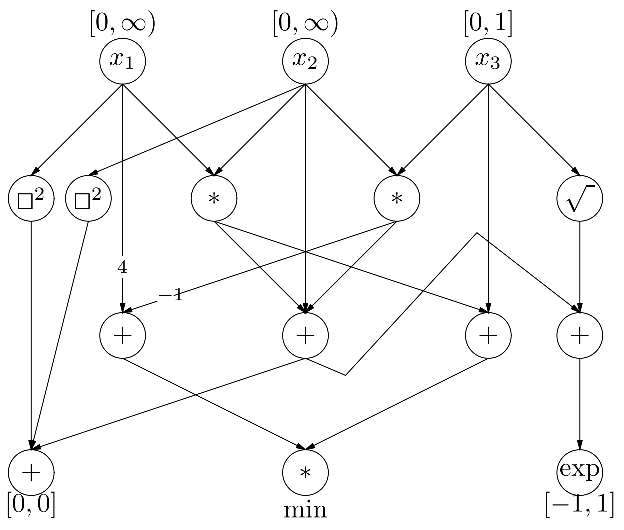


Figure 5.2. *DAG representation of problem (5.1)*

`EXPRINFO_CGFEM`**:** The node returns the value

$$\|A^T DAx - b\|_2^2,$$

where the matrix $(A|b)$ is stored in `params.m()`, and the nodes sequentially define D, x.

5.3.3 Models

The `model` class stores one optimization problem, which is passed to the various inference engines. The class is a subclass of `dag<expression_node>`, so the optimization problem is stored in form of a DAG. This representation has big advantages, see Section 4.1 for a detailed description.

A complete optimization problem is always represented by a *single* DAG of expression nodes, as explained in Section 5.3.2. Consider for example the optimization problem

$$\begin{aligned} &\min\ (4x_1 - x_2x_3)(x_1x_2 + x_3) \\ &\text{s.t.}\ \ x_1^2 + x_2^2 + x_1x_2 + x_2x_3 + x_2 = 0 \\ &\qquad e^{x_1x_2 + x_2x_3 + x_2 + \sqrt{x_3}} \in [-1, 1] \\ &\qquad x_1 \geq 0, x_2 \geq 0, x_3 \in [0, 1]. \end{aligned} \tag{5.1}$$

This defines the DAG depicted in Figure 5.2.

This DAG is optimally small in the sense that it contains every subexpression of objective function and constraints only once.

This section is grouped into three parts, each describing an important part of the model structure. Models are internally represented by a hierarchy of classes as in Figure 5.3. The topmost class `model_iddata` holds all data which has to be unique

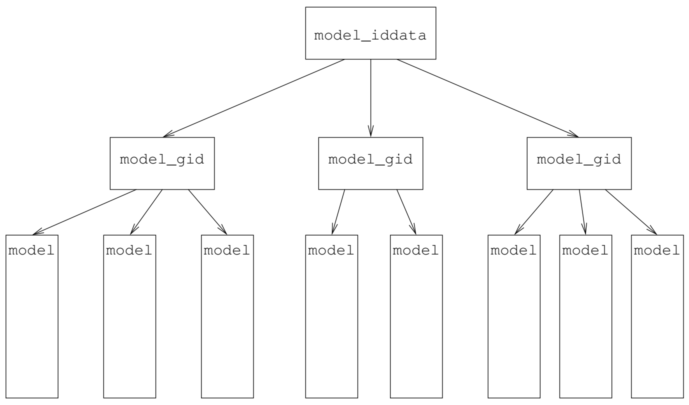


Figure 5.3. *Hierarchy of model classes*

througout all models, like the node numbers, variable numbers, and the like. For every set of models, which represent the same mathematical problem there is one `model_gid` structure. All models within one model group can be directly combined with each other. There is a common global node, variable, and constraint reference, and the like. Within every model group there are the individual models, each defined by one object of the `model` class.

Id Data

The `model_iddata` class is the topmost in a hierarchy of model classes. It represents the important global information, manages the free and used numbers for nodes, variables, and constraints. There is usually no need to directly access members or methods of this class. The class is managed automatically by the various methods of the `model_gid` and `model` classes.

Model Group Data

Models are grouped into sets, which can be directly combined with each other, like models and model deltas. In this class global references to nodes, variables, constants, and back references from constraint numbers to constraints are managed. Furthermore, the reference model, the main model to which all the deltas belong, is stored.

The most important methods in this class are

`remove_..._ref`: These methods (insert `node`, `var`, or `const`) remove the global references to the corresponding objects.

`mk_globref`, `mk_gvarref`, `mk_gconstref`: Global references to nodes, variables, or constants are constructed by these methods. The `mk_gconstref` method in addition adds the constraint back reference entry for the new constraint.

`empty`: This method returns `true` if the walker is the ground node of the reference model, which is used to represent empty entries in the various reference arrays.

`its_me`: With this method, a model can check, whether it is the reference model or not.

`empty_reference`: This method returns the ground node of the reference model, used as empty entry in the reference arrays.

`have_..._ref`: For `glob`, `gvar`, and `gconst` these methods check whether there is an object in this model group corresponding to this number.

Models

A model is a directed acyclic graph of expression nodes (see Section 5.3.2) plus additional management information.

The class has a few very important public members:

`ocoeff`: This is the sign of the objective function. If `ocoeff` is 1, the problem is a minimization problem, if it is -1, the optimization problem is a maximization problem, and if it is 0, it is a constraint satisfaction problem. For evaluation purposes, the objective function has to be multiplied with this constant.

`objective`: This walker points to the final node of the expression representing the objective function. An evaluator (see Section 5.6) has to use this walker as the starting point for evaluating the objective function.

`constraints`: This vector of walkers contains the pointers to the expressions representing all constraints, which are valid in this model.

There is a huge number of public methods defined for the `model` class. The constructors need some attention, as well.

- The constructor `model(model_gid* id = NULL, bool clone=false)` constructs a new model. If a `model_gid` object is passed, the new model receives that `model_gid` if `clone` is `true` and a copy of the `model_gid` if `clone` is `false`.
- The `model(model_gid* id, const erased_part& ep, bool clone=false)` constructor works like the one above, except that it does not construct an empty model but a model preinitialized with a DAG, which was previously erased from a bigger DAG.
- An empty model prepared for a prespecified number of variables is built by the constructor `model(int num_of_vars)`.
- In addition to the simple copy constructor there is an enhanced copy constructor `model(model_gid* id, const model& m);` which generates a copy of the model `m` within the model group `id`.

- The constructors `model(const char* name, bool do_simplify = true)` and `model(istream\& name, bool do_simplify = true)` read a DAG description, either from file `name` or from input stream `inp`. The `do_simplify` parameter specifies, whether the simplifier should be called for the model after reading it.

The simplifier can be applied to any model by calling the `basic_simplify` method. If a model is freshly constructed and the simplifier is not called, at least the `set_counters` method must be applied, and `arrange_constraints`, which sorts the constraints by complexity is useful, too, especially if there is need to iterate over subsets of constraints.

If by insertion and deletion the constraint numbers, variable numbers, or node numbers have become fragmented, they can be rearranged and compressed by calling the `compress_numbers` method. This *must* be done in a situation where no `work_node` exists, since simultaneously all models in all model groups are updated but the caches of the work nodes are not invalidated.

A model can be detached from a model group and put as reference model into a newly created model group by calling the `detach_gid` method.

The `write` method produces a serialization of the model on a specified `ostream` or to `cout` if no stream is provided.

Some methods are used for retrieving information on the model.

`number_of_...`, `number_of_managed_...`: These methods return the number of `nodes`, `variables`, and `constraints` which are defined globally across all models and locally in this model, respectively.

`var`: This returns a walker pointing to the node for the variable of the specified number.

`node`: With this method a walker pointing to the node with the specified node number can be retrieved.

`constraint`: This method returns a walker pointing to the constraint with the specified number.

`get_const_num`: The number of a constraint can be retrieved by this method if the node number of the constraints expression node is known. The method returns the constraint number via reference in the second parameter. The return value is `false` if no constraint exists for the specified node number.

`gid_data`: With this method the model group data structure for this model can be accessed.

`is_empty`: This method returns `true` if the model does not contain a single node.

The methods `store_node`, `store_variable`, `store_constraint`, and `store_ghost` create local and model group global references of the respective kind for the expression nodes referenced to by the provided walkers.

Nodes can be removed from all references, the ghost, variable, and constraints local and global reference arrays, by calling the three variants of the `remove_node` method.

A number of methods directly access the `model_gid` and `model_iddata` objects to retrieve additional information:

`var_name`: returns the name of the variable referred to by the supplied number,

`const_name`: provides the name of the constraint for the specified number,

`obj_name`: retrieves the name of the objective function,

`obj_adj`, `obj_mult`: the additive and multiplicative constants for the objective function (The real value of the objective function is

$$f_{\text{real}}(x) = c\,(m\,f(x) + a)$$

where $c =$ `ocoeff`, $m =$ `obj_mult`, and $a =$ `obj_adj`.),

`n_fixed_vars`, `fixed_var`: the number of fixed variables and their value,

`n_unused_vars`, `unused_var`: the number of unused variables, which are specified but not connected to the model,

`n_unused_constrs`, `unused_constr`: the number of unused constraints.

Finally, there are numerous methods for constructing new expressions within a model:

`constant`: This method constructs a node for the specified constant and stores it in the model.

`variable`: If the variable for the specified number exists already, a walker to its node is returned. Otherwise, a new variable is created, stored, and a reference is returned.

`ghost`: A new ghost is created for this node number if no such ghost exists. Then the method returns a walker pointing to the ghost for the provided node number.

`unary`: This method constructs a unary expression, whose child is the first argument. The second argument is the operator type. The optional third argument specifies the additional parameters needed for describing the expression, and the last argument, which is also optional, specifies the multiplier for the child, which is stored in the `coeffs` array (see Section 5.3.2). The return value is a walker pointing to the newly constructed expression.

`binary`: For constructing a binary operator this method can be used. Its first two parameters are the two children, and the third parameter is the operator type. The optional fourth parameter describes the additional parameters for the expression. The multiplicative coefficients for the children are passed via the optional parameters five and six. The method returns a walker pointing to the newly constructed expression.

`nary`: This method constructs a general n-ary expression. The first parameter is a vector of operands, the second one specifies the operator type. Additional parameters for the expression can optionally be passed through the third parameter, and the last parameter is the `coeffs` array of the new expression. A walker pointing to the new expression is returned.

`vnary`: For simple expressions without additional parameters and where alle coefficients of the children are 1, this variable number of arguments method can be used. After the operator type a list of pointers to expression node walkers have to be passed. The list must end with a `NULL` pointer. The return value is a walker pointing to the new expression.

5.3.4 Control Data

When data has to be passed to modules, regardless whether they are inference modules, management modules or report modules, it has to be passed via a `control_data` object. This class is a subclass of the `datamap` class described in Section 5.3.1.

The methods `set` correspond to the `datamap` methods `sinsert`, and it does *not* replace already existing entries.

The methods `get`, `is_set`, `unset`, and `which_set` correspond to the `datamap` methods `sfind`, `defd`, `remove`, and `which` methods, respectively.

The `assign` and `assign_i` methods work almost like `retrieve` and `retrieve_i`. However, the variants without default value raise an exception if the value to be retrieved is undefined instead of just returning `false`.

There are two additional methods defined for the `control_data` class. The `service` method takes a `string` parameter and sets the entry, which defines the service that should be provided by the module called. Inside the module the `check_service` method can be used to determine, whether the service has been set to the specified `string`.

5.4 Search Graph

The solution algorithm is an advanced branch-and-bound scheme which proceeds by working on the **search graph**, a *directed acyclic graph (DAG)* of search nodes (see Figure 5.4 and Section 5.4.1), each representing an optimization problem, a **model**. The **search nodes** come in two flavors: **full nodes** which record the complete description of a model, and **delta nodes** which only contain the difference between the model represented by the node and its (then only) parent. All search nodes "know" in addition their relation to their ancestors. They can be splits, reductions, relaxations, or gluings. The latter turn the graph into a DAG instead of a tree, as usual in branch-and-bound algorithms. The search graph is implemented using the Vienna Graph Template Library (`VGTL`), a library following the generic programming spirit of the `C++ STL` (Standard Template Library).

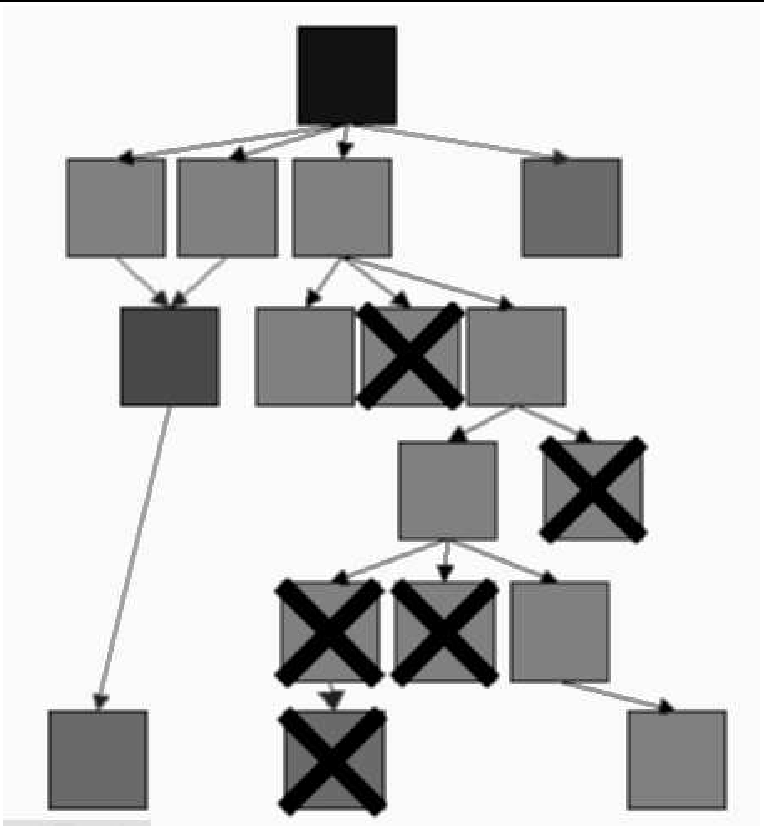


Figure 5.4. *Search graph*

A **reduction** is a problem, with additional or stronger constraints (**cuts** or **tightenings**), whose solution set can be shown to be equal to the solution set of its parent. A **relaxation** is a problem with fewer or weakened constraints, or a "weaker" objective function, whose solution set contains the solution set of its parent. Usually, relaxed problems have a simpler structure than their original. Typically *linear* or *convex* relaxations are used.

A problem is a **split** of its parent if it is one of at least two descendants and the union of the solution sets of all splits equals the solution set of their parent. Finally, a model is a **gluing** of several problems, if its solution set contains the solution sets of all the glued problems.

During the solution process some, and hopefully most, of the generated nodes will be solved, and hence become **terminal nodes**. These can be removed from the graph after their consequences (e.g., optimal solutions, . . .) have been stored in the search database. This has the consequence that the ancestor relation of a node can change in the course of the algorithm. If, e.g., all the splits but one have become terminal nodes, this split turns into a reduction. If all children of a node become terminal, the node itself becomes terminal, and so on.

The search graph has a **focus** pointing to the model which is worked upon. This model is copied into an enhanced structure - the **work node** (see Section 5.4.3).

A reference to this work node is passed to each inference engine activated by the strategy engine. The graph itself can be analyzed by the strategy engine using so-called **search inspectors**.

5.4.1 Search Nodes

The search graph is a directed acyclic graph of `search_node` pointers. The `search_node` class itself is not used in the graph, only its two subclasses `full_node` and `delta_node`.

The class `search_node` has a few *protected* members, which are essential for the `work_node` class described below.

`__global_model`: This member holds a global pointer reference (see Section 5.3.1) to the global (the original) model in the search graph.

`__dbase`: The search database can be referenced via this global pointer.

`_dbuser`: This member stores database userid used for all database accesses.

`_snr`: The relation between this search node and its parent is kept in this member. It can take the following values:

- `snr_root`: the root node of the search graph, i.e., the original problem,
- `snr_reduction`: a reduction of the parent, i.e., a problem which has the same solution set as its parent but a reduced search space,
- `snr_relaxation`: a relaxation of the parent, i.e., a problem whose solution set contains the solution set of the parent problem,
- `snr_split`: a split of the parent, i.e., one of two or more problems, all of which are also splits, such that the union of the solution sets of all split-problems is the solution set of the parent.
- `snr_glue`: a gluing of many parents, i.e., a problem which contains the union of the solution sets of all parents,
- `snr_worknode`: a work node, i.e., an enhanced node which is passed to the inference engines and not stored in the search graph,
- `snr_virtual`: a virtual node of the search graph, i.e., its `ground` or `sky`.

`_id`: This member stores the *unique* node id of the search node.

`_keep`: Since deltas and annotations for search nodes are stored in the search database, there has to be a method for removing these database entries when they are no longer needed. The `_keep` member defines which annotations are removed, when this search node is destroyed.

Most of the methods in the `search_node` class are accessors: `get_dbuser` for `_dbuser`, `global_model` for `__global_model`, `database` for `__dbase`, and `get_id` for `_id`.

The `is_delta` method is overloaded by the subclasses, and it is `true` for the `delta_node` and `false` for the `full_node` subclass.

For managing the annotations which are kept by the search graph there are methods `keep` and `unkeep`, which add and remove entries from the `_kept` container.

The `search_node` class has two subclasses: `full_node` which defines a node containing a complete model description, and `delta_node` a search node that stores only the difference to the parent node.

`delta_node`: The simpler search node is the delta node. It contains a list of delta references. The delta information is stored in the search database. Three methods exist in addition to the ones inherited from the `search_node` class. `n_deltas` returns the number of stored deltas. The method `get_delta_id` is used for retrieving the delta id of the ith delta, and `get_delta` returns the ith delta.

`full_node`: A full node contains a complete problem description, consisting of an expression DAG stored in the protected member `_m`, which is a global pointer to a `model`, and an array of annotations (see Section 5.4.2) kept in the public member `_ann`.

The methods `get_annotation` and `get_annotations` give access to one or all annotations stored in the full node.

There are two different accessors for the `model` contained in the full node: `get_model` returns a const pointer to the model, whereas `get_model_ptr` provides a pointer through which the model's contents can be changed.

In addition, there are accessors `get_database` and `get_database_ptr`, which give access to the inherited protected member `__dbase` of the base class.

5.4.2 Annotations

All models get additional annotations containing important information like local optima, exclusion boxes, and Lagrange multipliers. Those items are stored in the search database. The entries are determined by a database table and a row in that table. For both the ids are stored in a pair. The `tableid` of an annotation can be retrieved by the `get_table` method, and the `rowid` is returned by `get_entry`.

The vector of annotations, which are valid for a work node (see Section 5.4.3), determines its database view. Only the valid annotations are visible, all others hidden.

5.4.3 Work Nodes

Whenever the search focus is set on a search node of the search graph, the node is `extract`ed into an object of the class `work_node`. This class is a subclass of `full_node`, further enhanced with information and caches. The work node structure is highly important, because all inference engines analyze work nodes.

The public members of the `work_node` class are:

`deltas`: This array contains all ids of those deltas which have been applied in the sequence of their application.

`undeltas`: The undo information for all applied deltas is kept in this container.

`dtable`: This member points to the table of deltas in the search database.

`dtable_id`: This is the `tableid` of the delta table in the search database.

`node_ranges`: While one work node is analyzed this vector contains the currently best known range on each node in the expression DAG.

`infeasible`: If this entry is set to `true`, it has been proved that the solution set of this work node is empty.

`log_vol`: This is the logarithm of the volume of the search box in this node.

`gain_factor`: Whenever the bounds of a node are changed, this member is updated, roughly specifying the reduction in volume since the last reset.

`proposed_splits`: Every entry in this list represents one possibility of splitting the work node into smaller areas for the branching step of the branch-and-bound scheme. Every proposed split gets an associated `transaction_number` for referencing and undoing split deltas (see Section 5.5.6).

The public methods for the work node class are described in the following list:

`init_cnumbers`: This method initializes the internal caches of the work node.

`reset_node_ranges`: The ranges cache is a vector for fast access but the node numbers might not be continuous. Therefore, there might be some ranges entries, which have no meaning. To reset them to the default $[-\infty, \infty]$ range, this node must be used.

`make_node_ranges`: This method initializes the node ranges cache.

`compute_log_volume`: An approximation of the logarithm of the box volume is computed with this method. It makes sure that this number is finite, even for boxes with thin or infinite components.

`get_model`: This method returns a pointer to the `model` of the problem represented in this work node.

`get_delta`: These methods return a `delta` by retrieving it from the search database. When an unsigned integer i is passed the ith delta is returned, and if a `delta_id` is provided, the delta corresponding to that id is returned. The method raises an exception if the delta does not exist in the database.

`log_volume`: This is the accessor for the `log_vol` member.

`gain`: The member `gain_factor` can be accessed through this method.

`reset_gain`: A call to `reset_gain` resets the `gain_factor` to 1.

`n`: This method is used for counting constraints of specific types. The same type flags can be passed as for the `is` method of the `expression_node` class (see Section 5.3.2). The return value is the number of constraints, which are of the corresponding type. The most important values (number of linear, number of quadratic,…) are internally cached, so access is fast for those. For unusual combinations the time needed for this method is linear in the number of constraints.

The `work_node` class provides special iterators, `constraint_const_iterator` and `constraint_iterator`, for iterating over the constraints or a subset of them. The argument to the methods `get_begin` and `get_end` is a type specifier like for `n` and the `expression_node` method `is` (see Section 5.3.2). The method `get_begin` then returns a constraint iterator pointing to the expression node of the first constraint with the required property. Incrementing the iterator jumps to the next constraint with the property, and so forth. The `get_end` method returns an iterator pointing past the last constraint with the required property.

Finally, deltas can be applied to work nodes by "adding" their id with the operator `+`. They can be unapplied by "subtracting" their `delta_id` with the operator `-`. These operators work on single deltas as well as on sequences of deltas, as long as they are stored in any STL sequence container.

5.4.4 Search Graph

The class `search_graph` manages the DAG structure of the graph, the search foci, and the search inspectors. The publicly accessible data members are

`root`: This member is a pointer to the search node of the root of the search graph, the original problem.

`inspector_for_root`: The search inspector pointing to the root of the search graph is stored here.

`__dbase`: This is a global pointer reference to the search database.

`__dbuser`: This member stores the database `userid` used for all accesses to the database.

The handling of the search foci is performed by the three methods `new_focus` which creates a new focus pointing to the position of the provided search inspector, `destroy_focus` which removes the specified focus, and `set_focus` which changes the given focus to a new position.

Similarly, there are three methods for managing search inspectors (read only pointers to the search graph), namely `new_inspector` which creates an empty one or duplicates an existing inspector, and `destroy_inspector` which removes the inspector given from the graph.

For handling search nodes the following methods are provided:

`insert`: This method adds a new method to the search graph, as child of the node to which the search focus points. It returns a `search_inspector` to the newly created node.

`extract`: The search node pointed at by the search focus is copied into a `work_node` structure.

`replace`: This method replaces the search node in the focus by the new search node provided. The return value is a `search_inspector` to the new node.

`promote`: If a node is the only child of its parent, it can be promoted in the graph to replace its parent. This method removes the parent, replaces it with the child and resets the search focus to the former child. This new search focus is returned.

`remove`: This method removes the node in the search focus. The focus is invalidated by this operation.

`child`, `parent`: These two methods return a search inspector to the ith child (parent) of the provided node.

The types `search_focus` and `search_inspector` are defined as walkers and const walkers, respectively, of the search graph.

5.5 Deltas

As explained in Section 5.4 most of the nodes in the search graph only store differences to the parent node. These differences are stored in `delta`s. Every delta has a unique id once it is committed to the search database. These ids are kept in the delta nodes and in the work nodes. On demand the deltas are retrieved from the database to be analyzed or applied.

When a delta is applied to a work node, an `undelta` object is created for that delta. This object is used for the unapply operation. Applying and unapplying can be used to move from one position in the graph to another without the need to reconstruct the node by walking the full search graph.

In the next sections we will take a look at the base classes for the delta system and at those deltas which are currently implemented in the COCONUT environment.

5.5.1 Base Classes

The `delta` class is the main class of the ("positive") delta system. It is a wrapper class specifically designed for emulating a virtual copy constructor, which is not allowed directly in `C++`. Internally, the class holds a pointer to a `delta_base` object. It provides two accessors, `get_action` for the `action` specifier, a member of the `delta_base` class which describes what the delta actually does, and `get_base` which directly returns the `delta_base` pointer.

The methods `apply` and `apply3` apply the delta to the work node, and `apply` directly changes the node, while `apply3` constructs a new work node in the first parameter, which is a copy of the second parameter, on which the delta is applied. These methods directly use the `apply` and `apply3` methods of the `delta_base` class. Both methods return `true`, if applying the delta was successful.

The `convert` method is just a wrapper of the `convert` method for the `delta_base` class, which is called during the `store` operation. This converts a delta to the form in which it is saved in the database (see, e.g., Section 5.5.8), stores it in the database and generates a new `delta_id` for it. This `delta_id` is returned.

The `delta_base` class is the base class for all the different delta types, which are described in Sections 5.5.2–5.5.10. The constructor of the class takes a string argument (either `C++` or `C` format), which is used as the action specifier describing, what the delta does.

The `apply`, `apply3` take as second argument a pointer to an object of class `undelta_base`. This object must contain the unapply information for the delta just applied. This `undelta_base` object is wrapped into a `undelta` object and stored in the work node.

The `convert` and `get_action` methods have the same semantics as for the `delta` class. The `make_delta` method wraps the `delta_base` object into a `delta` object, and the `new_copy` method is the overloadable replacement of the copy constructor. It should create a new `delta_base` object (by using `new`), which is a copy of the object for which the method is called.

The `undelta` class is the main class of the ("negative") delta system. It is like `delta` a wrapper class specifically designed for emulating a virtual copy constructor. Internally, the class holds a pointer to an `undelta_base` object. It provides one accessor, `get_base` which directly returns the `undelta_base` pointer.

The methods `unapply` and `unapply3` unapply the delta from the work node, and, again, `unapply` directly changes the node, while `unapply3` constructs a new work node in the first parameter, which is a copy of the second parameter, from which the delta is unapplied. These methods also use the `unapply` and `unapply3` methods of the `undelta_base` class directly. Both methods return `true`, if unapplying the delta was successful.

The `undelta_base` class is the base class for all the different undelta types, which belong to the deltas described in Sections 5.5.2–5.5.10.

The `apply`, `apply3` take as second argument a pointer to an object of class `undelta_base`. This object must contain the unapply information for the delta just applied. This `undelta_base` object is wrapped into a `undelta` object and stored in the work node.

The `unapply` and `unapply3` methods have the same semantics as for the `undelta` class. The `make_undelta` method wraps the `undelta_base` object into an `undelta` object, and the `new_copy` method is again the overloadable replacement of the copy constructor. It should create a new `undelta_base` object (by using `new`), which is a copy of the object for which the method is called.

5.5.2 Infeasible Delta

The infeasible delta is used, if by some method the infeasibility of the model in the current worknode is proved. It sets the `infeasible` member in the `work_node` class to `true`.

5.5.3 DAG Delta

The `dag_delta` is the most complicated delta contained in the COCONUT API. It is responsible for changing the `model` inside the work node, especially the expression DAG, adding and removing constraints, changing the objective function, and the like.

A DAG delta contains three public members: `new_constraints` is a counted pointer, an intelligent pointer which deletes its contents after all references to them are removed. This relieves the system of copying large modules over and over again. The variable `rm_nodes` lists all nodes of the DAG which are to be removed. Actually this will remove the minimal subgraph defined by these nodes (see Section 5.1.3). The boolean variable `is_full_delta` determines whether all the model should be replaced by this delta, or whether the delta is an addition containing *ghost nodes*.

There are two `add_new` methods. The first takes a model reference, copies it into a new object and stores that in the counted pointer. The second one takes a pointer. This pointer's contents **must** be allocated with `new` and **must not** be deleted afterwards, since the *pointer* is copied into the counted pointer.

There is one constructor which takes as second argument a `model` pointer. This pointer's contents have to be allocated with `new`, as well.

The `remove` methods add new nodes to the `rm_nodes` array.

5.5.4 Bound Delta

A bound delta changes one or more bounds on expression nodes in the DAG. There are two public members. The `indices` vector contains the node numbers of the nodes whose bounds are changed. If the vector is empty, the bounds on *all* nodes are changed. The vector `new_f_bounds` contains the bounds. The bound `new_f_bounds[i]` applies to the node `indices[i]`.

There are two constructors. Either both vectors are set explicitly or a bound delta for a single node is constructed.

5.5.5 Semantics Delta

This delta changes the semantics information on nodes in the DAG. It has no public variable members. There are just public methods. All of these methods take a vector of indices (node numbers of the nodes to be changed) — if this vector is empty, all nodes are changed — and a vector of specifiers. E.g. `set_convex` takes a list of indices and a list of `convex_e` convexity specifiers. If the method is called, the semantics information of all nodes, whose number is contained in the vector of indices, will be changed to the new convexity status provided in the vector of convexity specifiers. The convexity information of all other nodes remains unchanged. The other methods `set_activity`, `set_separable`, `set_is_at_either_bound`, `set_integer`, `set_hard`, and `set_type` work the same way. The method `set` changes the whole semantics information of the listed nodes in one step.

5.5.6 Split Delta

The split delta is used to add new *proposed splits* to a work node. Note, that this delta does not actually split the work node. Splitting of work nodes is done by management modules.

There is one public member, `splits`, which is a list of a vector of deltas. Every list member defines one subproblem, and the vector describes the deltas which must be applied to generate the subproblem. In many cases the inner vector will be just one element; however the BCS inference engine creates bound and semantics updates at the same time.

The method `add_delta` inserts a new single delta (as a vector of one) or a vector of deltas into the `splits` list.

The `add_split` method adds a simple split, or a vector of those to the `splits` list by automatically creating the relevant `bound_delta`s (see Section 5.5.4).

5.5.7 Annotation Delta

The `annotation_delta` is used for adding annotations (see Section 5.4.2) to work nodes or for removing them. Annotations are strongly related to rows of database tables, so annotation deltas usually are not created by inference engines, unless they explicitly insert rows into certain database tables. Usually, an annotation table is automatically generated by `convert`ing (see Section 5.5.1) `table_delta`s (see Section 5.5.8).

The table deltas create the database entries and afterwards convert themselves to the annotation delta which contains the `tableid`–`rowid` pairs needed for accessing the newly created entries.

5.5.8 Table Delta

The `table_delta` class is the base class for a number of deltas, which deal with tables. When applied to a work node the table delta performs all required database updates and `convert`s itself to an `annotation_delta` which inserts the newly created table rows into the `_ann` member of the `work_node` (inherited from the `full_node` class).

There is one virtual function which needs to be written for all subclasses; `create_table` is called whenever an entry for a not yet existing table has to be made. Inside that function it is required from the subclass that it sets up the database table properly for all tables it uses. The table name is passed as third parameter, and through the second parameter a pointer to the newly created is returned. The function returns false, if creating the new table has failed.

The methods `add` can be used to add new table–row pairs, and the `rm` methods add new annotation entries to the work node, which should be prepared for removal.

5.5.9 Box Delta

The `boxes_delta` is a special class of table delta, which adds a new entry into the `box` table. The possible row names and their types are

Name	Type	Description
`x`	`vector<interval>`	the box
`exclusion box`	`bool`	whether the box is an exclusion box
`contains optimum`	`bool`	whether the box surely contains the solution
`eval region`	`bool`	whether all points in the box can be evaluated

5.5.10 Point Delta

The `point_delta` is a special class of table delta, which adds a new entry into the `point` table. The possible row names and their types are

Name	Type	Description
`x`	`vector<double>`	the point
`f`	`double`	the point's objective function value
`L_mult`	`vector<double>`	the constraint multipliers
`kappa`	`double`	the objective multiplier
`class`	`unsigned int`	the point class of the point
`best`	`bool`	this is the best known point
`feasible`	`bool`	this point is feasible
`optimal`	`bool`	this point is a local optimum
`global`	`bool`	this point is valid for the original problem
`relaxation`	`bool`	the entries are valid for a relaxation only

5.6 Evaluators

For expression graphs (DAG or tree), special forward and backward evaluators are provided. Currently implemented are *real function values*, *function ranges*, *gradients* (real, interval), and *slopes*.

In the near future evaluators for *Hessians* (real, interval) and *second order slopes* (see, e.g., [200]) will be provided, as well.

Evaluators are small modules in the API which can be used by inference engines, and which provide information about certain aspects of the model DAG or parts of the model DAG.

The evaluators were designed as intermediate layer between the inference engines and the internal representation of optimization problems. Their outside structure is more or less independent of DAGs, so if the internal representation would change in the future, inference engines using evaluators should not need lots of updates.

All evaluators are constructed from a very small number of base classes. They can make use of the `variable_indicator`, a bitmap recording which variables every node depends on. This variable indicator has methods `reserve` for preparing the indicator for the specified number of variables, `set` and `unset` for setting and unsetting single entries or ranges of entries, and `clear` for resetting the whole variable indicator. The method `test` can be used to determine, whether a single variable is set, and `match` returns `true`, if two variable indicators have an empty intersection of variables which are set.

For a full class description of all described classes see the reference manual of the COCONUT API Version 2 [197].

5.6.1 Base class

There are four different base classes for evaluator programming, depending on the type of evaluation:

- forward evaluation,
- backward evaluation,
- forward evaluation with caching,
- backward evaluation with caching.

For an evaluator, a number of methods have to be provided:

`initialize`: This method is called for every node before children are visited. For forward evaluation the method should return an `int`. If the return value is negative, the `short_cut_to` method will be called for performing a short-cut. If the return value is $i > 0$, then skip $i - 1$ children in the evaluation. If the return value is zero, then do not walk to the children.

`calculate`: For forward evaluation this method is called after all children have been visited, for backward evaluation it is called right before the first child is visited. For backward evaluation a return value of type `int` is expected, which follows the same rule as the return value of `initialize` for forward evaluators.

`calculate value`: This method is called before the evaluator is destroyed. It should return the evaluator's return value.

`cleanup`: This method is called just before `calculate_value`, and it should be used to get rid of allocated data,...

`update`: The `update` method is called every time after a child has finished. The second argument is the return value of the child. The method should return `int`. This number is the number of children which should be skipped during evaluation. If the return value is negative, no more children are visited.

`short_cut_to`: This method is only called, if initialize (for forward) or calculate (for backward) return a negative number. It should return a walker pointing to the point the short-cut leads to. The graph walk is continued there.

`is_cached`: This method should return a `bool` specifying whether the result of this node is already in the cache.

retrieve_from_cache: If `is_cached` returns `true`, this method is called. It should retrieve from the cache the result and store it somewhere such that `calculate_value` can produce a proper return value.

All evaluators are used in the same way. First they are constructed using all the relevant data. Then the `evaluate` function is called. This function takes two arguments. The first is the evaluator, the second argument is a `walker` of the model DAG pointing to the node (e.g., objective, constraint) which should be evaluated. The evaluate function returns the return value the evaluator provides for the start node.

Examples for the proper use of all evaluators defined here can be found in the subsections with corresponding titles and numbers in Section A.5.

5.6.2 Function Evaluation

This is the simplest evaluator. It provides a real function evaluation. The constructor has the form

```
  func_eval(const std::vector<double>& x, const variable_indicator& v,
            const model& m, std::vector<double>* c);
```

Here `x` defines the point on which the function is evaluated. `v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `m` contains a reference to the actual model, and `c` is the cache. If `NULL` is passed as the cache's pointer, the evaluator works without caching.

5.6.3 Gradient Evaluation

Gradient evaluation consists of **three** evaluators: `prep_d_eval`, `func_d_eval`, and `der_eval`.

The evaluator utilizes the backward mode of automatic differentiation. For this the partial derivatives of every node with respect to its children have to be computed. This is done in forward mode during a function evaluation.

The `prep_d_eval` evaluator is used to set up the data structure for automatic differentiation. It has to be called before `func_d_eval` is called. Its constructor is

```
prep_d_eval(std::vector<std::vector<double> >& d,
            unsigned int num_of_nodes);
```

where `d` is a reference to the automatic differentiation data structure.

After `prep_d_eval` has set up the data structure, `func_d_eval` is used to calculate a function value, like with `func_eval` with the difference that all partial derivatives in the tree are kept within the already set up data structure. The constructor of `func_d_eval` is:

```
func_d_eval(const std::vector<double>& x, const variable_indicator& v,
            const model& m, std::vector<std::vector<double> >& d,
            std::vector<double>* c);
```

where `x`, `v`, `m`, and `c` have the same semantics as in `func_eval` (see Section 5.6.2). The parameter `d` is a reference to the automatic differentiation data structure.

After the function evaluation the third evaluator `der_eval` is called, which computes the gradient. The constructor is

```
der_eval(std::vector<std::vector<double> >& der_data,
         variable_indicator& v, const model& m,
         std::vector<std::vector<double > >* d,
         std::vector<double>& grad);
```

Here `der_data` is a reference to the derivative structure filled by `func_d_eval` before. There is no internal status or storage in `der_eval`, and all information is passed via `der_data`. The parameters `v` and `m` are as before, and `d` is the cache for derivative evaluation. Finally, the parameter `grad` is a reference to a vector. The gradient evaluated is added to the vector stored in `grad`. So, it has to be initialized to 0, if just the gradient shall be computed. For, e.g., computing the gradient of some Lagrange function, a multiple α of the gradient of this function can be added to `grad`, if before `evaluate` is called, the `set_mult` method is used to preset α in the `der_eval` object.

5.6.4 Range Evaluation (Interval Evaluation)

This is the simplest interval evaluator. It provides a function range enclosure using interval arithmetic. The constructor has the form

```
interval_eval(const std::vector<interval>& x, const variable_indicator& v,
              const model& m, std::vector<interval>* c, bool do_i = true);
```

Here `x` defines the box over which the function range is enclosed. `v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `m` contains a reference to the actual model, and `c` is the cache.

If `do_i` is `true` (the default), then forward propagated values are intersected with original node bounds. If it is set to `false`, simple forward interval propagation is used.

5.6.5 Interval Derivative Evaluation

This evaluator set provides range enclosures for gradients. It consists of **three** evaluators, like gradient evaluation (see Section 5.6.3): `prep_id_eval`, `func_id_eval`, and `ider_eval`.

The evaluator utilizes the backward mode of automatic differentiation. For this the enclosures for all partial derivatives of every node with respect to its children have to be computed. This is done in forward mode while a function range enclosure is computed. The special results on improved ranges due to exploiting the results of constraint propagation from Section 4.1.3 based on the results in [201] have been implemented.

The `prep_id_eval` evaluator is used to set up the data structure for automatic differentiation. It has to be called before `func_id_eval` is called. Its constructor is

```
prep_id_eval(std::vector<std::vector<interval> >& d,
             unsigned int num_of_nodes);
```

where `d` is a reference to the automatic differentiation data structure.

After `prep_id_eval` has set up the data structure, `func_id_eval` is used to calculate a function range enclosure, like with `interval_eval` with the difference that ranges of partial derivatives in the tree are kept within the already set up data structure. The constructor of `func_id_eval` is:

```
func_id_eval(const std::vector<interval>& x,
             const std::vector<interval> rg,
             const variable_indicator& v, const model& m,
             std::vector<std::vector<interval> >& d,
             std::vector<interval>* c)
```

where x, v, m, and c have the same semantics as in interval_eval (see Section 5.6.4). The parameter d is a reference to the automatic differentiation data structure. The rg vector contains the best known range of all nodes in the DAG (even of the intermediate ones).

After the range evaluation the third evaluator ider_eval is called, which computes the gradient. The constructor is

```
ider_eval(std::vector<interval> x,
          std::vector<std::vector<interval> >& ider_data,
          variable_indicator& v, const model& m,
          std::vector<std::vector<interval> >* d,
          std::vector<interval>& grad)
```

Here ider_data is a reference to the derivative structure filled by func_id_eval before. There is no internal status or storage in ider_eval, and all information is passed via ider_data. The parameters v and m are as before, and d is the cache for derivative evaluation. Finally, the parameter grad is a reference to a vector, to which the interval gradient is added. The vector has to be initialized to 0, if the interval gradient of this function is to be computed. The first parameter x is a reference to the box over which the gradient should be enclosed. The set_mult method can be used to preset a scalar (an interval in this case) as for the der_eval class (see Section 5.6.3).

5.6.6 First order Slope Evaluation

This evaluator set provides first order slopes for functions. It consists of **three** evaluators, like gradient evaluation (see Section 5.6.3) does: prep_islp_eval, func_islp_eval, and islp_eval.

The evaluator utilizes the backward mode of automatic differentiation. For this the partial slopes of every node with respect to its children have to be computed. This is done in forward mode during a function evaluation on the center of the slope. The special results on improved ranges due to exploiting the results of constraint propagation from Section 4.1.3 based on the results in [201] have been implemented.

The prep_islp_eval evaluator is used to set up the data structure for automatic differentiation. It has to be called before func_islp_eval is called. Its constructor is

```
prep_id_eval(std::vector<std::vector<interval> >& d,
             unsigned int _num_of_nodes);
```

where d is a reference to the automatic differentiation data structure.

After prep_islp_eval has set up the data structure, func_islp_eval is used to calculate a function range enclosure, like with interval_eval with the difference that partial slopes in the tree are kept within the already set up data structure. The constructor of func_islp_eval is:

```
func_islp_eval(const std::vector<double>& z,
               const std::vector<interval> rg,
               const variable_indicator& v, const model& m,
               std::vector<std::vector<interval> >& d,
               std::vector<double>& f);
```

where `z` is the center of the slope, and `v`, and `m` have the same semantics as in `interval_eval` (see Section 5.6.4). The parameter `d` is a reference to the automatic differentiation data structure. The `rg` vector contains the best known range of all nodes in the DAG (even the intermediate ones). The parameter `f` is used to store center information. The evaluator returns a value of type `func_islp_return_type` consisting of an approximate function evaluation at the center z, a function range enclosure over the point interval $\mathbf{z} = [z, z]$, and a function range enclosure over the box, over which the slopes are to be calculated.

After the function evaluation at the center the third evaluator `islp_eval` is called, which computes the slope. The constructor is

```
islp_eval(std::vector<std::vector<interval> >& islp_data,
          variable_indicator& v, const model& m,
          std::vector<std::vector<interval> >* d,
          std::vector<interval>& islp);
```

Here `islp_data` is a reference to the derivative structure filled by `func_id_eval` before. There is no internal status or storage in `islp_eval`, and all information is passed via `__islp_data`. The parameters `v` and `m` are as before, and `d` is the cache for slope evaluation. Finally, the parameter `islp` is a reference to a vector, to which the computed slope is added, again a multiple $\alpha \in \mathbb{IR}$ can be added, if the `set_mult` method is used on the evaluator prior to the call to `evaluate`. For computing the raw slope, the vector `islp` has to be initialized to 0.

A number of evaluators for one-dimensional functions are defined, as well. They have been written by Knut Petras from the Technische Universität Braunschweig.

5.6.7 Analytic-Differentiable Evaluation

This evaluator produces in one sweep

- an interval for range evaluation
- an interval for derivative evaluation
- a complex (rectangular) interval symmetric with respect to the real axis (consisting of an interval representing the real part and a double representing the maximal distance from the real axis).

The constructor has the form

```
 analyticd_eval(const std::vector<analyticd>& __x, const variable_indicator& __v,
                const model& __m, std::vector<analyticd>* __c) : _Base()
```

Here __x defines the box over which the evaluation is done, and __v is a variable indicator, where all variables are set, if they have been changed since the last evaluation. The parameter __m contains a reference to the actual model, and __c is the cache.

This evaluator is not yet implemented for many operators, but the most important ones are covered.

5.6.8 Bounded Interval Evaluation

This is an interval evaluator for bounded evaluation. It provides a function range enclosure using interval arithmetic avoiding infinities. The constructor has the form

```
 b_interval_eval(const std::vector<b_interval>& x,
                 const variable_indicator& v,
                 const model& m, std::vector<b_interval>* c);
```

Here x defines the box over which the function range is enclosed. v is a variable indicator, where all variables are set, if they have been changed since the last evaluation. The parameter m contains a reference to the actual model, and c is the cache.

5.6.9 Complex Interval Evaluation

This evaluator provides enclosures of the function in complex (rectangular) intervals symmetric with respect to the real axis. The constructor has the form

```
 c_interval_eval(const std::vector<c_interval>& x,
                 const variable_indicator& v,
                 const model& m, std::vector<c_interval>* c);
```

Here x defines the box over which the function range is enclosed. v is a variable indicator, where all variables are set, if they have been changed since the last evaluation. The parameter m contains a reference to the actual model, and c is the cache.

5.6.10 Infinity-Bound Evaluation

This is an evaluator doing asymptotic arithmetic. It provides a function enclosure at $\pm\infty$. The constructor has the form

```
infbound_eval(const std::vector<infbound>& x, const variable_indicator& v,
              const model& m, std::vector<infbound>* c);
```

Here `x` defines the asymptotic region over which the function is enclosed. `v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `m` contains a reference to the actual model, and `c` is the cache.

5.7 Inference Modules

For the solution strategy, the most important class of modules are the **inference modules**. They provide the computational base for the algorithm, namely methods for problem structure analysis, local optimization, constraint propagation, interval analysis, linear relaxation, convex optimization, bisection, analysis of the search graph,. . . .

The inference modules are grouped in two classes, the **inference engines**, which act on problems stored in a work node, and the **graph analyzers**, that are responsible for analyzing the search graph.

Corresponding to every type of problem change, a class of inference engines is designed: **model analysis** (e.g. find convex part), **model reduction** (e.g. pruning, fathoming), **model relaxation** (e.g. linear relaxation), **model splitting** (e.g. bisection), **model gluing** (e.g. undo excessive splitting), **computing of local information** (e.g. probing, local optimization).

Inference engines calculate changes to a model that do not change the solution set. But they **never** change the model; the decision to apply the changes if they are considered useful is left to the strategy engine. Therefore, the result of an inference engine is a list of changes to the model together with a weight (the higher the weight the more important the change). Whether an advertised change is actually performed is decided by the strategy engine, and the actual change is executed by an appropriate management module. The inference engines are implemented as subclass of a single `C++` base class. In addition, there is a fixed documentation structure defined.

Several state of the art techniques are already provided:

- DONLP2-INTV, a general purpose nonlinear local optimizer for continuous variables [216],
- STOP, a heuristic starting point generator,
- Karush-John-Condition generator using symbolic differentiation,
- Point Verifier for verifying solution points,
- Exclusion Box generator, calculating an exclusion region around local optima [202],
- Interval constraint propagation [17, 13] in two variants,
- Linear Relaxation,

- Linear Programming: a CPLEX wrapper for the state of the art commercial linear programming solver by ILOG, an XPRESS-MP wrapper for the state of the art commercial LP solver by Dash Optimization, and an LPsolve wrapper for the public domain linear solver.
- Basic Splitter,
- BCS, a box covering solver [196, 209],
- Convexity detection, for simple convexity analysis,
- Valid lower bounds for convex problems.

All graph analyzers are subclasses of another `C++` base class, and the documentation structure is the same as for inference engines. The only graph analyzer implemented so far is the box chooser.

5.7.1 Base Classes

The base classes for the inference modules mainly are concerned with data transfer from the module. The data transfer to the module is realized by means of the `control_data` class.

Termination Reason

The first basic class is used to encapsulate the reason of termination of an inference engine. The class holds a termination code, an `int`, and a termination message (a `C++` string).

By definition, a termination code ≥ 0 states success (errorfree behaviour) of the engine. If the code is negative, a severe error has occurred. The message should always be useful output understandable by humans.

There are two important methods: `get_code` returns the termination code, and `get_message` the termination message.

Information Contents

The `info_contents` class is used for returning complex data from inference modules to other modules and the strategy engine. The `info_contents` class is a subclass of the `datamap` class (see Section 5.3.1) and uses the same methods.

Inference Engine Return Type

The return type of all inference modules, and of some management modules is the `ie_return_type` class. Objects of this class contain the deltas and corresponding weights, which are created by the inference modules, the `info_contents` part, and the termination reason inside a `termreason` object.

There is a large number of methods defined for the `ie_return_type` class.

`set_termination_reason`: This method sets the termination reason structure in the return type according to the parameter passed.

`term_reason`: This accessor returns the termination reason of the alorithm.

`set_information`, `set_information_i`: These two methods set parameters in the `info_contents` part of the `ie_return_type` object.

`information`: Retrieving one item from the `info_contents` structure is done by this method. The name of the entry is provided.

`has_information`: This method checks, whether a certain information piece is set in the information structure.

`unset_information`: A call to this method removes the specified entry from the `info_contents`.

`information_indices_set`: This method is the same as `which` for the `info_contents` part of the return type.

`retrieve_from_info`, `retrieve_from_info_i`: In analogy to the `retrieve` and `retrieve_i` operations in the `info_contents` class, these methods retrieve specific entries from the information part, and assign that information to variables.

`n_deltas`: This method returns the number of delta–weight pairs stored in the return type.

`get`: The `get` method takes one work node and a *double* threshold value. It takes all deltas from the return type, whose weight is bigger than the threshold, those deltas are `convert`ed and stored in the database. The list of delta ids is returned. This list can then be "added" to the work node.

The operator `+` adds a new weight–delta pair to the return type object.

Inference Engine Statistics

The `statistic_info` class is the base class of the statistics classes in the various inference modules. There is one required public members which has to be set by methods of the subclasses: `effectiveness` which should give a hint on *how* successful the inference engine was. The other member, `number_of_infers`, is updated automatically.

5.7.2 Inference Engines

The first base class is `inference_engine`, whose subclasses are those inference modules, which analyze work nodes. All inference engines have the same constructor, taking only a constant global pointer reference to a work node. The base class has a second parameter in the constructor, a *name* in `C++` string form.

The standard methods are

`update_engine`: This method is called with a new work node reference, whenever the inference engine has to be prepared for analyzing a new work node. It returns `true` if updating was successful and `false`, otherwise.

`last_call_stat`: The statistics of the last `infer` call of this engine is returned by this method.

`cumulative_stat`: This method returns the cumulative statistics for all `infer` calls.

`infer`: The most important method performs the actual computations. Its behaviour is influenced by its only parameter, the `control_data` structure. The return value is an `ie_return_type` containing `deltas`, `info_contents`, and the `termreason`.

Some protected members can be used in subclasses:

`__name`: the name of the engine,

`__wnode`: the global pointer reference to the work node,

`__wnc`: the work node context for evaluating database queries,

`__vdb`: a view onto the search database in the evaluation context,

`_old_deltas`, `_new_deltas`: For updating the engine, these deltas contain the difference between the previous and the new work nodes. Here, `_old_deltas` have to be unapplied and `_new_deltas` have to be applied for changing the previous work node into the new one.

5.7.3 Graph Analyzers

A graph analyzer performs computations on the search graph. Their base class is `graph_analyzer`. It is very similar to the `inference_engine` class. The main difference lies in the fact that two parameters are passed: the constant global pointer reference to the search graph, and optionally a constant pointer to the search focus. For the base class the module name is the final parameter.

The `update_engine` methods work like for the `inference_engine` class, and the most important method, `analyze` performs the computation. It is driven by a `control_data` parameter, and it returns an `ie_return_type` object.

The following protected members can be used in subclasses:

`__name`: the name of the engine,

`__sgraph`: the global pointer reference to the search graph,

`__sgc`: the search graph context for evaluating database queries,

`__vdb`: a view onto the search database in the evaluation context,

`__sfoc`: the current search focus. This can be `NULL` if no search focus exists.

5.8 Management Modules

Management modules are the interface between the strategy engine and the internal representation of data and modules, taking care of the management of models, resources, initialization, the search graph, the search database, ...

They are provided to make it possible to change the implementation of the search graph and the internal representation of problems without having to change all of the modules. Management modules just perform some of the changes which have been advertised by inference engines; they **never** calculate anything.

There are two different classes of managment modules, the "standard" *management modules*, and the *initializers*, which are used for initializing and destroying models, search databases, and search graphs.

5.8.1 Management Modules

The `management_module` class is the base class of all modules which perform management tasks other than creating or destroying the search graph, search database, or the global model.

There is a number of protected members, which can be used in the subclasses:

`__name`: the name of the module,

`__wnode`: the current work node,

`__sfocus`: the current search focus,

`__sinsp`: a search inspector,

`__dbase`: the search database,

`__sgraph`: the search graph.

The constructors of the management modules initialize all of them, and there is a requirement that every subclass provides at least those constructor which contains parameters for *every* one of these members. The base class furthermore contains a lot of constructors whose parameter list is restricted. The writers of subclasses are encouraged to define constructors with a minimal number of parameters for the usage within hard coded strategies.

The most important member function of a management module is `manage`, which has a `control_data` structure as its only parameter and returns an `int`. A return value of zero means success, a positive value is still OK but identifies some minor issue. A negative return value is only provided if a fatal error occurs.

5.8.2 Initializers

For very special management tasks which construct or destroy the main components *search graph*, *global model*, and *search database* a special base class is defined. The `initializer` class has similar protected members

`__name`: the name of the module,

`__wnode`: the current work node,

`__sfocus`: the current search focus,

`__dbase`: the search database,

`__sgraph`: the search graph.

with the distinction that all these parameters (except name) are double pointers to global references, and so they can be allocated or destroyed by the module.

The standard method is `initialize`, which takes a `control_data` and returns an `ie_return_type`. This is mainly important for verbose error reporting.

5.9 Report Modules

The final class of modules, called **report modules**, produce output. Human or machine readable progress indicators, solution reports, the interface to modeling languages [109] (currently only `AMPL` [58] is supported), and the biggest part of the checkpointing is realized via report modules. All parameter passed to report modules are const references, so report modules can never change anything.

5.9.1 Base Class

The base class of all report modules is `report_module`.

There is a number of protected class members which can be used in subclasses:

`__name`: the name of the engine,

`__wnode`: the global pointer reference to the work node

`__sgroot`: the global pointer reference to the root of the search graph,

`__dbase`: the search database

`__wnc`: the work node context for evaluating database queries,

`__vdb`: a view onto the search database in the evaluation context,

`__ier`: an `ie_return_type`.

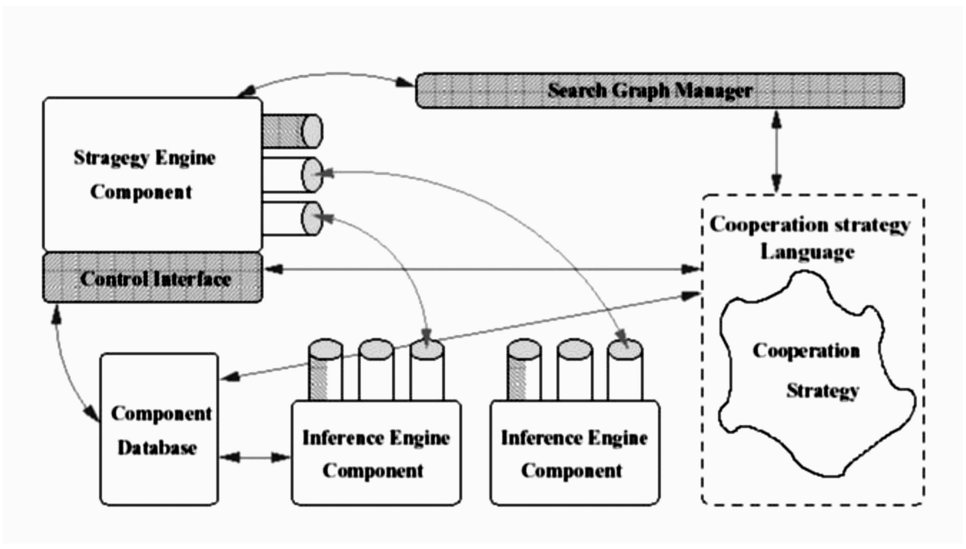

Search Graph Manager
Stragegy Engine Component
Control Interface
Cooperation strategy Language
Cooperation Strategy
Component Database
Inference Engine Component
Inference Engine Component

Figure 5.5. *The Strategy Engine Component Framework*

Some of these pointers may be `NULL` because they are irrelevant for the services provided by the report module. Like for management modules and initializers there are many constructors. There is a requirement that every report module at least admits the biggest constructor. However, writers of report modules are encouraged to define constructors with a minimal number of parameters.

5.10 The strategy engine

The **strategy engine** is the main part of the algorithm. It makes decisions, directs the search, and invokes the various modules. It was written by Brice Pajot and Eric Monfroy from IRIN, Univ. of Nantes.

The strategy engine consists of the logic core ("**search**") which is essentially the main solution loop, special **decision makers** (very specialized inference engines, see Section 5.7) for determining the next action at every point in the algorithm. It calls the management modules, the report modules, and the inference modules in a sequence defined by programmable search strategies.

The engine can be programmed using a simple **strategy language**, an interpreted language based on Python. Since it is interpreted, (semi-)interactive and automatic solution processes are possible, and even debugging and single-stepping of strategies is supported. The language is object oriented, garbage collecting, and provides dynamically typed objects. These features make the system easily extendable.

Furthermore, the strategy engine manages the search graph via the **search graph manager**, and the search database via the **database manager**.

The strategy engine uses a component framework (see Figure 5.5) to communicate with the inference engines. This makes it possible to launch inference engines dynamically (on need, also remote) to avoid memory overload. Since the strategy engine is itself a component, even multilevel strategies are possible.

Chapter 6
COCONUT modules

This chapter is devoted to the description of the modules, which are available for the environment. All of the modules follow the standard format and belong to one of the five module classes described in Chapter 5. In the environment all of the modules come together with a simple C++ wrapper function, which makes it easy to use the module in a hardcoded strategy.

6.1 Inference Engines

The most important modules are the inference engines. In this section, the inference engines which are implemented will be described in a very short form.

6.1.1 Basic splitter

`Basic Splitter` is a **splitter module** which computes useful splitting coordinates and splitting points for the model contained in the work node.

Service split

`Basic Splitter` computes a series of splits, each consisting of a split coordinate and one or more split points in this coordinate.

If there are Lagrange multipliers available, the slope s of the Lagrangian in the original variables x is computed using these multipliers. Then the variable with highest $\mathrm{wid}\,(s_i)\widehat{\mathrm{wid}}\,(x_i)$ is chosen as splitting coordinate. If there are no multipliers, the widest coordinate is chosen.

If there are exclusion boxes available, $\widehat{\mathrm{wid}}\,(x_i)$ is computed as $\mathrm{wid}\,(x_i) - \mathrm{wid}\,(b_i)$ where $\boldsymbol{b}$ is the exclusion box.

If the exclusion box is comparatively big (its volume is bigger than `excl_ratio` times the volume of the variable box), then the box is trisected along $\underline{\boldsymbol{b}_i}$ and $\overline{\boldsymbol{b}_i}$. Otherwise the box is bisected along one of these coordinates, depending on when the resulting boxes are more similar in size.

If there is no exclusion box, the box is bisected in the middle (geometric mean) or at 0, if $\boldsymbol{b}_i$ contains 0. If $\boldsymbol{b}_i$ is unbounded, then the splitting coordinate is either chosen using an evaluation region (trisect at the bound of this evaluation region) or at the signed square of the finite bound.

The code returns with a number `split_delta`s each representing a possible and useful split.

There is no useful statistical information collected, the efficiency is always 1.

Control parameters:

`excl_ratio`: This parameter is of type `double` and specifies the ratio of exclusion box size vs. box size at which the exclusion box coordinates are used for trisection instead of bisection. The default is `1`.

`number of splits`: With this parameter of type `int` the number of alternative splits is specified. The default is `1`.

`split multipliers`: This parameter of type `bool` determines whether variables corresponding to Lagrange multipliers of the original problem should be considered for splitting as well. The default is `false`.

`multiplier`: Using this `vector<double>` the user can specify an estimate for the Lagrange multipliers. These are then use as weights for the constraints. Default: no multiplier.

There is no termination reason except `'SUCCESS'` (0), `'no valid split coordinate'` (-2), and `'NO MODEL DEFINED'` (-1).

Service hard variables

`Basic Splitter` uses the split finding algorithms to determine the variables which are most likely to be split, and therefore can be considered hard.

The code returns a list of variables in the `information` entry with index `hard variables` as a `vector<unsigned int>`.

There is no useful statistical information collected, the efficiency is always 1.

Control parameters:

`excl_ratio`: This parameter is of type `double` and specifies the ratio of exclusion box size vs. box size at which the exclusion box coordinates are used for trisection instead of bisection. The default is `1`.

`number of splits`: With this parameter of type `int` the number of alternative splits is specified. The default is `1`.

`multiplier`: Using this `vector<double>` the user can specify an estimate for the Lagrange multipliers. These are then use as weights for the constraints. Default: no multiplier.

There is no termination reason except `'SUCCESS'` (0), `'no valid split coordinate'` (-2), and `'NO MODEL DEFINED'` (-1). No useful statistical information is collected.

6.1.2 BCS

The BCS engine was written by a group of people at EPFL Lausanne. This section is built from material from their documentation.

`BcsEngine` is a

Splitter for Optimization

Solver for Constraint Satisfaction

This is an inference engine which provides a `split_delta` for branching in searches. Each `split_delta` is a list of deltas. working as branches in searches. Each such a delta is currently a combination of changes in bounds of variables and changes in the activity status of constraints. BCS Inference Engine provides four classes of algorithms:

1. `DMBC` [210, 224]: dichotomous maintaining bound-consistency.
2. `DMBC+` [224, 225]: dichotomous maintaining bound-consistency enhanced by checking if a box is feasible (for all constraints).
3. `UCA6` [210, 224]: combination of dichotomous splitting and box splitting, immediate propagation.
4. `UCA6-Plus` [224, 225]: does the things similar to `UCA6`, but enhanced by a precision-controlled combination of boxes. It's more suitable than `UCA6` to work as a technique for solving subproblems of high-dimensional problems.

Dependencies: This module requires C++ ILOG Solver Libary for `gcc 3.0.x` on Linux or IRIN jail and the IRIN solver platform.

Service solve

Control Parameters:

1. *Name*: `Algorithm`

   Type: `int`

   Description: `0==UCA6-Plus, 1==UCA6, 2==DMBC+, 3==DMBC`

   Default value: 0

2. *Name*: `IsRelativePrecision`

   Type: `bool`

   Description: whether or not `BcsParameters::_Epsilon` is a relative precision?

   Default value: `true`

3. *Name*: `Epsilon`

   Type: `double`

   Description: a predefined precision

   dependency: `BcsParameters::_IsRelativePrecision`

   Default value: 0.01 (i.e., 1 percent)

4. *Name*: `DefaultPrecision`

   Type: `double`

   Description: a predefined (absolute) precision used for operations not associated with a specific precision

Default value: 0.1

5. *Name*: `FeasibilityPrecision`

   Type: `double`

   Description: a predefined (absolute) precision used for feasibility checker

   Default value: 0.1

6. *Name*: `FeasibilityChkLevel`

   Type: `int`

   Description: the level of feasibility checking for ε-bounded boxes:
   = 0: do nothing w/ them, just simply consider them as undiscernible boxes
   = 1: only check to see if ε-bounded boxes are infeasible?
   = 2: only check to see if ε-bounded box are feasible?
   = 3: do both checks: infeasible and feasible?

   Default value: 3 (recommended: 0 or 1)

7. *Name*: `FragmentationRatio`

   Type: `double`

   Description: The ratio of distance between the ends of two intervals and the longest interval can be considered as too near.

   Default value: 0.25

8. *Name*: `SearchMode`

   Type: `int`

   Description: 0==breadth search, 1==depth search

   Default value: 1

9. *Name*: `SplitMode`

   Type: `int`

   Description:
   = 0: combination of CBS and DS Slitting,
   = 1: DS Splitting only (can be called `DMBC++`)

   Default value: 0

10. *Name*: `NumOfCBC`

    Type: `int`

    Description:
    = −1: compute Complementary-Box for all active constraints
    >= 0: maximum number of Complementary-Box of active constraints to be computed

    Default value: −1

Termination Reason:

Integer: 0 (success)

String:

- "in `DAG_to_BCS_Expression::postorder(...)`: do not know how to handle constant vectors";
- "in `DAG_to_BCS_Expression::postorder(...)`: do not know how to handle `EXPRINFO_LIN`";
- "in `DAG_to_BCS_Expression::postorder(...)`: do not know how to handle `EXPRINFO_QUAD`";
- "in `DAG_to_BCS_Expression::postorder(...)`: DAG node of unexpected type";
- "in `DAG_to_BCS_Expression::postorder(...)`: do not know how to handle functions defined by the user";
- "in `DAG_to_BCS_Expression::postorder(...)`: do not know how to handle `EXPRINFO_GHOST`".

6.1.3 Linear Programming — CPLEX

The `cplex` module is an LP solver, a general purpose linear optimizer for continuous variables. It is intended for the minimization of a linear real function f subject to linear inequality and equality constraints . Such problems can be stated in the following form:

$$\begin{aligned} &\min c^T x \\ &\text{s.t. } Ax \in \boldsymbol{b} \\ &\qquad x \in \boldsymbol{x}. \end{aligned}$$

Here A is a matrix of dimension `m`×`n`.

This module requires ILOG CPLEX and a **license** for that program.

Service solve

The code calls CPLEX to solve the LP and returns a minimizer, the multipliers from the multiplier rule, and the corresponding function value, or it states that the problem is infeasible. It also collects some statistical information.

Given a work node the code evaluates the model DAG and extracts the required information from this. `cplex` assumes that the objective function is linear and extracts the linear constraints from the model DAG. This is individual information about the coefficients of the linear constraints and the bounds on the variables and the constraints.

The code returns a minimizer, the multipliers from the multiplier rule and the corresponding function value. It also collects some statistical information. This is specified by an object of the class `point_delta` or `infeasible_delta` from the COCONUT API

Control Parameters:

Name: `lp_algorithm`

Type: `string`

Description: the LP algorithm to use; the options are "`primal`", "`dual`", "`barrier`", "`automatic`", "`safe dual`"

Default value: "`safe dual`"

Reason for termination:

Integer: 0 (success) or the negative of the ILOG CPLEX status (see the ILOG CPLEX Reference Manual)

String:

- "SUCCESS" in the case of success

Description: For the explanation of the status values, see the ILOG CPLEX Reference Manual.

It collects the `_it_cnt`, the number of iterations, for statistical analysis.

6.1.4 Local optimization — donlp2_intv

Part of the information in this section is due to P. SPELLUCCI [216]. `donlp2_intv` is a general purpose nonlinear optimizer for continuous variables. It is intended for the minimization of an (in general nonlinear) differentiable real function f subject to (in general nonlinear) inequality and equality constraints. Specifically, the problem is regarded in the following form:

$$\begin{aligned} f(x^*) &= \min\{f(x) : x \in \mathcal{S}\} \\ \mathcal{S} &= \{x \in \mathbb{R}^n : x_u \le x \le x_o \,, \\ &\qquad b_u \le Ax \le b_o \,, \\ &\qquad c_u \le c(x) \le c_o\} \,. \end{aligned}$$

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } Ax \in \boldsymbol{b} \\ &\qquad c(x) \in \boldsymbol{c} \\ &\qquad x \in \boldsymbol{x}, \end{aligned}$$

where $\boldsymbol{x} \in \mathbb{IR}^n$. Here A is a matrix of dimension `nlin`$\times$`n` and c is a vector-valued function of dimension `nonlin`. Since there is no internal preprocessor, in case that an interval is thin, an explicit equality constraint is used internally. Under weak assumptions on the problem one can show that the solver determines a point satisfying the first order optimality conditions from arbitrary initial guesses.

Since the algorithm makes no use of sparse matrix techniques, its proper use will be limited to small and medium sized problems with dimensions up to 500 (for the number of unknowns) say. The number of inequality constraints however may be much larger. (e.g. the code did solve the ecker-kupferschmid-marin "robot-design" examples in sijsc 15, 1994, with n=51 unknowns and 3618 inequalities successfully within 400-900 sec's depending on the variation of the problem on a hp9000/735 workstation). Due to a fully dynamic memory management, there are no hardcoded limits. The minimum memory requirement for a problem with 10 variables and 10 constraints, using 40 steps at most is about 0.2 MB and for a problem with 100 variables and 1500 constraints about 60MB.

Service solve

`donlp2` computes a local solution to a nonlinear programming problem satisfying the first order necessary optimality conditions.

Given a work node the code evaluates the model DAG and extracts the necessary information from this. This is individual information about the coefficients of the linear constraints, the bounds on the variables and the constraints and the values of the objective function and the nonlinear constraints as well as their gradients.

The model is required to represent a differentiable function.

The code returns with an approximate local minimizer, the multipliers from the multiplier rule and the corresponding function values. It also collects some statistical information and gives an indicator of the problems sensitivity.

Control parameters: For detailed explanation, see the user guide of `donlp2`. Only parameters without default must be set by the user.

Purpose	Name	Type	Default
Bound for constraint violation during run	tau0	double	1.0
Bound for considering an inequality as active	del0	double	1.0
Tolerated infeasibility in solution	delmin	double	1.e-6
Tolerated error in multiplier rule	epsx	double	1.e-5
Cold start indicator	cold	bool	true
Tolerated number of small inefficient steps	nreset	int	n

Here eps stands for the precision of the computers floating point arithmetic.

There is an termination indicator (int) and a corresponding text. For the explanation of these indicators, see the user guide.

1. `'CONSTRAINT EVALUATION RETURNS ERROR WITH CURRENT POINT'` (-9)

2. `'OBJECTIVE EVALUATION RETURNS ERROR WITH CURRENT POINT'` (-8)
3. `'QPSOLVER: EXTENDED QP-PROBLEM SEEMINGLY INFEASIBLE '` (-7)
4. `'QPSOLVER: NO DESCENT FOR INFEAS FROM QP FOR TAU=TAU_MAX'` (-6)
5. `'QPSOLVER: ON EXIT CORRECTION SMALL, INFEASIBLE POINT'` (-5)
6. `'STEPSIZESELECTION: COMPUTED D FROM QP NOT A DIR. OF DESCENT'` (-4)
7. `'MORE THAN MAXIT ITERATION STEPS'` (-3)
8. `'STEPSIZESELECTION: NO ACCEPTABLE STEPSIZE IN [SIGSM,SIGLA]'` (-2)
9. `'SMALL CORRECTION FROM QP, INFEASIBLE POINT'` (-1)
10. `'KT-CONDITIONS SATISFIED, NO FURTHER CORRECTION COMPUTED'` (=0)
11. `'COMPUTED CORRECTION SMALL, REGULAR CASE '` (1)
12. `'STEPSIZESELECTION: X ALMOST FEASIBLE, DIR. DERIV. VERY SMALL'` (2)
13. `'KT-CONDITIONS (RELAXED) SATISFIED, SINGULAR POINT'` (3)
14. `'VERY SLOW PRIMAL PROGRESS, SINGULAR OR ILLCONDITONED PROBLEM'` (-16)
15. `'MORE THAN NRESET SMALL CORRECTIONS IN X '` (-15)
16. `'CORRECTION FROM QP VERY SMALL, ALMOST FEASIBLE, SINGULAR '` (-14)
17. `'NUMSM SMALL DIFFERENCES IN PENALTY FUNCTION,TERMINATE'` (-13)

The code collects some information which allows, considered relative to the dimension `n` of the problem a assessment of the problems difficulty. This is:

1. num_of_iter : number of steps total .
2. num_of_f : number of objective function evaluations .

3. num_of_gradf: number of objective gradient evaluations.
4. num_of_con : number of nonlinear constraint evaluations.
5. num_of_gradcon: number of nonlinear constraints gradients evaluations.
6. num_of_restarts: number of restarts for the quasi Newton Hessian update.
7. num_of_full_upd: number of quasi Newton updates without modification.
8. num_of_upd : number of all quasi Newton updates.
9. num_of_full_sqp : number of steps with problem regularization through addition of slacks.

6.1.5 Exclusion boxes

`Exclusion box KJISLP` is an annotation generator which generates exclusion regions.

Service generate

`Exclusion box KJISLP` computes two boxes B_i and B_x which have the following property: B_i contains at least one (or exactly one) Karush-John point for the presented model, $B_i \subset B_x$, and $B_x \backslash B_i$ does not contain a Karush-John point.

Given an approximate Karush-John point z we iteratively produce boxes B_r centered at z with radius r. In these boxes we compute the interval slope $F[z, B_r]$ for all equality constraints, an approximate midpoint inverse $C(r)$, and the matrix $H(r) = F[z, B_r]C(r)$. Then we find a zero $\tilde{r}$ for $f(r) = \|H(r) - I\| - 1$ using a secant method, and set $B_x = B_{\tilde{r}}$.

B_i for uniqueness is computed by using the same test with the interval derivative $F'[\mathbf{x}]$ inside $\frac{1}{2}B_r$ and applying the Krawczyk operator iteratively.

Any work node containing the Karush-John conditions and an (approximate) Karush-John point can be processed.

The code returns an `exclusion_box_delta` containing B_x and a `uniqueness_area_delta` containing B_i.

The statistics information collected is the quotient of the radii of B_x and B_r and of the radii of $\mathbf{x}$ (the original box) and of B_r.

Control parameters: There are two *boolean* control parameter:

`exclusion_only`: Only construct an exclusion box B_x. Default: `false`

`with_uniqueness`: Construct B_i using interval derivatives such that uniqueness of the Karush-John point is guaranteed. Default: `true`

In addition we have the control parameter `center` of type `vector<double>` which specifies the center at which the slopes are being computed. The default value for this parameter is the safeguarded midpoint of the enclosures of all the variables.

Termination reason:

`'SUCCESS'` (0)

`'NO MODEL DEFINED'` (-1).

Statistical information collected: The ratios of the radii $\mathrm{rad}\,(B_x)/\mathrm{rad}\,(B_i)$, and of the radii $\mathrm{rad}\,(\mathbf{x})/\mathrm{rad}\,(B_x)$.

6.1.6 Karush-John condition generator

`KJ generator` is a constraint generator which generates the Karush–John conditions for the model contained in the work node.

Service generate

`KJ generator` computes a directed acyclic graph representing all operations and constraints needed to code the Karush–John conditions.

Consider the optimization problem

$$\begin{aligned}
&\min f(x)\\
&\qquad\text{s.t.}\quad A_Bx \in \mathbf{b_B},\quad A_Lx \ge b_L,\quad A_Ux \le b_U,\quad A_Ex = b_E,\\
&\qquad\qquad F_B(x) \in \mathbf{F_B},\quad F_L(x) \ge F_L,\quad F_U(x) \le F_U,\quad F_E(x) = F_E
\end{aligned}\tag{6.1}$$

If this problem is represented in the work node passed, the KJ generator produces the corresponding Karuhsh–John conditions. In the generated DAG, for every new variable (Lagrange multiplier) and every new node the `ex_kj` flag is set.

The KJ conditions are generated by adding the following constraints to the existing

DAG.

$$\begin{gathered}
\eta g(x) - A_B^T y_B - A_L^T y_L + A_U^T y_U + A_E^T y_E - F_B'(x)^T z_B \\
-F_L'(x)^T z_L + F_U'(x)^T z_U + F_E'(x) z_E = 0 \\
z_L * (F_L(x) - F_L) = 0, \quad z_L \geq 0, \\
z_U * (F_U(x) - F_U) = 0, \quad z_U \geq 0, \\
z_B * (F_B(x) - \underline{F_B}) * (F_B(x) - \overline{F_B}) = 0, \\
z_B * (F_B(x) - \underline{F_B}) \leq 0, \quad z_B * (F_B(x) - \overline{F_B}) \leq 0, \\
y_L * (A_L x - b_L) = 0, \quad y_L \geq 0, \quad y_U * (A_U x - b_U) = 0, \quad y_U \geq 0, \\
y_B * (A_B x - \underline{b_B}) * (A_B x - \overline{b_B}) = 0, \\
y_B * (A_B x - \underline{b_B}) \leq 0, \\
y_B * (A_B x - \overline{b_B}) \leq 0 \\
\eta + z_B^T z_B + \|z_L\|_1 + \|z_U\|_1 + z_E^T z_E = 1, \quad \eta \geq 0
\end{gathered} \tag{6.2}$$

where $*$ denotes componentwise multiplication of two vectors.

The code returns with a `dag_delta` containing all additional variables (the multipliers), operations and constraints needed to represent the Kuhn-Tucker conditions.

For every node of the generated DAG the semantics information `kj` is set to `true`.

There is no useful statistical information collected, the efficiency is always 1.

There are no control parameters.

There is no termination reason except `'SUCCESS'` (0), and `'NO MODEL DEFINED'` (-1).

6.1.7 Linear relaxations

`Linear Relaxation SLP` is a constraint generator which generates a linear relaxation of the original problem using interval slopes.

Service generate

`Linear Relaxation SLP` computes a directed acyclic graph representing a linear relaxation of the model represented by the work node passed.

For every constraint $f(x) \in \mathbf{b}$ we compute an interval slope enclosure of the form

$$f(x) \in \mathbf{f} + \sum \mathbf{s}_i(\mathbf{x}_i - z_i).$$

Then we compute a linear underestimator by

$$\underline{l}(x) = \underline{f} + \sum \overline{s}_i(\underline{x}_i - z_i) + \frac{\underline{s}_i(\overline{x}_i - z_i) - \overline{s}_i(\underline{x}_i - z_i)}{\overline{x}_i - \underline{x}_i}(x_i - \underline{x}_i).$$

Analogously, we compute a linear overestimator $\overline{l}(x)$.

Then we add the two new constraints

$$\begin{aligned}\overline{l}(x) &\geq \underline{b}\\ \underline{l}(x) &\leq \overline{b}.\end{aligned}$$

All constraints which would involve infinities are left out.

The code returns with a `dag_delta` containing all constraints needed to represent the linear relaxation. Either this delta replaces all original constraints or the delta adds the constraints to the original problem. This is decided by setting the control parameter `full_delta`.

If `full_delta` is `true` the output model is *linear*.

For every node of the generated DAG the semantics information `redundant` is set to `true` if `full_delta` is `false`.

There is no useful statistical information collected, the efficiency is always 1.

Control parameters: There are two *boolean* control parameters:

`with_kj`: If this is `false` use only the original problem to generate a relaxation. If this parameter is set to `true`, also the Karush-John conditions are used to construct the relaxation. Default: `true`

`full_delta`: This decides whether the `dag_delta` adds constraints to the original problem (`false`) or if the generated constraints replace the original constraints in the model (`true`). Default: `false`

In addition we have the control parameter `center` of type `vector<double>` which specifies the center at which the slopes are being computed. The default value for this parameter is the safeguarded midpoint of the enclosures of all the variables.

There is no termination reason except `'SUCCESS'` (0), and `'NO MODEL DEFINED'` (-1).

No useful statistical information is collected.

6.1.8 Convexity detection

`Simple Convexity` is an annotation generator which performs a simple convexity test to all constraints and the objective function.

Service convexity test

`Simple Convexity` tries to determine the convexity information for all nodes of the DAG. It uses a simple propagation algorithm to determine concavity and convexity of expressions.

The code returns with a `semantics_delta` changing the convexity information of all nodes and setting the convexity algorithm type to 1.

There is no useful statistical information collected, the efficiency is always 1.

There are no control parameters, and no termination reason except `'SUCCESS'` (0), and `'NO MODEL DEFINED'` (-1).

6.1.9 STOP

This module was written by E. Petrov in IRIN, University of Nantes. The information in this section is taken from his documentation.

`stop_inf_eng` is a solver which implements the STOP algorithm for bound-constraint minimization with a penalty function approach to constrained minimization. The module depends on

1. JAIL library from IRIN
2. IRIN platform for interval constraints

Service infer The module takes the control data specifying the value f^* to reach and searches for a point either improving v or exceeding it by some fraction of $|f^*|$ returns the point delta specifying the best point and the best function value found the deltas generated by successive calls to infer specify a decreasing sequence of function values.

An object of the class `point_delta` from the COCONUT API is generated which specifies a point feasible wrt. the bound constraints from the current work node

Control Parameters:

1. *Name*: `f_best`

   Type: `double`

   Description: the value f^* of the objective to reach

   Default value: $-\infty$

2. *Name*: `f_to_improve`

   Type: `double`

   Description: the known upper bound on the global minimum of the objective

   Default value: ∞

Termination reason:

Integer: 0 (success)

String:

- "" (the empty string)
- "in `DAG_to_platform_Expression::postorder(...)`: do not know how to handle constant vectors";
- "in `DAG_to_platform_Expression::postorder(...)`: do not know how to handle `EXPRINFO_LIN`";
- "in `DAG_to_platform_Expression::postorder(...)`: do not know how to handle `EXPRINFO_QUAD`";
- "in `DAG_to_platform_Expression::postorder(...)`: DAG node of unexpected type";
- "in `DAG_to_platform_Expression::postorder(...)`: do not know how to handle functions defined by the user";
- "in `DAG_to_platform_Expression::postorder(...)`: do not know how to handle `EXPRINFO_GHOST`".

Description: always successful; the string is the history of the messages from the services called previously; adds a message to the history, if the value supplied throu parameter `f_best` has not been reached

6.1.10 Urupa

This module was provided by Christian Keil from the institute Computer Science III of the Technical University of Hamburg-Harburg. The description of this module is taken from his documentation.

`Urupa` is an underestimator for the objective function of a convex minimization problem. It computes a verified bound for the objective function of a convex optimization problem. It is the implementation of an algorithm developed by C. Jansson [102]. To do this an approximate optimal vertex and its lagrangian parameters are needed as input. No assumptions are made about the quality of these parameters, although the quality of the bound increases with the quality of the approximations. In the case of moderate finite simple bounds the bound is computed just by post processing using simple interval operations. Otherwise the solution of a pertubated linear approximation of the convex problem is used. This may result in a small number iterations (in many cases only between one and three).

At this time all the simple bounds must be finite.

Service bound

`Urupa` computes the bound using the supplied approximate solution and its lagrange parameters. Any convex optimization problem can be passed to the module. The bound for the objective function is returned as `bound` in the `information` entry.

Control parameters:

`solution:` The approximate solution to use for computing the bound.

`lagrange:` The langrange parameters of the approximate optimal solution.

Both parameters are mandatory.

Termination reason:

`'SUCCESS' (0):` The bound was successfully computed.

`'optimization problem not convex' (-1):` The current optimization problem is not convex.

No statistical information is collected at this time. This might be changed in the future to return

$$\frac{|f^*_{app} - \underline{f^*}|}{|f^*_{app}|}$$

as the effectiveness with f^*_{app} denoting the objective value of the approximate optimum and $\underline{f^*}$ denoting the computed bound.

6.2 Graph Analyzers

At the moment there is only one graph analyzer module available.

6.2.1 Box Chooser

`Box Chooser` is a module which chooses a new work node from the search graph.

Service choose

`Box Chooser` chooses one of the leafs of the search graph as new work node and returns its search node id through the `node id` parameter in the `information` structure.

Control parameters

`method:` This parameter is of type `int` specifies the method used to determine the next box.

0. Choose the box randomly.
1. Choose the box with lowest entry in some table.

2. Choose the box with highest entry in some table.

`table`: The name of the table to search in.

`col`: The name of the column, in which the minimum/maximum should be found.

6.3 Management Modules

This section is devoted to the management modules available in the COCONUT environment.

6.3.1 Delta Management

`Delta Management` is a module which performs various management tasks for the deltas in a work node

Service compress

`Delta Management` collects all deltas of a work node of a specific type and compresses them into one delta performing the same change. Until now this is only implemented for the `bound_delta` type.

The return value is always 0.

Control parameters:

`type`: This parameter is of type `string` and specifies the class of deltas on which the change is performed.

`max gainfactor`: With this parameter of type `double` for compression of `bound_deltas` (type is `bounds`) the maximum allowed gainfactor for changes in a single variable are specified. I.e., setting `max gainfactor` to 0.5 would ignore all changes in single components which do not at least halve the width of the node range. The default is `1.`.

Service undo

`Delta Management` unapplies all deltas of a work node of a specific type. The return value is always 0.

Control parameters:

`type`: This parameter is of type `string` and specifies the class of deltas on which the change is performed.

Service undo last

`Delta Management` unapplies the last n deltas of a work node of a specific type. The return value is always 0.

Control parameters:

`type`: This parameter is of type `string` and specifies the class of deltas on which the change is performed.

`n`: With this parameter of type `unsigned int` the number of deltas to be unapplied is specified. The default is `1`..

6.3.2 Focus Management

`Focus Management` is a module which manipulates the focus of the search graph.

Service focus

`Focus Management` initializes a new focus. The return value is 0 if there is a node to which the focus can be initialized and 1 otherwise.

Service kill focus

`Focus Management` removes the node the focus is pointing to from the search graph and destroys the focus. The return value is 0.

Service promote focus

`Focus Management` promotes the node the focus is pointing to to its parent node and moves the focus accordingly. The return value is 0 if promoting was successful and 1 otherwise (i.e., if the parent has more than one child).

Service destroy focus

`Focus Management` removes the focus. The return value is 0.

6.3.3 Full Model Management

`Full Model Management` is a module which updates the model in the work node absolutely. This means that all changes and deltas performed in the work node are written into the model specification, hereby generating a work node which can be stored as `full_node` into the search graph.

Service model update

`Full Model Management` writes all DAG updates, semantics changes, and the bound changes into the model DAG and stores all annotation deltas in the search database. The return value is always 0.

6.3.4 Solution Management

`Solution Management` is a module which manages the `solution` table in the search database.

Service add

`Solution Management` stores the box represented by the work node in the `solution` table of the search database. The return value is 0.

Service add/remove

`Solution Management` stores the box represented by the work node in the `solution` table of the search database and removes the node from the search graph. The return value is 0 if the node was removed and if it was not the last child of its parent. The return value is 1 if the removed node was the last child of its parent.

Service purge

`Solution Management` removes all boxes from the `solution` table, whose lower bound of the objective function is bigger than a given threshold. The return value is 0.

Control parameters:

`threshold`**:** This parameter is of type `double` specifies the threshold.

6.3.5 Split Management

`Split Management` is a module which performs splits on the work node and stores the generated nodes in the search graph. It is possible to update the work node and the focus in such a way that one of the splits is chosen as new work node.

Service split

`Split Management` performs one of the proposed splits to the work node and stores all generated nodes in the search graph.

One of the splits can be kept to update the work node and reset the focus to the updated work node. The function returns 0 if a split was performed and -1 if no suitable split could be found.

Control parameters:

`transaction`: This parameter is of type `unsigned int` and specifies the entry number in the proposed splits map of the work node for the split to be chosen.

`max splits`: With this parameter of type `int` the maximum number of allowed splits is specified. The default is `INT_MAX`.

`min splits`: With this parameter of type `int` the minimum number of allowed splits is specified. The default is `1`.

`min gain`: This parameter of type `double` specifies the minimum gain in box size in every produced box. The default is `1`.

`keep`: Using this `bool` the user can specify whether one of the splits ought to be kept as new work node. Default: `false`.

`keep type`: This `int` codes which algorithm is used to determine which node to keep if `keep` is `true`:

- **`SPLIT_MGMT_KEEP_BIGGEST` (0):** The node with biggest `log_vol` is kept. If many boxes have the same volume, keep the first.
- **`SPLIT_MGMT_KEEP_FIRST` (1):** The first node is kept.
- **`SPLIT_MGMT_KEEP_BIGGEST_RANDOM` (2):** The node with biggest `log_vol` is kept. If many boxes have the same volume, keep any, randomly selected.
- **`SPLIT_MGMT_KEEP_RANDOM` (3):** Keep any box, randomly selected.
- **`SPLIT_MGMT_KEEP_PROM_CENTER` (4):** Evaluated a weighted sum of objective function value and constraints. Take the box with lowest value at the center.
- **`SPLIT_MGMT_KEEP_PROM_LMPOINT` (4):** Evaluated a weighted sum of objective function value and constraints. Take the box with lowest value at the most promising point in the box, which is computed by using a linear model.
- **`SPLIT_MGMT_KEEP_SPEC` (6):** The n–th generated node is kept, where n is determined by the parameter `keep idx`.

`keep idx`: This `unsigned int` specifies which node should be kept if `keep type` is `SPLIT_MGMT_KEEP_SPEC`.

Service clear splits

`Split Management` clears the list of proposed splits in the work node. The function returns 0.

6.4 Report Modules

This section gives a description of the report modules, which are available.

6.4.1 Range Report

`Ranges Write` is a report module which prints the ranges of all variable nodes or of all nodes of the work node.

Service print

`Ranges Write` prints the ranges of all variable nodes or of all nodes either in compact format as an array or in large format containing node numbers or variable names.

Control parameters:

`preamble`**:** This parameter is of type `string` specifies what is printed before the ranges. The default is `"ranges:¨`.

`postamble`**:** Also of type `string`, this parameter is used to define what is printed after the ranges. The default is `"\n\n"`.

`compact`**:** With this parameter of type `bool` it is decided whether the ranges are print as a comma separated list of intervals, or whether it is printed with additional information like node numbers or variable names. The default is `true`.

`variables`**:** Using this `bool` the user can specify whether all node ranges are printed or only the ranges of the variable nodes. Default: `false`.

`entries per line`**:** This integer specifies how many entries are printed per line of output. Default: `4`.

6.4.2 Table Report

`Table Report` is a report module which prints the entries of a specified table from the search database.

Service print

`Table Report` prints specified rows from one table of the search database. The user can decide whether all table entries should be printed or whether only entries from the work node view should be reported.

Control parameters:

`table`: This parameter is of type `string` specifies entries of which table are printed.

`col`: This `int`-indexed parameter of type `string` specifies the column name of the ith output.

`enable`: This `int`-indexed parameter of type `bool` specifies whether ith output should be printed. The default is `true`.

`prefix`: This `int`-indexed parameter of type `bool` specifies what is printed before the value of the ith output row.

In addition there are a number of `string` parameters influencing the format of the output.

`header`: This string is printed before row entries are printed. Default: `""`

`footer`: This is printed after all rows. Default: `"\n"`

`separator`: This string is printed between two rows. Default: `"\n\n"`

`interdisp`: This is the separator of two columns of the same row. Default: `"\n"`

6.4.3 DAG Writer

`DAG Write` is a module which outputs the DAG of the work node in sequentialized format suitable for reading.

Service print

`DAG Write` produces an ASCII serialized representation of the model DAG in the work node.

6.4.4 GAMS Report

`GAMS Write` is a report module which produces a `GAMS` model corresponding to the model stored in the work node.

Service print

GAMS Write produces the GAMS model.

6.4.5 Control parameters

file name: This string parameter sets the filename, from which the model was generated for including it into the comments. The default is the empty string.

print default bounds: This parameter is of type bool and specifies whether default bounds for all variables should be added, even if the variables are unbounded in the internal representation. The default is false.

default bounds: With this parameter of type double the user sets the absolute value of the default bounds. The default is 1000.

In addition there is a number of string parameters which can be used to change function and variable names and insert additional code into between the generated C expressions.

objective constraint: The name of the equation for the objective function.

constraint: The name of the constraints

optimization direction: This must be min or max.

solver name: The name of the solver to be called from GAMS.

model name: The name of the GAMS model.

x: The name of the variables array.

obj: The name of the objective function.

6.4.6 Solution Report

Solution Report is a report module which prints the solution information.

Service print

Solution Report prints the best points from the point table and the entries of the solution table.

6.4.7 DAG to C converter

C Write is a report module which produces C code for evaluating all constraints and the objective function of the model stored in the work node.

Service print

`C Write` produces two `C` functions, one for evaluating the objective and one for evaluating the constraints.

Control parameters:

`print multipliers`**:** This parameter of type `bool` specifies whether the variables associated with the Karush-John conditions (i.e. the multipliers) are printed as `y` and the multiplier of the gradient of the objective as `k`. If the value is `false` all variables are printed alike. The default is `true`.

`with comments`**:** With this parameter of type `bool` the user decides whether additional comments like bounds and node numbers are printed within the generated `C` code. The default is `true`.

In addition there is a number of string parameters which can be used to change function and variable names and insert additional code into between the generated `C` expressions.

header: The header part of the `C` file.

objective preamble: The part before the objective function.

objective postamble: The part between function definition and the generated expression.

constraints preamble: The part before the constraints function.

constraints postamble: The part between function definition and the generated expression.

switch preamble: The part before the `switch` which distinguishes between the different constraints.

switch postamble: The function part after the `switch`.

constraints function end: The remaining part of the constratints function.

footer: The end of the `C` file.

x: The name of the variables array.

y: The name of the multipliers array.

kappa: The name of the multiplier of the gradient of the objective function.

type: The type of the variables and multipliers.

kappatype: The type of the multiplier of the gradient of the objective function.

6.4.8 DAG to Fortran 90 converter

`GLOBSOL Write` is a report module which produces a `GLOBSOL` model corresponding to the model stored in the work node. A `GLOBSOL` model consists of two files, a `FORTRAN 90` representation of the constraints and the objective function and an additional control file.

Service F90 file

`GLOBSOL Write` produces the `FORTRAN 90` file for the model.

Control parameters:

`file name`: This `string` parameter sets the filename, from which the model was generated for including it into the comments. The default is the empty string.

In addition there is a number of string parameters which can be used to change function and variable names and insert additional code into between the generated `C` expressions.

equality constraints: The name of the array for the equality constraints.

inequality constraints: The name of the array for the inequality constraints.

`x`: The name of the variables array.

Service control file

`GLOBSOL Write` produces the control file for the `GLOBSOL` model.

Control parameters:

`file name`: This `string` parameter sets the filename, from which the model was generated for including it into the comments. The default is the empty string.

`default bounds`: With this parameter of type `double` the user sets the absolute value of the default bounds for variables. The default is `1.0e5`.

`default accuracy`: With this parameter of type `double` the user sets the default accuracy for boxes. The default is `1.0e-8`.

6.4.9 DAG to GAMS converter

`GAMS Write` is a report module which produces a `GAMS` model corresponding to the model stored in the work node.

Service print

`GAMS Write` produces the `GAMS` model.

Control parameters:

`file name`**:** This `string` parameter sets the filename, from which the model was generated for including it into the comments. The default is the empty string.

`print default bounds`**:** This parameter is of type `bool` and specifies whether default bounds for all variables should be added, even if the variables are unbounded in the internal representation. The default is `false`.

`default bounds`**:** With this parameter of type `double` the user sets the absolute value of the default bounds. The default is `1000`.

In addition there is a number of string parameters which can be used to change function and variable names and insert additional code into between the generated `C` expressions.

objective constraint: The name of the equation for the objective function.

constraint: The name of the constraints

optimization direction: This must be `min` or `max`.

solver name: The name of the solver to be called from `GAMS`.

model name: The name of the `GAMS` model.

x: The name of the variables array.

obj: The name of the objective function.

Appendix A
The COCONUT environment, `C++` code

This appendix is a `C++` reference of the most important classes and methods of the API, the VGTL, and the VDBL. The section numbering is chosen to exactly match the section numbering in Chapter 5, where the software structure is described.

A.1 Vienna Graph Template Library (VGTL)

A.1.1 Core components

C array to vector adaptor

```
template <class _T>
class array_vector : public std::vector<_T>
{
public:
  array_vector();

  // constructor building an array_vector from pointer __a with size n
  array_vector(_T* __a, int n);

  ~array_vector();

  // assign an array __a of length n to this array_vector.
  void assignvector(_T* __a, int n);
};
```

Reverse search

```
template <class BidirectionalIterator, class T>
BidirectionalIterator rfind(BidirectionalIterator first,
                            BidirectionalIterator last,
                            const T& val);

template <class BidirectionalIterator, class Predicate>
BidirectionalIterator rfind_if(BidirectionalIterator first,
                               BidirectionalIterator last,
                               Predicate pred);
```

A.1.2 Walker

Recursive Walker

```
unsigned int n_children() const;
unsigned int n_parents() const;
bool is_root() const;
bool is_leaf() const;
bool is_ground() const;
bool is_sky() const;

children_iterator child_begin();
children_iterator child_end();
```

```
parents_iterator parent_begin();
parents_iterator parent_end();

bool operator==(const walker&) const;
bool operator!=(const walker&) const;

walker operator<<(parents_iterator);
walker operator>>(children_iterator);
walker& operator<<=(parents_iterator);
walker& operator>>=(parents_iterator);

walker& operator=(const walker&);
```

A.1.3 Container

Directed Graphs and DAGs

```
template <class T>
class dag : dag_base<T>   // this is from vgtl_dagbase.h and defines
{
private:
  typedef dag<T> _Self;
  typedef dag_base<T> _Base;
  typedef dag_walker<T> walker;

  // constructors, destructor
  dag();
  dag(const _Self& __dag);
  ~dag();

  // assignment
  _Self& operator=(const _Self& __x);

  // comparison
  friend bool operator== (const _Self& __x, const _Self& __y);

  bool empty() const;

  void clear();

  // here the exact point of insertion can be chosen
  walker between(const walker& parent, const children_iterator& cit,
                 const walker& child, const parents_iterator& pit,
                 const T& x);

  // here the new edge is appended to the list of edges in the parents
  // and children vectors
```

```
walker between_back(const walker& parent, const walker& child,
                    const T& x);

// here the new edge is prepended to the list of edges in the parents
// and children vectors
walker between_front(const walker& parent, const walker& child,
                     const T& x);

// insert between a list of parents and children. The new edges are
// appended. The following calls work for any sequence containers SequenceCtr
walker between(const SequenceCtr<walker>& parents,
               const SequenceCtr<walker>& children,
               const T& x);

// insert between one parent and a list of children
walker between(const walker& parent, const children_iterator& cit,
               const SequenceCtr<walker>& children,
               const T& x);
walker between_back(const walker& parent,
                    const SequenceCtr<walker>& children,
                    const T& x);
walker between_front(const walker& parent,
                     const SequenceCtr<walker>& children,
                     const T& x);

// insert between one child and many parents
walker between(const SequenceCtr<walker>& parents,
               const walker& child, const parents_iterator& pit,
               const T& x);
walker between_back(const SequenceCtr<walker>& parents,
                    const walker& child, const T& x);
walker between_front(const SequenceCtr<walker>& parents,
                     const walker& child, const T& x);

// here the exact point of insertion can be chosen
walker split(const walker& parent, const children_iterator& cit,
             const walker& child, const parents_iterator& pit,
             const T& x);

// here the new edge is appended to the list of edges in the parents
// and children vectors
walker split_back(const walker& parent, const walker& child,
                 const T& x);

// here the new edge is prepended to the list of edges in the parents
// and children vectors
walker split_front(const walker& parent, const walker& child,
                   const T& x);

// insert between a list of parents and children. The new
// edges are appended. Again this call works for any sequence
// container _SequenceCtr.
walker split(const SequenceCtr<walker>& parents,
             const SequenceCtr<walker>& children,
             const T& x);

// insert between one parent and a list of children
walker split(const walker& parent, const children_iterator& cit,
             const SequenceCtr<walker>& children,
             const T& x);
walker split_back(const walker& parent,
                  const SequenceCtr<walker>& children,
                  const T& x);
walker split_front(const walker& parent,
                   const SequenceCtr<walker>& children,
                   const T& x);

// insert between one child and many parents
walker split(const SequenceCtr<walker>& parents,
             const walker& child, const parents_iterator& pit,
             const T& x);
walker split_back(const SequenceCtr<walker>& parents,
                  const walker& child, const T& x);
walker split_front(const SequenceCtr<walker>& parents,
                   const walker& child, const T& x);

// insert a whole subgraph
void insert_subgraph(_Self& subgraph,
                     const std::vector<walker>& parents.
                     const std::vector<walker>& children);

// add a single edge
// where you want in the lists
void add_edge(const walker& parent, const children_iterator& ch_it,
              const walker& child, const parents_iterator& pa_it);
// always in the end
void add_edge_back(const walker& parent, const walker& child);
// always in the front
void add_edge_front(const walker& parent, const walker& child);

// remove an edge
void remove_edge(const walker& parent, const walker& child);

/** remove one egde and don't reconnect the node to sky/ground */
void remove_edge_and_deattach(const walker& parent,
                              const walker&child);

/**
 * change the edge from @c parent to @c child_old to an edge
 * from @c parent to @c child_new.
```

```
   */
  void replace_edge_to_child(const walker& parent, const walker& child_old,
                             const walker& child_new);

  /**
   * change the edge from @c parent_old to @c child to an edge
   * from @c parent_new to @c child.
   */
  void replace_edge_to_parent(const walker& parent_old,
                              const walker& parent_new,
                              const walker& child)

  // erase a node from the graph
  void erase(const walker& position);

  // erase a child node if it is a leaf
  bool erase_child(const walker& position, const children_iterator& It);

  // erase a parent node if it is a root
  bool erase_parent(const walker& position, const parents_iterator& It);

  /** merge two nodes, call also the merge method for the node data */
  void merge(const walker& position, const walker& second,
             bool merge_parent_edges = true, bool merge_child_edges = true);

  erased_part erase_minimal_subgraph(const walker& position)
  erased_part erase_maximal_subgraph(const walker& position)

  // this works for every sequential container SequenceCtr.
  erased_part erase_maximal_subgraph(
      const SequenceCtr<walker,_Allocator>& positions)
  erased_part erase_minimal_subgraph(
      const SequenceCtr<walker,_Allocator>& positions)

  erased_part erase_minimal_pregraph(const walker& position)
  erased_part erase_maximal_pregraph(const walker& position)

  // this works for every sequential container SequenceCtr.
  erased_part erase_maximal_pregraph(
      const SequenceCtr<walker,_Allocator>& positions)
  erased_part erase_minimal_pregraph(
      const SequenceCtr<walker,_Allocator>& positions)

  walker ground();
  walker sky();

  const_walker ground() const;
  const_walker sky() const;
};
```

A.1.4 Algorithms and Visitors

A.2 Vienna Database Library (VDBL)

A.2.1 Database

The methods of the `database` class:

```
/**
 * create a new table
 *   - _C_i: name
 *   - _C_u: user id
 *   - __f: the table flags (if they are not default)
 * return true, if creating the table was successful.
 */
bool create_table(const std::string& _C_i, const userid& _C_u,
                  const tableflags& __f = tableflags());
bool create_table(const char* _C_i, const userid& _C_u,
                  const tableflags& __f = tableflags());

/**
 * return the table id for a given name
 */
tableid get_tableid(const std::string& _C_i,
                    const userid& _C_u) const;

/**
 * delete a table, whose table id is provided.
 * return true, if deleting the table has worked.
 */
bool drop_table(const tableid& _C_i, const userid& _C_u);
/**
 * delete a table, whose name is provided.
 * return true, if deleting the table has worked.
 */
bool drop_table(const std::string& _C_i, const userid& _C_u);
bool drop_table(const char* _C_i, const userid& _C_u)

/**
 * check whether the table with id_C_i exists
 */
bool has_table(const tableid& _C_i, const userid& _C_u) const;
/**
 * check whether the table with name_C_i exists
 */
bool has_table(const std::string& _C_i, const userid& _C_u) const;
bool has_table(const char* _C_i, const userid& _C_u) const
```

```
/**
 * return a pointer to the table with id _C_i.
 */
table* get_table(const tableid& _C_i, const userid& _C_u) const;
/**
 * return a pointer to the table with name _C_i.
 */
table* get_table(const std::string& _C_i, const userid& _C_u) const;
table* get_table(const char* _C_i, const userid& _C_u) const;

/**
 * create a new standard view with name _C_i, evaluation context __c,
 * for table _C_t, of type __e.
 * return true if creating worked, and false otherwise.
 */
bool create_view(const std::string& _C_i, const userid& _C_u,
                 const context& __c,
                 const std::string& _C_t, const _V_enum& __e)
bool create_view(const char* _C_i, const userid& _C_u,
                 const context& __c,
                 const std::string& _C_t, const _V_enum& __e)
bool create_view(const std::string& _C_i, const userid& _C_u,
                 const context& __c,
                 const char* _C_t, const _V_enum& __e)
bool create_view(const char* _C_i, const userid& _C_u,
                 const context& __c,
                 const char* _C_t, const _V_enum& __e)

/**
 * return the view id of view _C_i.
 */
viewid get_viewid(const std::string& _C_i, const userid& _C_u) const

/**
 * delete a view, whose id is provided.
 * return true, if deleting the table has worked.
 */
bool drop_view(const viewid& _C_i, const userid& _C_u)

/**
 * delete a view, whose name is provided.
 * return true, if deleting the table has worked.
 */
bool drop_view(const std::string& _C_i, const userid& _C_u)
bool drop_view(const char* _C_i, const userid& _C_u)

/**
 * check whether the view with id _C_i exists
 */
bool has_view(const viewid& _C_i, const userid& _C_u) const
```

```
/**
 * check whether the view _C_i exists
 */
bool has_view(const std::string& _C_i, const userid& _C_u) const
bool has_view(const char* _C_i, const userid& _C_u) const

/**
 * return a pointer to the view with id _C_i.
 */
view* get_view(const viewid& _C_i, const userid& _C_u) const
/**
 * return a pointer to the view with name _C_i.
 */
view* get_view(const std::string& _C_i, const userid& _C_u) const
view* get_view(const char* _C_i, const userid& _C_u) const
```

A.2.2 Tables

This section contains the method reference for the standard table class as it is used in the search database of the COCONUT environment.

Standard Table

```
typedef std::pair<const std::string*,const _VDBL_col*> ptrcolspec;
typedef std::pair<std::string,_VDBL_col> colspec;


/**
 * constructor defining a table using a list of columns. This list can
 * be contained in any STL sequence container.
 */
template <template <class __Tp, class __AllocTp> class __SequenceCtr,
          class Allocator1>
standard_table(const __SequenceCtr<triple<std::string, col,
               colflags>,Allocator1>& __cc);

/**
 * return iterator to first column
 */
col_const_iterator col_begin() const;
/**
 * return iterator beyond last column
 */
col_const_iterator col_end() const;

/**
```

```
 *  return iterator to first row
 */
row_const_iterator row_begin() const;
/**
 *  return iterator beyond last row
 */
row_const_iterator row_end() const;

/**
 * add a new column of name _C_n, which contains column __c, and has
 * column flags __f.
 * The function returns true, if adding the column was successful.
 */
bool add_col(const std::string& _C_n, const col& __c,
             const colflags& __f);
/**
 * add a new column of name _C_n, with data __c, and column flags __f.
 * The function returns true, if adding the column was successful.
 */
template <class _CB>
bool add_col(const char* _C_n, const _CB& __c, const colflags& __f);

template <class _CB>
bool add_col(const std::string& _C_n, const _CB& __c,
             const colflags& __f);

/**
 * modify the column of name _C_n, to new contents __c, and new
 * column flags __f.
 * The function returns true, if modifying was successful.
 */
bool modify_col(const std::string& _C_n, const col& __c,
                const colflags& __f);
/**
 * modify the contents of column of name _C_n
 * The function returns true, if modifying was successful.
 */
bool modify_col(const std::string& _C_n, const col& __c);
/**
 * modify the column flags of column of name _C_n
 * The function returns true, if modifying was successful.
 */
bool modify_col(const std::string& _C_n, const colflags& __f);


/**
 * remove the column _C_n from the table
 * The function returns true, if erasing was successful.
 */
bool drop_col(const std::string& _C_n);
```

```
/**
 * rename the column _C_old to new name _C_new.
 * The function returns true, if renaming was successful.
 */
bool rename_col(const std::string& _C_old, const std::string& _C_new);


/**
 * insert a new row of specification _row into the table.
 * The function returns true, if inserting was successful.
 */
bool insert(const std::vector<ptrcolspec>& _row);
/**
 * insert a new row of specification _row into the table, and
 * return the row id of the newly created row in _r.
 * The function returns true, if inserting was successful.
 */
bool insert(const std::vector<ptrcolspec>& _row, rowid& _r);
/**
 * insert a new row of specification _row into the table, and
 * return the row id of the newly created row in _r.
 * Take any sequential STL container to hold the row entries of the
 * column.
 * The function returns true, if inserting was successful.
 */
template <template <class __Tp, class __AllocTp> class __SequenceCtr,
          class Allocator1>
bool insert_row(const __SequenceCtr<colspec,Allocator1>& _row);
/**
 * insert many new rows of specifications _rows into the table.
 * The list of rows can be contained in any sequential STL container,
 * which holds any other sequential STL container of column entries.
 * The function returns true, if inserting was successful for all
 * rows.
 */
template <template <class __Tp1, class __AllocTp1> class __SequenceCtrOut,
          template <class __Tp2, class __AllocTp2> class __SequenceCtrIn,
          class AllocatorOut, class AllocatorIn>
bool insert_row(const __SequenceCtrOut<
                             __SequenceCtrIn<colspec,AllocatorIn>,
                             AllocatorOut>& _rows);

/**
 * remove the row with it _ri from the table
 */
bool remove(const rowid _ri);

/**
 * return whether the column _C_n is defined in the table
 */
bool has_col(const std::string& _C_n) const;
bool has_col(const char* _C_n) const;

/**
 * return a (const) reference to the row with id _ri. If an error
 * occurs, set error to true, otherwise to false.
 */
const row& get_row(const rowid& _ri, bool& error) const
row& get_row(const rowid& _ri, bool& error);

/**
 * return a (const) reference to the default column with id _ci.
 * If an error occurs, set error to true, otherwise to false.
 */
const col& get_def(const colid& _ci, bool& error) const;
col& get_def(const colid& _ci, bool& error)

/**
 * return whether the column with id _ci has a default value
 */
bool has_def(const colid& _ci) const;

/**
 * return whether the column with id _ci has a value in row _ri
 */
bool has_col(const rowid& _ri, const colid& _ci) const;

/**
 * retrieve the value of column (id = _c) in row (id = _r)
 * under evaluation context _ctx to _val. Return whether
 * retrieval was successful. If there is no defined value
 * in row _r for column _c, then return the default value of
 * column _c, if defined.
 */
bool retrieve(const rowid& _r, const colid& _c,
              const context& _ctx, alltype_base*& _val) const;

colid get_colid(const char* _C_n) const;
```

A.2.3 Columns

The method reference is split in three parts. The first one specifies, general methods valid for all classes. The other two parts define those methods, which are specific for one of the two derived classes.

Column base class

```
  //! set the context for value retrieval
  void setcontext(const context* _c, const row* _r);

  /**
   *  This function stores a copy of the column value into c.
   */
  template <class _R>
  void get(_R& c) const;
  /**
   *  This function stores a copy of the column default value into d.
   */
  template <class _R>
  void def(_R& d) const;
  /**
   * This function sets c to a const pointer pointing to the
   * column's actual value. Here, no copying is done.
   */
  template <class _R>
  void get_ptr(_R const *& c) const;
  /**
   * This function returns a pointer to a copy of the column's value.
   * The copy of the value is allocated by new. It has to be
   * deleted by the user to avoid memory leaks.
   */
  template <class _R>
  void get_copy(_R*& p) const;
  /**
   * This function returns a pointer to a copy of the column's default
   * value. The copy of the value is allocated by new. It has to be
   * <tt>delete</tt>d by the user to avoid memory leaks.
   */
  template <class _R>
  void def_copy(_R*& p) const;

  /**
   * The print operation for generic columns. This implicitely calls
   * operator<< for the columns type. So it is necessary that this
   * operator is indeed defined.
   */
  friend std::ostream& operator<<(std::ostream& o, const col& c);
};
```

Typed column

```
  /**
   * explicit constructor setting the column's value
   **/
  typed_col(const type& __t);

  /**
   * set the column value
   **/
  void set(const type& __t);

  /**
   *  set the default value for this column. This is actually equivalent
   *  to set, since default and standard columns coincide for constant
   *  values.
   **/
  void set_default(const type& __t);

  /**
   *  get a const reference to the column value
   **/
  const type& get_val() const;
```

Method column

```
  /**
   * constructor for explicitely setting the method
   */
  method_col(const method& __m);
```

Column Evaluation Methods

```
  /**
   * compute the value
   */
  const return_type& operator() ();
  /**
   * compute the default value
   */
  const return_type& def();
  /**
   * set the evaluation context and the evaluation row.
   */
  virtual void setcontext(const context* _c, const _VDBL_row* _r);
};
```

A.2.4 Rows

This section contains the method reference for the row class.

```
/**
 * get a reference to the column with id _id.
 * If the column existed, error will be false,
 * otherwise error will be true.
 */
col& get_col(const colid& _id, bool& error);
const col& get_col(const colid& _id, bool& error) const;

/**
 * return whether a column with id _id exists in this row.
 */
bool has_col(const colid& _id) const;

/**
 * insert the new column _col with id _id in this row.
 * If this id exists, return false, otherwise return true.
 */
bool insert(const colid& _id, const col& _col);

/**
 * remove the column with id _id from this row. Return true
 * if erasing was successful, and false if the column does not
 * exist.
 */
bool drop(const colid& _id);

/**
 * update the column with id _id with the value _col.
 * If the column does not yet exist, insert it. Otherwise,
 * change its value.
 */
void update(const colid& _id, const col& _col);
```

A.2.5 Views

In this section I will describe all methods relevant for views. First those are described which are accessible from all views, and afterwards the more specialized ones are defined.

General Views

```
/**
 * insert a new row of specification _row into the view.
 * The function returns true, if inserting was successful.
 */
bool insert(const std::vector<_T_colspec>& _row);
```

```
/**
 * remove the row with id _ri.second from the table _ri.first.
 */
bool remove(std::pair<tableid,rowid> _ri);

/**
 * return whether the column _C_n is defined in the view
 */
bool has_col(const std::string& _C_n) const;

/**
 * return table id and column id of column _C_n
 */
std::pair<tableid,colid> get_col_id(
                             const std::string& _C_n) const;

/**
 * return a const reference to the column with column id _ci
 * in row _ri.second of table ri.first. If successful,
 * set error to false, and to true otherwise.
 * Note that in this function the context for the column is NOT set.
 */
const col& get_raw_col(const std::pair<tableid,rowid>& _ri,
                       const colid& _ci, row const *& _rr,
                       bool& error) const;

/**
 * print the contents of the column with column id _ci
 * in row _ri.second of table ri.first. If the row
 * is empty and no default is defined, print nothing.
 */
std::ostream& print_col(std::ostream& o,
                        const std::pair<tableid,rowid>& _ri,
                        const colid& _ci, bool& printed) const;

/**
 * return a const reference to the row with row id
 * _ri.second of table _ri.first. If successful,
 * set error to false, and to true otherwise.
 */
const row& get_row(const std::pair<tableid,rowid>& _ri,
                   bool& error) const;

/**
 * return a const reference to the default column with column id
 * _ri.second of table ri.first. If successful,
 * set error to false, and to true otherwise.
 */
const col& get_def(const std::pair<tableid,colid>& _ri,
                   bool& error) const;
```

```
/**
 * return iterator to first (beyond last) default column
 */
default_const_iterator defaults_begin() const;
default_const_iterator defaults_end() const;

/**
 * return iterator to first (beyond last) row
 */
row_const_iterator rows_begin() const;
row_const_iterator rows_end() const;

/**
 * return iterator to first (beyond last) column
 */
col_const_iterator cols_begin(const rowid& _r) const;
col_const_iterator cols_end(const rowid& _r) const;

/**
 * return the type of this view
 */
_V_enum view_type() const;
```

Standard View

```
/**
 * standard constructor which initalizes the table and the tableid,
 * the evaluation context, and the view type.
 */
view(const tableid& __ti, table* __t,
     const context& __c, _V_enum __e);
/**
 * standard constructor which initalizes the table and the tableid,
 * the evaluation context, and the view type. In addition the vector
 * _rs contains a list of rows, which should be visible in this view.
 */
view(const tableid& __ti, table* __t,
     const context& __c, _V_enum __e,
     const std::vector<rowid>& _rs);
/**
 * get the data from column _ci in row _ri.second of
 * table _ri.first. The data stored in the column must
 * be of type _R.
 */
template <class _R>
bool get(const std::pair<tableid,rowid>& _ri,
         const colid& _ci, _R& r) const
template <class _R>
```

```
bool get(const tableid& _ti, const rowid& _ri,
         const colid& _ci, _R& r) const;
template <class _R>
bool get(const rowid& _ri, const std::string& _c, _R& r) const;
template <class _R>
bool get(const rowid& _ri, const char* _c, _R& r) const;

/**
 * get a const ptr to the data from column _ci in row
 * _ri.second of table _ri.first. The data
 * stored in the column must be of type _R. In this function
 * no data copying is done. Note that this function returns a
 * pointer to the columns raw data, so it can only be used to
 * refer to constant columns.
 */
template <class _R>
bool get_raw_ptr(const std::pair<tableid,rowid>& _ri,
                 const colid& _ci, _R const *& r) const;
template <class _R>
bool get_raw_ptr(const tableid& _ti, const rowid& _ri, const colid& _ci,
                 _R const *& r) const;
```

Hierarchical View

```
/**
 * standard constructor which initalizes the table and the tableid of
 * the master table, the evaluation context, and the view type.
 */
hierarchicalview(const tableid& __ti, table* __t,
                  const context& __c, _V_enum __en);

/**
 * standard constructor which initalizes the table and the tableid
 * of the master table, the evaluation context, and the view type.
 * In addition the vector _rs contains a list of rows, which should
 * be visible in this view.
 */
hierarchicalview(const tableid& __ti, table* __t,
                 const context& __c, _V_enum __en,
                 const std::vector<rowid>& _rs);

/**
 * This pushes a new table onto the top of the hierarchical view stack.
 */
void push_table(const tableid& __ti, table* __t);

/**
 * This pushes a new table onto the top of the hierarchical view stack.
 * Additionally, a subset of the table's rows, which are visible in
```

```
   * the view, can be specified.
   */
  void push_table(const tableid& __ti, table* __t,
                  const std::vector<rowid>& _rs);

  /**
   * remove the topmost table from the view, and return its table id.
   */
  tableid pop_table();
```

A.2.6 View Database

The following section is the method reference for the `viewdbase` class.

```
  /**
   * constructor which builds a view to the database from
   *   - db -- the database
   *   - c  -- the evaluation context for all views
   */
  viewdbase(const database& db, const userid& uid, const context& c);
  /**
   * constructor which builds a view to the database from
   *   - db  -- the database
   *   - c   -- the evaluation context for all views
   *   - uid -- the user id of the user who owns the view
   * The fourth argument is any sequential container of table,row
   * pairs to which the view shall be restricted.
   */
  template <template <class _C, class _A> class _SqCtr, class _Al>
  viewdbase(const database& db, const userid& uid, const context& c,
            const _SqCtr<std::pair<tableid,rowid>,_Al>& __an);

  /**
   * return the table id (and view id) of table _C_i
   */
  tableid get_tableid(const std::string& _C_i) const;

  /**
   * check whether a given view (associated to table id _C_i) exists
   */
  bool has_view(const tableid& _C_i) const;
  bool has_view(const std::string& _C_i) const;
  bool has_view(const char* _C_i) const

  /**
   * this method returns a pointer to the view associated to _C_i.
   */
  viewbase* get_view(const tableid& _C_i) const;
  viewbase* get_view(const std::string& _C_i) const;
  viewbase* get_view(const char* _C_i) const;
```

A.2.7 Contexts

The context class is very flexible. The only methods defined handle the constant `table` pointer.

```
  /**
   * retrieve a pointer to the table this object belongs to
   */
  const table* table() const;

  /**
   * set the table pointer
   */
  void table(const _VDBL_table* t);
```

A.3 The API

This section contains the class definitions and the most important subset of the *public* methods therein.

A.3.1 Helper Classes

Basic Types

There is an all-in-one type `additional_info_u`, which is a union containing all elementary types allowed in intermodule communication.

```
class basic_alltype
{
public:
  basic_alltype();
  basic_alltype(bool __x);
  basic_alltype(int __x);
  basic_alltype(unsigned int __x);
  basic_alltype(double __x);
  basic_alltype(interval __x);
  basic_alltype(const char* __cp);
  basic_alltype(const std::string& __x);
  basic_alltype(const std::vector<bool>& __x);
  basic_alltype(const std::vector<int>& __x);
  basic_alltype(const std::vector<unsigned int>& __x);
  basic_alltype(const std::vector<double>& __x);
  basic_alltype(const std::vector<interval>& __x);
  basic_alltype(const matrix<double>& __x);
  basic_alltype(const matrix<int>& __x);
  basic_alltype(const matrix<interval>& __x);

  ~basic_alltype();

  basic_alltype(const basic_alltype& __a);

  basic_alltype& operator=(bool __x);
  basic_alltype& operator=(int __x);
  basic_alltype& operator=(unsigned int __x);
  basic_alltype& operator=(double __x);
  basic_alltype& operator=(interval __x);
  basic_alltype& operator=(const std::string& __x);
  basic_alltype& operator=(const char* __x);
  basic_alltype& operator=(const std::vector<bool>& __x);
  basic_alltype& operator=(const std::vector<int>& __x);
  basic_alltype& operator=(const std::vector<unsigned int>& __x);
  basic_alltype& operator=(const std::vector<double>& __x);
  basic_alltype& operator=(const std::vector<interval>& __x);
  basic_alltype& operator=(const matrix<double>& __x);
  basic_alltype& operator=(const matrix<int>& __x);
  basic_alltype& operator=(const matrix<interval>& __x);
  basic_alltype& operator=(const basic_alltype& __a);

  basic_alltype& clear();

  bool nb() const;
  int nn() const;
  unsigned int nu() const;
  double nd() const;
  interval ni() const;
  std::string& s() const;
  std::vector<bool>& b() const;
  std::vector<int>& n() const;
  std::vector<unsigned int>& u() const;
  std::vector<double>& d() const;
  std::vector<interval>& i() const;
  matrix<double>& m() const;
  matrix<int>& nm() const;
  matrix<interval>& im() const;

  bool is_allocated() const;
  bool is_vector() const;
  bool is_matrix() const;
  bool is_scalar() const;
  bool empty() const;
  int contents_type() const;
```

```
};
```

Datamap

```
/** @file datamap.h */

class datamap : std::map<std::string, basic_alltype>
{
public:
  datamap() : _Base(), __empty() {}
  datamap(const std::string& __n, const basic_alltype& __v)
    : _Base(), __empty()
    { insert(std::make_pair(__n,__v)); }
  datamap(const char* __n, const basic_alltype& __v)
    : _Base(), __empty()
    { insert(std::make_pair(std::string(__n),__v)); }
  datamap(const datamap& __c) : _Base(__c), __empty() {}
  virtual ~datamap() {}

  bool sinsert(const std::string& __s, const basic_alltype& __h,
               bool replace);
  bool sinsert(const char* __cp, const basic_alltype& __h, bool replace);

  bool sinsert(const std::string& __s, int i, const basic_alltype& __h,
               bool replace);
  bool sinsert(const char* __cp, int i, const basic_alltype& __h,
               bool replace);

  const basic_alltype& sfind(const std::string& __s) const;
  const basic_alltype& sfind(const char* __cp) const;

  const basic_alltype& sfind(const std::string& __s, int i) const;
  const basic_alltype& sfind(const char* __cp, int i) const;

  void remove(const std::string& __s);
  void remove(const char* __cp);

  void remove(const std::string& __s, int i);
  void remove(const char* __cp, int i);

  bool defd(const std::string& __s) const;
  bool defd(const char* __cp) const;

  bool defd(const std::string& __s, int i) const;
  bool defd(const char* __cp, int i) const;

  bool which(const std::string& __s, std::vector<int>& idx) const;
  bool which(const char* __cp, std::vector<int>& idx) const;
  bool retrieve(const std::string& __s, bool& __b) const;
  bool retrieve(const std::string& __s, bool& __b, bool __def) const;
  bool retrieve(const std::string& __s, int& __d) const;
  bool retrieve(const std::string& __s, int& __d, int __def) const;
  bool retrieve(const std::string& __s, unsigned int& __d) const;
  bool retrieve(const std::string& __s, unsigned int& __d,
      unsigned int __def) const;
  bool retrieve(const std::string& __s, double& __d) const;
  bool retrieve(const std::string& __s, double& __d, double __def) const;
  bool retrieve(const std::string& __s, interval& __b) const;
  bool retrieve(const std::string& __s, interval& __b,
      const interval& __def) const;
  bool retrieve(const std::string& __s, std::string& __is) const;
  bool retrieve(const std::string& __s, std::string& __is,
      const std::string& __def) const;
  bool retrieve(const std::string& __s, const std::vector<bool>*& __b) const;
  bool retrieve(const std::string& __s, const std::vector<bool>*& __b,
      const std::vector<bool>* __def) const;
  bool retrieve(const std::string& __s, const std::vector<int>*& __b) const;
  bool retrieve(const std::string& __s, const std::vector<int>*& __b,
      const std::vector<int>* __def) const;
  bool retrieve(const std::string& __s,
      const std::vector<unsigned int>*& __b) const;
  bool retrieve(const std::string& __s, const std::vector<unsigned int>*& __b,
      const std::vector<unsigned int>* __def) const;
  bool retrieve(const std::string& __s, const std::vector<double>*& __b) const;
  bool retrieve(const std::string& __s, const std::vector<double>*& __b,
      const std::vector<double>* __def) const;
  bool retrieve(const std::string& __s, const std::vector<interval>*& __b) const;
  bool retrieve(const std::string& __s, const std::vector<interval>*& __b,
      const std::vector<interval>* __def) const;
  bool retrieve(const std::string& __s, const matrix<double>*& __b) const;
  bool retrieve(const std::string& __s, const matrix<double>*& __b,
      const matrix<double>* __def) const;
  bool retrieve(const std::string& __s, const matrix<int>*& __b) const;
  bool retrieve(const std::string& __s, const matrix<int>*& __b,
      const matrix<int>* __def) const;
  bool retrieve(const std::string& __s, const matrix<interval>*& __b) const;
  bool retrieve(const std::string& __s, const matrix<interval>*& __b,
      const matrix<interval>* __def) const;

  bool retrieve(const char* __s, bool& __b) const;
  bool retrieve(const char* __s, bool& __b, bool __def) const;
  bool retrieve(const char* __s, int& __d) const;
  bool retrieve(const char* __s, int& __d, int __def) const;
  bool retrieve(const char* __s, unsigned int& __d) const;
  bool retrieve(const char* __s, unsigned int& __d, unsigned int __def) const;
  bool retrieve(const char* __s, double& __d) const;
  bool retrieve(const char* __s, double& __d, double __def) const;
  bool retrieve(const char* __s, interval& __b) const;
  bool retrieve(const char* __s, interval& __b,
      const interval& __def) const;
  bool retrieve(const char* __s, std::string& __is) const;
  bool retrieve(const char* __s, std::string& __b,
      const std::string& __def) const;
  bool retrieve(const char* __s, const std::vector<bool>*& __b) const;
  bool retrieve(const char* __s, const std::vector<bool>*& __b,
      const std::vector<bool>* __def) const;
  bool retrieve(const char* __s, const std::vector<unsigned int>*& __b) const;
  bool retrieve(const char* __s, const std::vector<unsigned int>*& __b,
      const std::vector<unsigned int>* __def) const;
  bool retrieve(const char* __s, const std::vector<int>*& __b) const;
  bool retrieve(const char* __s, const std::vector<int>*& __b,
      const std::vector<int>* __def) const;
  bool retrieve(const char* __s, const std::vector<double>*& __b) const;
  bool retrieve(const char* __s, const std::vector<double>*& __b,
      const std::vector<double>* __def) const;
  bool retrieve(const char* __s, const std::vector<interval>*& __b) const;
  bool retrieve(const char* __s, const std::vector<interval>*& __b,
      const std::vector<interval>* __def) const;
  bool retrieve(const char* __s, const matrix<double>*& __b) const;
  bool retrieve(const char* __s, const matrix<double>*& __b,
      const matrix<double>* __def) const;
  bool retrieve(const char* __s, const matrix<int>*& __b) const;
  bool retrieve(const char* __s, const matrix<int>*& __b,
      const matrix<int>* __def) const;
  bool retrieve(const char* __s, const matrix<interval>*& __b) const;
  bool retrieve(const char* __s, const matrix<interval>*& __b,
      const matrix<interval>* __def) const;

  bool retrieve_i(const std::string& __s, int i, bool& __b) const;
  bool retrieve_i(const std::string& __s, int i, bool& __b, bool __def) const;
  bool retrieve_i(const std::string& __s, int i, int& __d) const;
  bool retrieve_i(const std::string& __s, int i, int& __d, int __def) const;
  bool retrieve_i(const std::string& __s, int i, unsigned int& __d) const;
  bool retrieve_i(const std::string& __s, int i, unsigned int& __d,
      unsigned int __def) const;
  bool retrieve_i(const std::string& __s, int i, double& __d) const;
  bool retrieve_i(const std::string& __s, int i, double& __d, double __def) const;
  bool retrieve_i(const std::string& __s, int i, interval& __b) const;
  bool retrieve_i(const std::string& __s, int i, interval& __b,
      const interval& __def) const;
  bool retrieve_i(const std::string& __s, int i, std::string& __is) const;
  bool retrieve_i(const std::string& __s, int i, std::string& __is,
      const std::string& __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<bool>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const std::vector<bool>*& __b,
      const std::vector<bool>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<int>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const std::vector<int>*& __b,
      const std::vector<int>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<unsigned int>*& __b) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<unsigned int>*& __b,
      const std::vector<unsigned int>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<double>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const std::vector<double>*& __b,
      const std::vector<double>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<interval>*& __b) const;
  bool retrieve_i(const std::string& __s, int i,
      const std::vector<interval>*& __b,
      const std::vector<interval>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const matrix<double>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const matrix<double>*& __b,
      const matrix<double>* __def) const;
  bool retrieve_i(const std::string& __s, int i, const matrix<int>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const matrix<int>*& __b,
      const matrix<int>* __def) const;
  bool retrieve_i(const std::string& __s, int i,
      const matrix<interval>*& __b) const;
  bool retrieve_i(const std::string& __s, int i, const matrix<interval>*& __b,
      const matrix<interval>* __def) const;

  bool retrieve_i(const char* __s, int i, bool& __b) const;
  bool retrieve_i(const char* __s, int i, bool& __b, bool __def) const;
  bool retrieve_i(const char* __s, int i, int& __d) const;
  bool retrieve_i(const char* __s, int i, int& __d, int __def) const;
  bool retrieve_i(const char* __s, int i, unsigned int& __d) const;
  bool retrieve_i(const char* __s, int i, unsigned int& __d,
      unsigned int __def) const;
  bool retrieve_i(const char* __s, int i, double& __d) const;
  bool retrieve_i(const char* __s, int i, double& __d, double __def) const;
  bool retrieve_i(const char* __s, int i, interval& __b) const;
  bool retrieve_i(const char* __s, int i, interval& __b,
      const interval& __def) const;
  bool retrieve_i(const char* __s, int i, std::string& __is) const;
  bool retrieve_i(const char* __s, int i, std::string& __b,
      const std::string& __def) const;
  bool retrieve_i(const char* __s, int i, const std::vector<bool>*& __b) const;
  bool retrieve_i(const char* __s, int i, const std::vector<bool>*& __b,
      const std::vector<bool>* __def) const;
  bool retrieve_i(const char* __s, int i, const std::vector<int>*& __b) const;
  bool retrieve_i(const char* __s, int i, const std::vector<int>*& __b,
      const std::vector<int>* __def) const;
```

```
  bool retrieve_i(const char* __s, int i,
      const std::vector<unsigned int>*& __b) const;
  bool retrieve_i(const char* __s, int i, const std::vector<unsigned int>*& __b,
      const std::vector<unsigned int>* __def) const;
  bool retrieve_i(const char* __s, int i, const std::vector<double>*& __b) const;
  bool retrieve_i(const char* __s, int i, const std::vector<double>*& __b,
      const std::vector<double>* __def) const;
  bool retrieve_i(const char* __s, int i,
      const std::vector<interval>*& __b) const;
  bool retrieve_i(const char* __s, int i, const std::vector<interval>*& __b,
      const std::vector<interval>* __def) const;
  bool retrieve_i(const char* __s, int i, const matrix<double>*& __b) const;
  bool retrieve_i(const char* __s, int i, const matrix<double>*& __b,
      const matrix<double>* __def) const;
  bool retrieve_i(const char* __s, int i, const matrix<int>*& __b) const;
  bool retrieve_i(const char* __s, int i, const matrix<int>*& __b,
      const matrix<int>* __def) const;
  bool retrieve_i(const char* __s, int i, const matrix<interval>*& __b) const;
  bool retrieve_i(const char* __s, int i, const matrix<interval>*& __b,
      const matrix<interval>* __def) const;
};
```

Database Tools

This is the basic type used in all applications, which generate new entries.

```
typedef std::list<vdbl::col_spec> dbt_row;

template < class _C >
inline void add_to_dbt_row(dbt_row& dbr, const std::string& nm,
                           const _C& cont);

template < class _C >
inline void add_to_dbt_row(dbt_row& dbr, const char* nm,
                           const _C& cont);
```

Global Pointers

The class definition of the `gptr` templated class and its subclass `ptr`, which defines local global pointers.

```
template <class _Tp>
class gptr
{
public:
  virtual ~gptr();
  virtual reference operator*();
  virtual const_reference operator*() const;

  virtual pointer operator->();
  virtual const_pointer operator->() const;

  virtual pointer get_local_copy();
  virtual const_pointer get_local_copy() const;
};


template <class _Tp>
class ptr : public gptr<_Tp>
{
public:
  ptr(_Tp& __p);
  ptr(_Tp* __p);
  ptr(const _Self& __p);

  ~ptr();

  reference operator*();
  const_reference operator*() const;

  pointer operator->();
  const_pointer operator->() const;

  pointer get_local_copy();
  const_pointer get_local_copy();

  _Self& operator=(const _Self& __p);
  _Self& operator=(_Tp& __p);
};
```

A.3.2 Expressions

This section contains the building blocks of the mathematical functions, the basic expressions. They are essentially split in two classes, the `expression_node` class, which is the node type of the expression DAGs, and the `semantics` class, which contains additional information on the expressions important during the solution process. In principle, `expression_node` contains the static information on the mathematical functions, while `semantics` contains a lot of dynamic information, which can change during the solution process (e.g., information on convexity,. . .).

Semantics

```
typedef enum { c_convex=1, c_linear=0, c_concave=-1, c_maybe=2 } convex_info;
typedef enum { v_exists=0, v_forall=1, v_free=2, v_stochastic=3 }
                                                         type_annotation;

/**
 * @ingroup expression
 * This enum describes the activity state known about a constraint and
 * whether it is redundant or not. The meaning of the enum entries is:
 *      - @c a_redundant:  The constraint is known to be redundant.
 *                         There can be two reasons for that. It can
 *                         be inactive on both sides, or it was constructed,
 *                         knowing that it would be redundant (e.g. cuts).
 *      - @c a_active_lo:  The @c work_node may contain points, for which
 *                         this constraint is active at the lower bound but
 *                         no points, for which the constraint is active at
 *                         the upper bound.
 *      - @c a_active_hi:  The @c work_node may contain points, for which
 *                         this constraint is active at the upper bound but
 *                         no points, for which the constraint is active at
 *                         the lower bound.
 *      - @c a_active:     The @c work_node may contain points, for which
 *                         this constraint is active at the upper bound and
 *                         points, for which the constraint is active at
 *                         the lower bound.
 *      - @c ..._red:      A combination with @c _red means that although
 *                         the inactivity of the constraint could not be
 *                         proved, it is still known to be redundant.
 */

typedef enum { a_redundant=1,
               a_active_lo=2, a_active_lo_red=a_active_lo|a_redundant,
               a_active_hi=4, a_active_hi_red=a_active_hi|a_redundant,
               a_active=a_active_lo|a_active_hi,
               a_active_red=a_active|a_redundant }      activity_descr;

class convex_e
{
public:
  convex_e();
  convex_e(const convex_info& __e, uint16_t __t);
  convex_e(const convex_e& __e);
  ~convex_e() {}

  const convex_info& i() const;
  uint16_t t() const;

  convex_e operator-() const;
  convex_e& operator=(const convex_e& __c);
  convex_e& operator=(const convex_info& __i);
  convex_e& operator=(unsigned int __t);
  convex_e& operator=(uint16_t __t);
  friend std::ostream& operator<< (std::ostream& o, const convex_e& __s);
};

class semantics
{
public:
  struct {
    convex_e c_info;
    activity_descr act;                 // activity descriptor
    tristate separable;
    bool is_at_either_bound;
  } property_flags;                     // mathematical constraint properties

  struct {
    bool kj;                            // is from KJ conditions
    bool integer;                       // is integer / not real
    type_annotation type;               // exists, forall, free, stochastic
    bool hard;                          // hard or soft constraint
  } annotation_flags;

  struct {
    tristate has_0chnbase;
  } info_flags;                         // algorithmic properties

  unsigned int _0chnbase;

  int addinfo;                          // for KJ variables this number
                                        // is the corresponding constraint
                                        // number, -1 if kappa (for obj.)

  int degree;                           // -1: essential non-linearity
  int dim;                              // num of vars involved
  int stage;                            // for multistage problems
public:
  semantics();
  semantics(const semantics& __s);
  ~semantics() {}

  friend std::ostream& operator<< (std::ostream& o, const semantics& __s);

  // access and storage methods for flags which are
  // not automatically set by the simplifier
  const convex_e& convexity() const;
  void convexity(const convex_e& __c);

  const tristate& separability() const;
  void separability(const tristate& __c);

  const activity_descr& activity() const;
  void activity(const activity_descr& __c);

  bool kj_flag() const;
  void kj_flag(bool __c);

  bool integer_flag() const;
  void integer_flag(bool __c);

  bool hard_flag() const;
  void hard_flag(bool __c);

  bool is_at_any_bound() const;
  void is_at_any_bound(bool __c);

  const type_annotation& type();
  void type(const type_annotation& __c);

  bool redundancy() const;
  bool inactive_hi() const;
  bool inactive_lo() const;
  bool inactive() const;

  friend std::ostream& operator<< (std::ostream& o, const semantics& __s);
};
```

Expression Nodes

```
enum { // type flags
       ex_bound=1, ex_linear=1<<1, ex_quadratic=1<<2, ex_polynomial=1<<3,
       ex_other=1<<4,

       // automatic from Karush-John conditions (for vars. and constr.)
       // ex_kj .... only additional vars, constr. from KJ-conditions
       // ex_org ... original vars, constr. (both set = none set)
       ex_kj=1<<7, ex_org=1<<8,

       // redundant (both set = none set)
       ex_redundant=1<<9, ex_notredundant=1<<10,

       // activity
       // both set = none set = all constraints
       ex_active_lo=1<<11, ex_inactive_lo=1<<12,
       ex_active_hi=1<<13, ex_inactive_hi=1<<14,
       ex_active=ex_active_lo|ex_active_hi,
       ex_inactive=ex_inactive_lo|ex_inactive_hi,

       // integer
       ex_integer=1<<15,
       // forall, exists, stochastic, free
       // none set = select all
       ex_exists = 1<<16,
       ex_forall = 1<<17,
       ex_free = 1<<18,
       ex_stochastic = 1<<19,

       // convexity flags (known to be!)
       ex_convex=1<<20, ex_concave=1<<21,

       // bound flags
       ex_inequality=1<<28, ex_equality=1<<29,
       ex_leftbound=1<<30, ex_rightbound=1<<31,

       // short cuts
       ex_atmlin=ex_bound|ex_linear,                    // at most linear
       ex_atmquad=ex_atmlin|ex_quadratic,               // at most quadratic
       ex_atmpoly=ex_atmquad|ex_polynomial,             // at most polynomial
       ex_nonlin=ex_quadratic|ex_polynomial|ex_other,   // non linear
       ex_nonbnd=ex_linear|ex_nonlin,                   // not bounds
       ex_any=ex_atmlin|ex_nonlin,                      // any
       ex_bothbound=ex_leftbound|ex_rightbound};

typedef interval rhs_t; // use intervals as right hand sides of constraints.
                        // more general sets could be introduced later
typedef std::vector<void*> evaluator_v;

#include <addinfo.h>

class expression_node
{
public:
  unsigned int node_num;  // node number for indexing

  int operator_type;      // this is a number describing the operation.
                          // negative numbers describe standard operations
                          // as defined above.
                          // the positive numbers describe other operations
                          // like elementary functions (exp, pow, sin,...)
                          // or more complicated functions like linear or
                          // general quadratic terms, user-defined functions,
                          // piecewise linear approximations,...

  unsigned int n_parents, n_children;
                          // number of parents, children

  std::vector<double> coeffs;  // coefficients of the sub_expressions

  basic_alltype params; // additional expression info
  rhs_t f_bounds;      // bounds on this node

  unsigned short is_var;  // this node represents is_var variables
  std::vector<unsigned int> var_idx;  // the variable indices corresponding to this node

  semantics sem;          // additional semantics information like
                          // integer (y/n), stochastic (y/n), karush-john (y/n)
                          // convexity,...

  variable_indicator v_i; // this node depends on which variables?

  evaluator_v* ev;        // optional evaluators used instead of tree walks
                          // NULL means no evaluators, map if some or all
                          // are defined. This is for recursive_cached_walk

  // constructors and destructor
  expression_node();
  expression_node(int et, int nn);
  expression_node(const expression_node& __x);
  ~expression_node();

  bool operator<(const expression_node& __x) const;

  void merge(const expression_node& __s);

  void set_bounds(interval __i);
  void set_bounds(double __d = 0.);
  void set_bounds(int __i);
  void set_bounds(double lo, double up);

  void add_is_var(unsigned int idx);
  void rm_is_var(unsigned int idx);

  bool is(unsigned int __tp) const;

  // and others needed
  // print the expression_node
  friend std::ostream& operator<< (std::ostream& o, const expression_node& __x);

  // and others needed
  const variable_indicator& var_indicator() const;
};
```

A.3.3 Models

This section contains the C++ description of the three model base classes, as explained in Section 5.3.3. The class definitions are reduced such that they contain only the most important methods.

Id Data

```
class model_iddata
{
public:
  model_iddata(unsigned int n = 0);
  ~model_iddata();

  void new_ref(model_gid& __m);
  bool delete_ref(model_gid& __m);

  unsigned int get_node_id();
  void remove_node_id(unsigned int n);
  unsigned int get_var_id();
  void remove_var_id(unsigned int n);
  unsigned int get_const_id();
  void remove_const_id(unsigned int n);
  void compress_numbers(bool renum_vars, bool renum_consts = false);
};
```

Model Group Data

```
class model_gid
{
public:
  void remove_node_ref(unsigned int _n);
  void remove_var_ref(unsigned int _n);
  void remove_const_ref(unsigned int _n);

  model_gid(model& __m, model_iddata* __i=NULL);
  model_gid(model& __m, unsigned int n, model_iddata* __i = NULL);
  model_gid(model& __mr, const model_gid& __m);
  ~model_gid();

  void mk_globref(unsigned int n, const model::walker& __w);
  void mk_gvarref(unsigned int n, const model::walker& __w);
  void mk_gconstref(unsigned int n, const model::walker& __w);

  void make_const_back_ref(unsigned int node, unsigned int cnum);

  bool empty(const model::walker& __x) const;
  bool its_me(const model& __m) const;
  model::walker empty_reference() const;

  bool have_glob_ref(unsigned int _nnum) const;
  bool have_gvar_ref(unsigned int _vnum) const;
  bool have_gconst_ref(unsigned int _cnum) const;
};
```

Models

```
class model: public dag<expression_node>
{
public:
  matrix<double> lin;           // matrix of linear constraints
  int ocoeff;                   // max (-1) or min (1) or nothing (0)?
  walker objective;             // pointer to the objective
  std::vector<walker> constraints;   // pointer to the constraints

public:
  model(model_gid* id = NULL, bool clone = false);
  model(model_gid* id, const erased_part& ep, bool clone = false);
  model(int num_of_vars);
  model(const model& m);
  model(model_gid* id, const model& m);
  model(istream& inp, bool do_simplify = true);
  model(const char* name, bool do_simplify = true);

  ~model();

  unsigned int number_of_variables() const;
  unsigned int number_of_nodes() const;
  unsigned int number_of_constraints() const;
  unsigned int number_of_managed_nodes() const;
  unsigned int number_of_managed_variables() const;
  unsigned int number_of_managed_constraints() const;

  const walker& var(unsigned int i) const;
  const walker& node(unsigned int i) const;
  const walker& constraint(unsigned int i) const;

  model_gid* gid_data() const;
  void detach_gid();

  void compress_numbers();
  void renumber_variables();
  void renumber_constraints();

  bool basic_simplify();
  void arrange_constraints();
  void set_counters();
  void clr_sky_ground_link();

  void write(std::ostream& __o = std::cout) const;
```

```
ref_iterator ghost_begin();
const_ref_iterator ghost_begin() const;
ref_iterator ghost_end();
const_ref_iterator ghost_end() const;

walker store_node(const walker& _w);
walker store_variable(const walker& _w);
walker store_ghost(const walker& _w);
walker store_constraint(const walker& _w);

void free_node_num(unsigned int _nnum);

void remove_node(const walker& _w, unsigned int _nnum);
void remove_node(const walker& _w);
void remove_node(unsigned int __node_num);

void new_variables(int _new_num_of_vars);

walker ghost(unsigned int _nnum);

walker constant(double _constant);
walker constant(const std::vector<double>& _constant);

walker variable(unsigned int _vnum);

walker binary(const walker& _op1, const walker& _op2, int expr_type,
              double _coeff1 = 1.0, double _coeff2 = 1.0);
walker binary(const walker& _op1, const walker& _op2, int expr_type,
              basic_alltype _params, double _coeff1 = 1.0,
              double _coeff2 = 1.0);

walker unary(const walker& _op1, int expr_type, double _coeff = 1.0);
walker unary(const walker& _op1, int expr_type, basic_alltype _params,
    double _coeff = 1.0);

walker nary(const std::vector<walker>& _op, int expr_type,
            const std::vector<double>& _coeffs = std::vector<double>());
walker nary(const std::vector<walker>& _op, int expr_type,
            basic_alltype _params,
            const std::vector<double>& _coeffs = std::vector<double>());

walker vnary(int expr_type, ...);

bool is_empty(const walker& _w) const;
walker empty_reference() const;

const std::string var_name(unsigned int n) const;
const std::string const_name(unsigned int n) const;
const std::string obj_name() const;
double obj_adj() const;
```

```
  double obj_mult() const;
  size_t n_fixed_vars() const;
  std::pair<const std::string, double> fixed_var(unsigned int n) const;
  size_t n_unused_vars() const;
  const std::string& unused_var(unsigned int n) const;
  size_t n_unused_constrs() const;
  const std::string& unused_constr(unsigned int n) const;

  bool get_const_num(unsigned int node_num, unsigned int& const_num) const;

  bool get_linear_coeffs(const walker& expr, sparse_vector<double>& coeffs,
      double& constant, const std::vector<interval>& _ranges);
  bool get_linear_coeffs(const walker& expr, sparse_vector<double>& coeffs,
      double& constant);
};

typedef model::walker expression_walker;
typedef model::const_walker expression_const_walker;
```

A.3.4 Control Data

This class is used to pass control parameters to all modules.

```
/** @file control_data.h */

class control_data : protected datamap
{
public:
  control_data();
  control_data(const std::string& __n, const basic_alltype& __v);
  control_data(const char* __n, const basic_alltype& __v);
  control_data(const control_data& __c);
  control_data(const std::string& serv, const info_contents& inf);
  virtual ~control_data();

  control_data& operator=(const info_contents& __i);

  void service(const std::string& __s);
  void service(const char* __s);
  const std::string& service() const;

  bool check_service(const std::string& __n) const;
  bool check_service(const char* __n) const;

  void set(const info_contents& __i);
  void set(const std::string& __s, const basic_alltype& __h);
  void set(const char* __cp, const basic_alltype& __h);
  void set(const std::string& __s, int i, const basic_alltype& __h);
  void set(const char* __cp, int i, const basic_alltype& __h);

  const basic_alltype& get(const std::string& __s) const;
  const basic_alltype& get(const char* __cp) const;
  const basic_alltype& get(const std::string& __s, int i) const;
  const basic_alltype& get(const char* __cp, int i) const;

  void unset(const std::string& __s);
  void unset(const char* __cp);
  void unset(const std::string& __s, int i);
  void unset(const char* __cp, int i);

  bool is_set(const std::string& __s) const;
  bool is_set(const char* __cp) const;
  bool is_set(const std::string& __s, int i) const;
  bool is_set(const char* __cp, int i) const;

  bool which_set(const std::string& __s, std::vector<int>& __idx) const;
  bool which_set(const char* __cp, std::vector<int>& __idx) const;

  template <class _S>
  void assign(const std::string& __s, const std::vector<_S>*& __b) const;

  template <class _S>
  void assign(const std::string& __s, const matrix<_S>*& __b) const;

  template <class _S>
  void assign(const std::string& __s, _S& __b) const;

  template <class _S>
  void assign(const std::string& __s, const std::vector<_S>*& __b,
              const std::vector<_S>* __def) const;

  template <class _S>
  void assign(const std::string& __s, const matrix<_S>*& __b,
              const matrix<_S>* __def) const;

  template <class _S>
  void assign(const std::string& __s, _S& __b, _S __def) const;

  template <class _S>
  void assign(const char* __s, const std::vector<_S>*& __b) const;

  template <class _S>
  void assign(const char* __s, const matrix<_S>*& __b) const;

  template <class _S>
  void assign(const char* __s, _S& __b) const;

  template <class _S>
```

```
  void assign(const char* __s, const std::vector<_S>*& __b,
              const std::vector<_S>* __def) const;

  template <class _S>
  void assign(const char* __s, const matrix<_S>*& __b,
              const matrix<_S>* __def) const;

  template <class _S>
  void assign(const char* __s, _S& __b, _S __def) const;

  template <class _S>
  void assign_i(const std::string& __s, int i,
                const std::vector<_S>*& __b) const;

  template <class _S>
  void assign_i(const std::string& __s, int i, const matrix<_S>*& __b) const;

  template <class _S>
  void assign_i(const std::string& __s, int i, _S& __b) const;

  template <class _S>
  void assign_i(const std::string& __s, int i, const std::vector<_S>*& __b,
                const std::vector<_S>* __def) const;

  template <class _S>
  void assign_i(const std::string& __s, int i, const matrix<_S>*& __b,
                const matrix<_S>* __def) const;

  template <class _S>
  void assign_i(const std::string& __s, int i, _S& __b, _S __def) const;

  template <class _S>
  void assign_i(const char* __s, int i, const std::vector<_S>*& __b) const;

  template <class _S>
  void assign_i(const char* __s, int i, const matrix<_S>*& __b) const;

  template <class _S>
  void assign_i(const char* __s, int i, _S& __b) const;

  template <class _S>
  void assign_i(const char* __s, int i, const std::vector<_S>*& __b,
                const std::vector<_S>* __def) const;

  template <class _S>
  void assign_i(const char* __s, int i, const matrix<_S>*& __b,
                const matrix<_S>* __def) const;

  template <class _S>
  void assign_i(const char* __s, int i, _S& __b, _S __def) const;
```

```
};
```

A.4 Search Graph

In this section, the class definitions of all parts of the API are given, which are related to the search graph. The search graph consists of search nodes, which come in two flavours, full_node and delta_node.

A.4.1 Search Nodes

```
class search_node
{
protected:
  gptr<search_node>* __global_model;
  gptr<vdbl::database>* __dbase;
  vdbl::userid _dbuser;
  search_node_relation _snr;
  search_node_id _id;
  std::vector<annotation> _keep;

public:
  virtual bool is_delta() const;

  search_node(const search_node_id& _i, const vdbl::userid& _dui,
              gptr<search_node>& _gm, gptr<vdbl::database>& _db,
              search_node_relation __snr = snr_reduction);
  search_node(const search_node_id& _i, const vdbl::userid& _dui,
              gptr<search_node>* _gm, gptr<vdbl::database>& _db,
              search_node_relation __snr = snr_reduction);
  search_node(const search_node_id& _i, const vdbl::userid& _dui,
              gptr<vdbl::database>& _db, search_node_relation __snr = snr_root);

  virtual ~search_node();

  vdbl::userid get_dbuserid() const;

  gptr<search_node>* global_model();

  gptr<vdbl::database>* database();

  search_node_id get_id() const;

  void keep(const annotation& _an);
  void keep(const std::vector<annotation>& _anv);

  void unkeep(const annotation& _an);
  void unkeep(const std::vector<annotation>& _anv);
};



class delta_node : public search_node
{
public:
  delta_node(const search_node_id& _i, const vdbl::userid& _dui,
             std::vector<delta_id>& __d, gptr<search_node>& _gm,
             gptr<vdbl::database>& _db,
             search_node_relation _snr = snr_reduction);
  virtual ~delta_node();

  unsigned int n_deltas() const;

  delta_id get_delta_id(unsigned int i) const;
  delta get_delta(unsigned int i);
  const delta& get_delta(unsigned int i) const;
};



class full_node : public search_node
{
protected:
  gptr<model>*          _m;

public:
  std::vector<annotation> _ann;

  full_node(const search_node_id& _i, const vdbl::userid& _dui,
            gptr<model>& __mod, gptr<search_node>& _gm,
            gptr<vdbl::database>& _db,
            search_node_relation _snr = snr_reduction);
  full_node(const search_node_id& _i, const vdbl::userid& _dui,
            gptr<model>& __mod, gptr<search_node>& _gm,
            gptr<vdbl::database>& _db, const std::vector<annotation>& _a,
            search_node_relation _snr = snr_reduction);
  virtual ~full_node();

  bool is_delta() const;

  const annotation& get_annotation(unsigned int i) const;
  const std::vector<annotation>& get_annotations() const;

  const model* get_model() const;
  const vdbl::database* get_database() const;

  model* get_model_ptr() const;
  vdbl::database* get_database_ptr() const;
};
```

A.4.2 Annotations

This section is about annotations, table entries in the search database, which belong to the current worknode.

```
class annotation : public std::pair<vdbl::tableid,vdbl::rowid>
{
public:
  annotation(const vdbl::tableid& _ti, const vdbl::rowid& _ri);
  virtual ~annotation();

  vdbl::tableid get_table() const;
  vdbl::rowid get_entry() const;
};
```

A.4.3 Work Nodes

```
class work_node : public full_node
{
public:
  std::list<delta_id> deltas;
  std::map<delta_id,undelta> undeltas;
  vdbl::standard_table *dtable;
  vdbl::tableid dtable_id;

  std::vector<interval> node_ranges;
  bool infeasible;

  double log_vol;
  double gain_factor;

  std::map<transaction_number,std::list<std::vector<delta> > > proposed_splits;

  work_node(const search_node_id& _i, const vdbl::userid& _dui,
            gptr<model>& __m, gptr<vdbl::database>& __d,
            const std::vector<annotation>& __an,
            const std::list<delta_id>& __de, gptr<search_node>* _gm);

  virtual ~work_node();

  transaction_number get_transaction_number();

  void init_cnumbers();
  void reset_node_ranges(); // reset all unused node ranges to [-I,I]
  void make_node_ranges(bool keep_old_ranges);
  double compute_log_volume(const std::vector<interval>& _r) const;

  const model* get_model() const;
  model* get_model();

  // usage e.g.:
  //     constraint_iterator b = get_begin(ex_linear);
  //     while(b != get_end(ex_linear))
  //     {
  //         do_something(*b);
  //         ++b;
  //     }
  // to iterate through some constraints
  constraint_const_iterator get_begin(unsigned int __type) const;
  constraint_const_iterator get_end(unsigned int __type) const;

  constraint_iterator get_begin(unsigned int __type);
  constraint_iterator get_end(unsigned int __type);

  delta get_delta(const delta_id& _id);
  const delta& get_delta(const delta_id& _id) const;

  double log_volume() const;
  double gain() const;
  double reset_gain();

  // usage:   n(ex_linear|ex_equality)  returns the number of linear
  //                                    equality constraints
  unsigned int n(unsigned int __type) const;

  friend work_node operator+(const work_node& _w, const delta_id& _d);
  friend work_node operator-(const work_node& _w, const delta_id& _d);
  friend work_node& operator+=(work_node& _w, const delta_id& _d);
  friend work_node& operator-=(work_node& _w, const delta_id& _d);

  template <template <class _Tp, class _A> class _Ctr, class _Al>
  friend work_node operator+(const work_node& _w, const _Ctr<delta_id,_Al>& _d);

  template <template <class _Tp, class _A> class _Ctr, class _Al>
  friend work_node operator-(const work_node& _w, const _Ctr<delta_id,_Al>& _d);

  template <template <class _Tp, class _A> class _Ctr, class _Al>
  friend work_node& operator+=(work_node& _w, const _Ctr<delta_id,_Al>& _d);

  template <template <class _Tp, class _A> class _Ctr, class _Al>
  friend work_node& operator-=(work_node& _w, const _Ctr<delta_id,_Al>& _d);
};
```

A.4.4 Search Graph

```
/** @file search_graph.h */
```

```
class search_graph : public dag<search_node*>
{
public:
  search_node *root;
  search_inspector inspector_for_root;
  gptr<vdbl::database> *__dbase;
  vdbl::userid __dbuser;


  search_graph(model& root_model, gptr<vdbl::database>& _db);
  search_graph(model & root_model, gptr<vdbl::database>& _db,
               const std::vector<annotation>& _a);
  ~search_graph();

  search_focus& new_focus(const search_inspector& _si);
  void destroy_focus(const search_focus& _sf);
  search_focus& set_focus(search_focus& _fc, const search_inspector& _si);

  search_inspector& new_inspector(const search_inspector& inspector_to_add);
  search_inspector& new_inspector();
  void destroy_inspector(const search_inspector& inspector_to_destroy);

  work_node extract(const search_inspector& where);
  work_node extract(const search_focus& where);

  search_inspector& insert(const search_focus& where, const search_node& what);

  search_inspector& replace(search_focus& where, const search_node& what);

  void remove(search_focus& _sf);

  search_focus& promote(search_focus& _sf);

  search_inspector child(search_inspector& n, unsigned int i);
  search_inspector parent(search_inspector& n, unsigned int i);
};

typedef search_graph::const_walker search_inspector;
typedef search_graph::walker search_focus;
```

A.4.5 Deltas

This section contains the C++ description of all types of deltas available in the environment, as well as the definitions and most important *public* methods of the base classes.

A.4.6 Base Classes

```
class delta
{
public:
  delta();
  delta(const delta_base& __d);
  delta(const delta& __d);

  ˜delta();

  const std::string& get_action() const;
  const delta_base* get_base() const;

  bool apply(work_node& _x, const delta_id& _d) const;
  bool apply3(work_node& _x, const work_node& _y, const delta_id& _d) const;
  void convert(work_node& _x);
  delta_id store(work_node& _x);

  delta& operator=(const delta& _d);

  friend std::ostream& operator<< (std::ostream& o, const delta& t);
};

class delta_base
{
public:
  delta_base(const std::string& a);
  delta_base(const char* a);
  delta_base(const delta_base& __d);

  virtual delta_base* new_copy() const;
  virtual void destroy_copy(delta_base* __d);

  virtual ˜delta_base() {}

  delta make_delta(const std::string& a);
  const std::string& get_action() const;

  virtual void convert(work_node& _x, delta_base*& _d);
  virtual bool apply(work_node& _x, undelta_base*& _u) const;
  virtual bool apply3(work_node& _x, const work_node& _y, undelta_base*& _u);
};

class undelta
{
public:
  undelta();
  undelta(undelta_base* __d);
  undelta(const undelta& __d);

  ~undelta();

  const undelta_base* get_base() const;

  bool unapply(work_node& _x) const;
  bool unapply3(work_node& _x, const work_node& _y) const;

  undelta& operator=(const undelta& _u);
};


class undelta_base
{
public:
  undelta_base();
  undelta_base(const undelta_base& __d);

  virtual undelta_base* new_copy() const;
  virtual void destroy_copy(undelta_base* __d);

  virtual ~undelta_base();

  undelta make_undelta();

  virtual bool unapply(work_node& _x) const;
  virtual bool unapply3(work_node& _x, const work_node& _y) const;
};
```

A.4.7 Infeasible Delta

```
/** @file infeasible_delta.h */
class infeasible_delta : public delta_base
{
public:
  infeasible_delta() : delta_base("infeasible") {}
  ~infeasible_delta() {}
};
```

A.4.8 DAG Delta

```
/** @file dag_delta.h */
class dag_delta : public delta_base
{
public:
  counted_ptr<model> new_constraints;   // dags referring to ghost nodes
  std::vector<walker> rm_nodes;         // nodes that need to be removed
                                        // in the model

  bool is_full_delta;                   // this means that new_constraints
                                        // is a full model which completely
                                        // replaces old_constraints

  dag_delta(const std::string& __a, bool full=false);
  dag_delta(const std::string& __a, model* __nc, bool full=false);
  ~dag_delta() {}

  void add_new(model* __m);
  void add_new(model& __m);

  void remove(const walker& _nn);
  void remove(const std::vector<walker>& _nn);
};
```

A.4.9 Bound Delta

```
/** @file bound_delta.h */

class bound_delta : public delta_base
{
public:
  std::vector<unsigned int> indices;         // if all variables are being
                                             // updated this vector is empty
  std::vector<interval> new_f_bounds;        // new bounds

  bound_delta(const std::vector<unsigned int>& __i,
              const std::vector<interval>& __b);

  bound_delta(unsigned int __i, interval __b);
};
```

A.4.10 Semantics Delta

```
/** @file semantics_delta.h */

class semantics_delta : public delta_base
{
public:
  semantics_delta(const std::vector<unsigned int>& __i,
                  const std::vector<uint32_t>& __b);
  semantics_delta(unsigned int __i, uint32_t __b);
  semantics_delta();
  void set(const std::vector<unsigned int>& _i,
           const std::vector<semantics>& _s);
  void set(const std::vector<unsigned int>& _i,
           const std::vector<uint32_t>& _s);
  void set(unsigned int _i, const semantics& _s);
  void set_convex(const std::vector<unsigned int>& _i,
                  const std::vector<convex_e>& c);
  void set_convex(unsigned int _i, const convex_e& c);
  void set_activity(const std::vector<unsigned int>& _i,
                  const std::vector<activity_descr>& a);
  void set_activity(unsigned int _i, const activity_descr& a);
  void set_separable(const std::vector<unsigned int>& _i,
                     const std::vector<tristate>& t);
  void set_separable(unsigned int _i, const tristate& t);
  void set_is_at_either_bound(const std::vector<unsigned int>& _i,
                         const std::vector<bool>& b);
  void set_is_at_either_bound(unsigned int _i, bool b);
  void set_integer(const std::vector<unsigned int>& _i,
                   const std::vector<bool>& b);
  void set_integer(unsigned int _i, bool b);
  void set_hard(const std::vector<unsigned int>& _i,
                const std::vector<bool>& b);
  void set_hard(unsigned int _i, bool b);
  void set_type(const std::vector<unsigned int>& _i,
                const std::vector<type_annotation>& a);
  void set_type(unsigned int _i, const type_annotation& a);
};
```

A.4.11 Split Delta

```
/** @file split_delta.h */

class split_delta : public delta_base
{
public:
  std::list<std::vector<delta> > splits;
  // splits represents a list of newly created submodels. Each of these
  // submodels is generated from the work node by a number of deltas stored
  // in the inner vector.

  split_delta();

  split_delta(unsigned int _node_num, const interval& _l, const interval& _u);
  split_delta(unsigned int _node_num, const std::vector<interval>& _m);
  split_delta(const std::vector<unsigned int>& _i,
            const std::vector<interval>& _l, const std::vector<interval>& _u);
  split_delta(const std::list<std::vector<delta> >& __dl);
  ~split_delta() {}
  void add_delta(const delta& __d);
  void add_deltas(const std::vector<delta>& __d);

  void add_split(unsigned int _nnum, interval _l, interval _u);
  void add_split(const std::vector<unsigned int>& _i,
                 const std::vector<interval>& _l,
                 const std::vector<interval>& _u);
};
```

A.4.12 Annotation Delta

```
/** @file annotation_delta.h */

class annotation_delta : public delta_base
{
public:
  std::vector<annotation> add;          // these are to be added
  std::vector<annotation> rm;           // these are to be removed

  annotation_delta(const std::string& _act);
  annotation_delta(const std::string& _act,
                   const std::vector<annotation>& __a,
                   const std::vector<annotation>& __r);
  annotation_delta(const std::string& _act, const annotation& _ad);
  annotation_delta(const std::string& _act, bool _dummy,
                   const annotation& _rm);
  annotation_delta(const annotation_delta& __d);
  annotation_delta(const char* _act,
                   const std::vector<annotation>& __a,
                   const std::vector<annotation>& __r);
  annotation_delta(const char* _act, const annotation& _ad);
  annotation_delta(const char* _act, bool _dummy, const annotation& _rm);
};
```

A.4.13 Table Delta

```
/** @file table_delta.h */

class table_delta : public delta_base
{
public:
  typedef std::pair<std::string,dbt_row> t_line;
  typedef std::vector<t_line> t_ctr;

  table_delta(const std::string& a);
  table_delta(const std::string& a, const t_ctr& _n,
              const std::vector<annotation>& _r);

  table_delta(const std::string& a, const std::string& __tn,
              const dbt_row& __t);
  table_delta(const std::string& a, const std::string& __tn,
              const std::vector<dbt_row>& __t);

  void add(const t_line& _tl);
  void add(const std::string& _tn, const dbt_row& _r);
  void add(const std::vector<t_line>& _tlv);

  void rm(const annotation& _tr);
  void rm(const std::vector<annotation>& _trv);

  virtual void create_table(work_node& _x, vdbl::standard_table*& ptb,
                            const std::string& __t);
};
```

A.4.14 Box Delta

```
/** @file boxes_delta.h */

class boxes_delta : public table_delta
{
public:
  boxes_delta(bool _add=true);
  boxes_delta(const dbt_row& _b, bool _add=true);
  boxes_delta(const std::vector<dbt_row>& _b, bool _add=true);
};
```

A.4.15 Point Delta

```
/** @file point_delta.h */

class point_delta : public table_delta
{
public:
  point_delta();
  point_delta(const dbt_row& __p);
};
```

A.5 Evaluators

This section contains C++ examples for using the various evaluator variants.

A.5.2 Function Evaluation

In this code snipplet the objective function is evaluated.

```
#include <model.h>
#include <func_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  func_eval fv(x, v_ind, model, NULL);
  double fx = evaluate(fv, DAG.objective);
  std::cout << "The function value is: " << fx << std::endl;
  return 0;
}
```

A.5.3 Gradient Evaluation

The following piece of code evaluated the gradient of the objective function.

```
#include <model.h>
#include <der_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> x(n), grad(n), zero(n, 0.);
  std::vector<std::vector<double> > d_data;
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  //prepare data structure for derivative
  prep_d_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_d_eval fv(x, v_ind, model, d_data, NULL);
  double fx = evaluate(fv, DAG.objective);

  der_eval dv(d_data, v_ind, DAG, NULL, grad);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function value is: " << fx << std::endl;
  std::cout << "The gradient is: ";
  std::copy(grad.begin(), grad.end(),
            std::ostream_iterator<double>(std::cout, ", "));
  return 0;
}
```

A.5.4 Range Evaluation (Interval Evaluation)

In this code snipplet the range of the objective function is enclosed.

```
#include <model.h>
#include <int_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  interval_eval fv(x, v_ind, model, NULL);
  interval fx = evaluate(fv, DAG.objective);
  std::cout << "The function range is in: " << fx << std::endl;
  return 0;
}
```

A.5.5 Interval Derivative Evaluation

The following code piece evaluates the range of the gradient of the objective function.

```
#include <model.h>
#include <ider_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<interval> x(n), grad(n), zero(n, 0.);
  std::vector<interval> rg(DAG.number_of_nodes());
  std::vector<std::vector<interval> > d_data;
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  /**
   * constraint propagation happens here,
   * storing the node ranges in rg.
   */

  //prepare data structure for derivative
  prep_id_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_id_eval fv(x, rg, v_ind, model, d_data, NULL);
  interval fx = evaluate(fv, DAG.objective);

  der_eval dv(x, d_data, v_ind, DAG, NULL, grad);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function range is: " << fx << std::endl;
  std::cout << "The interval derivative is: ";
  std::copy(grad.begin(), grad.end(),
            std::ostream_iterator<interval>(std::cout, ", "));
  return 0;
}
```

A.5.6 First order Slope Evaluation

The following code piece evaluates the slope of the objective function.

```
#include <model.h>
#include <islp_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> z(n), f(DAG.number_of_nodes());
  std::vector<interval> x(n), slope(n), zero(n, 0.);
  std::vector<interval> rg(DAG.number_of_nodes());
  std::vector<std::vector<interval> > slp_data;
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  /**
   * constraint propagation happens here,
   * storing the node ranges in rg.
   */

  //prepare data structure for derivative
  prep_islp_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_islp_eval fv(z, rg, v_ind, model, d_data, f);
  interval fx = evaluate(fv, DAG.objective);

  der_eval dv(x, d_data, v_ind, DAG, NULL, slope);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function range is: " << fx.rg << std::endl;
  std::cout << "The enclosure of the center evaluation is: " <<
        fx.fi << std::endl;
  std::cout << "The approximation of the center evaluation is: "
        << fx.f << std::endl;
  std::cout << "The slope is: ";
  std::copy(slope.begin(), slope.end(),
            std::ostream_iterator<interval>(std::cout, ", "));
  return 0;
}
```

A.5.7 Analytic-Differentiable Evaluation

The following code snipplet evaluates the objective function.

```
#include <model.h>
#include <ade_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<analyticd> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  /**
   *  Here initialize x
   */

  analyticd_eval fv(x, v_ind, model, NULL);
  analyticd fx = evaluate(fv, DAG.objective);
  return 0;
}
```

A.5.8 Bounded Interval Evaluation

This code snipplet encloses the objective function:

```
#include <model.h>
#include <bint_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<b_interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  b_interval_eval fv(x, v_ind, model, NULL);
  b_interval fx = evaluate(fv, DAG.objective);
  // produce output
  return 0;
}
```

A.5.9 Complex Interval Evaluation

In this code snipplet the objective function is evaluated.

```
#include <model.h>
#include <cint_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<c_interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  c_interval_eval fv(x, v_ind, model, NULL);
  c_interval fx = evaluate(fv, DAG.objective);
  // produce output
  return 0;
}
```

A.5.10 Infinity-Bound Evaluation

In this code the objective function is enclosed.

```
#include <model.h>
#include <infb_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<infbound> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  infbound_eval fv(x, v_ind, model, NULL);
  infbound fx = evaluate(fv, DAG.objective);
  // output
  return 0;
}
```

A.6 Inference Engines

This section contains the class reference for the inference modules and related classes.

A.6.1 Base Classes

Termination Reason

```
/** @file termreason.h */

class termination_reason
{
public:
  termination_reason();
  termination_reason(int termr_r, const std::string& termr_ref);
  termination_reason(const termination_reason& __t);

  termination_reason& operator=(const termination_reason& __t);

  const std::string& get_message() const;

  int get_code() const;

  friend std::ostream& operator<< (std::ostream& o,
                                   const termination_reason& __x);
};
```

Information Contents

```
/** @file info_contents.h */
```

```
class info_contents : public datamap
{
public:
  info_contents();
  info_contents(const std::string& __n, const basic_alltype& __v);
  info_contents(const char* __n, const basic_alltype& __v);
  info_contents(const info_contents& __c);

  virtual ~info_contents();

  info_contents& operator=(const info_contents& __c);
};
```

Inference Engine Return Type

```
/** @file ie_rettype.h */

class ie_return_type
{
public:
  ie_return_type();
  ie_return_type(const termination_reason& __t);
  ie_return_type(const delta_base* __d, double __w);
  ie_return_type(const std::pair<delta_base*,double>& __d);
  ie_return_type(delta __d, double __w);
  ie_return_type(const std::pair<delta,double>& __d);
  ie_return_type(const std::list<delta>& __d,
                 const std::list<double>& __w);
  ie_return_type(const delta_base* __d, double __w,
                 const termination_reason& __t);
  ie_return_type(const std::pair<delta_base*,double>& __d,
                 const termination_reason& __t);
  ie_return_type(delta __d, double __w, const termination_reason& __t);
  ie_return_type(const std::pair<delta,double>& __d,
                 const termination_reason& __t);
  ie_return_type(const std::list<delta>& __d,
                 const std::list<double>& __w,
                 const termination_reason& __t);
  ie_return_type(const ie_return_type& __r);

  virtual ~ie_return_type();

  void set_termination_reason(const termination_reason& __t);

  const termination_reason& term_reason() const;

  bool set_information(const std::string& iname, const basic_alltype& a,
                       bool force = false);
  bool set_information_i(const std::string& iname, int i,
                         const basic_alltype& a, bool force = false);

  const basic_alltype& information(const std::string& iname) const;
  const basic_alltype& information(const char* iname) const;
  const basic_alltype& information(const std::string& iname, int i) const;
  const basic_alltype& information(const char* iname, int i) const;

  bool has_information(const std::string& __s) const;
  bool has_information(const char* __cp) const;
  bool has_information(const std::string& __s, int i) const;
  bool has_information(const char* __cp, int i) const;

  bool information_indices_set(const std::string& __s,
                               std::vector<int>& __idx) const;
  bool information_indices_set(const char* __cp,
                               std::vector<int>& __idx) const;

  void unset_information(const std::string& __s);
  void unset_information(const char* __cp);
  void unset_information(const std::string& __s, int i);
  void unset_information(const char* __cp, int i);

  template <class _S>
  bool retrieve_from_info(const std::string& __s,
                          const std::vector<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info(const std::string& __s,
                          const matrix<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info(const std::string& __s, _S& __b) const;

  template <class _S>
  bool retrieve_from_info(const std::string& __s,
                          const std::vector<_S>*& __b,
                          const std::vector<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info(const std::string& __s,
                          const matrix<_S>*& __b,
                          const matrix<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info(const std::string& __s, _S& __b,
                          const _S& __def) const;

  template <class _S>
  bool retrieve_from_info(const char* __s,
                          const std::vector<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info(const char* __s,
                          const matrix<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info(const char* __s, _S& __b) const;

  template <class _S>
  bool retrieve_from_info(const char* __s, const std::vector<_S>*& __b,
                          const std::vector<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info(const char* __s, const matrix<_S>*& __b,
                          const matrix<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info(const char* __s, _S& __b,
                          const _S& __def) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i,
                            const std::vector<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i,
                            const matrix<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i, _S& __b) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i,
                            const std::vector<_S>*& __b,
                            const std::vector<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i,
                            const matrix<_S>*& __b,
                            const matrix<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info_i(const std::string& __s, int i, _S& __b,
                            const _S& __def) const;

  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i,
                            const std::vector<_S>*& __b) const;
```

```
  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i,
                            const matrix<_S>*& __b) const;

  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i, _S& __b) const;

  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i,
                            const std::vector<_S>*& __b,
                            const std::vector<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i,
                            const matrix<_S>*& __b,
                            const matrix<_S>* __def) const;

  template <class _S>
  bool retrieve_from_info_i(const char* __s, int i, _S& __b,
                            const _S& __def) const;

  ie_return_type operator+(const std::pair<delta_base*,double>& __d);
  ie_return_type& operator+=(const std::pair<delta_base*,double>& __d);

  ie_return_type& operator=(const ie_return_type& __d);

  unsigned int n_deltas() const { return deltas.size(); }

  // get all deltas with weight greater or equal thresh
  // a delta can be got at most once. The method converts and
  // stores the delta in the deltas table. A second call to
  // get returns deltas which have not previously been got.
  const std::list<delta_id>& get(work_node& wn, double thresh=-INFINITY);
};
```

Inference Engine Statistics

```
/** @file inference_engine.h */

class statistic_info
{
public:
  double effectiveness;
  int number_of_infers;

  statistic_info();
  virtual ~statistic_info();
};
```

A.6.2 Inference Engines

```
class inference_engine
{
protected:
  std::string __name;
  const gptr<work_node>* __wnode;
  work_node_context* __wnc;
  vdbl::viewdbase __vdb;
  std::vector<delta_id> _old_deltas, _new_deltas;

public:
  // the constructor for a basic inference engine
  inference_engine(const gptr<work_node>& wnode, const std::string& __n);

  virtual ~inference_engine();

  virtual bool update_engine(const gptr<work_node>& wnode);

  // returns the deltas to be applied and unapplied
  std::pair<std::list<delta_id>,std::list<delta_id> > new_deltas();

  const delta* get_delta(const delta_id& __d) const;

  const model* get_model() const;

  virtual ie_return_type infer(const control_data& __c);

  const std::string& get_name() const { return __name; }

  // collect statistics
  virtual statistic_info last_call_stat();
  virtual statistic_info cumulative_stat();
};
```

A.6.3 Graph Analyzers

```
/** @file graph_analyzer.h */

typedef ie_return_type ga_return_type;

class graph_analyzer
{
protected:
  std::string __name;
  const gptr<search_graph>* __sgraph;
  search_focus* __sfoc;
  search_graph_context* __sgc;
  vdbl::viewdbase __vdb;

public:
  // the constructor for a basic graph analyzer
  graph_analyzer(const gptr<search_graph>& sgraph,
                 const search_focus& sfoc,
                 const std::string& __n);

  graph_analyzer(const gptr<search_graph>& sgraph,
                 const std::string& __n);

  virtual ~graph_analyzer();

  virtual bool update_engine(const gptr<work_node>& sgraph,
                             const search_focus& sfoc);

  virtual bool update_engine(const gptr<work_node>& sgraph);

  virtual ga_return_type analyze(const control_data& __c);

  const std::string& get_name() const;
};
```

A.7 Management Modules

This section contains the class references for the two management module base classes.

A.7.1 Management Modules

```
class management_module
{
protected:
  std::string __name;
  gptr<work_node>* __wnode;
  search_focus* __sfocus;
  const search_inspector* __sinsp;
  gptr<vdbl::database>* __dbase;
  gptr<search_graph>* __sgraph;

public:
  // the constructor for a basic management module
  management_module(const std::string& __n,
                    gptr<work_node>& wnode,
                    search_focus& sfoc,
                    const search_inspector* sinsp,
                    gptr<search_graph>& sgraph,
                    gptr<vdbl::database>& dbase);

  management_module(const std::string& __n,
                    gptr<work_node>& wnode,
                    search_focus& sfoc,
                    const search_inspector* sinsp,
                    gptr<search_graph>& sgraph);

  ...

  virtual ~management_module() {}

  const model* get_model() const;

  const std::string& get_name();

  virtual int manage(const control_data& _c);
};
```

A.7.2 Initializers

```
class initializer
{
protected:
  std::string __name;
  gptr<work_node>** __wnode;
  search_focus** __sfocus;
  gptr<vdbl::database>** __dbase;
  gptr<search_graph>** __sgraph;

public:
  // the constructor for a basic management module
  initializer(const std::string& __n,
              gptr<work_node>*& wnode,
              search_focus*& sfoc,
              gptr<search_graph>*& sgraph,
              gptr<vdbl::database>*& dbase);

  initializer(const std::string& __n,
              gptr<work_node>*& wnode,
              search_focus*& sfoc,
              gptr<search_graph>*& sgraph);
  ...

  virtual ~initializer();

  const std::string& get_name();
  virtual ie_return_type initialize(const control_data& _c);
};
```

A.8 Report Modules

In this section the class reference for the report module base class can be found.

```
/** @file report_module.h */

class report_module
{
protected:
  std::string __name;
  const gptr<work_node>* __wnode;
  search_inspector* __sgroot;
  const gptr<vdbl::database>* __dbase;
  work_node_context* __wnc;
  vdbl::viewdbase __vdb;
  const ie_return_type* __ier;

public:
  // the constructor for a basic report module
  report_module(const std::string& __n,
                const gptr<work_node>& wnode, const search_inspector& si,
                const gptr<vdbl::database>& dbase,
                const ie_return_type& _ir);

  virtual ~report_module();

  const model* get_model() const;

  virtual void print(const control_data& __c, std::ostream& o = std::cout)
                                                                      const;
  const std::string& get_name() const;
};
```

A.9 The strategy engine

Bibliography

The numbers in square brackets at the end of every entry in the bibliography are page numbers referring to the citation of the entry in the main text.

[1] F.A. Al-Khayyal and J.E. Falk. Jointly constrained biconvex programming. *Math. Operations Res.,*, 8:273–286, 1983. [80]

[2] AllFusion component modeler. Available from World Wide Web: `http://www3.ca.com/Solutions/Product.asp?ID=1003`. WWW Page. [9]

[3] I.P. Androulakis, C.D. Maranas, and C.A. Floudas. αBB: a global optimization method for general constrained nonconvex problems. *J. Global Optimization*, 7:337–363, 1995. [22, 30]

[4] K. Appel and W. Haken. Every planar map is four colorable. part i. discharging. *Illinois J. Math.*, 21:429–490, 1977. [62]

[5] K. Appel and W. Haken. Every planar map is four colorable. part ii. reducibility. *Illinois J. Math.*, 21:491–567, 1977. [62]

[6] K. Appel and W. Haken. Every planar map is four colorable. *Contemporary Math.*, 98, 1989. [62]

[7] R.W. Ashford and R.C. Daniel. *XPRESS-MP Reference Manual.* Dash Associates, Blisworth House, Northants NN73BX, 1995. [13]

[8] L.E. Baker, A.C. Pierce, and K.D. Luks. Gibbs energy analysis of phase equilibria. *Society of Petroleum Engineers Journal*, 22:731–742, 1982. [20]

[9] T. Baker, J. Gill, and R.Solovay. Relativizations of the p =? np question. *SIAM J. of Computing*, 4:431–442, 1975. [ii]

[10] V. Balakrishnan and S. Boyd. Global optimization in control system analysis and design. In C.T. Leondes, editor, *Control and Dynamic Systems: Advances in Theory and Applications*, volume 53, pages 1–56. Academic Press, New York, 1992. [19, 65]

[11] Mokhtar Bazaraa, Hanif D. Sheraldi, and C.M. Shetty. *Nonlinear Programming, Theory and Algorithms.* Wiley, Chichester, UK, 2nd edition, 1993. [32, 33]

[12] E.A. Bender. *An Introduction to Mathematical Modeling.* Dover Publications, Mineola, NY, 2000. [3]

[13] F. Benhamou, David McAllester, and Pascal Van Hentenryck. CLP intervals revisited. In Maurice Bruynooghe, editor, *Proceedings of ILPS'94, International Logic Programming Symposium*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press. [177]

[14] F. Benhamou, D. McAllister, and P. Van Hentenryck. CLP(intervals) revisited. In *Proc. International Symposium on Logic Programming*, pages 124–138, Ithaka, NY, 1994. MIT Press. [78]

[15] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1997. [88, 93]

[16] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *J. Logic Programming*, 32:1–24, 1997. [78]

[17] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997. [177]

[18] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, 1996. [-]

[19] M. Berz and J. Hoefkens. Verified high-order inversion of functional dependencies and interval Newton methods. *Reliable Computing*, 7:379–398, 2001. [77, 101]

[20] M. Berz and K. Makino. Verified integration of odes and flows using differential algebraic methods on high-order taylor models. *Reliable Computing*, 4:361–369, 1998. [99]

[21] Martin Berz. COSY INFINITY version 8 reference manual. Technical report, National Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich., 1997. MSUCL–1008. [99]

[22] G.D. Birkhoff. The reducibility of maps. *Amer. J. Math.*, 35:114–128, 1913. [62]

[23] R.E. Bixby, S. Ceria, C.M. McZeal, and M.W.P. Savelsbergh. MIPLIB 3.0, January 1996. Available from World Wide Web: `http://www.caam.rice.edu/~bixby/miplib/miplib.html`. Library of test problems. [63]

[24] Ch. Bliek. Fast evaluation of partial derivatives and interval slopes. *Reliable Computing*, 3:259–268, 1997. [84, 92]

[25] Ch. Bliek, P. Spellucci, L.N. Vicente, A. Neumaier, L. Granvilliers, E. Monfroy, F. Benhamou, E. Huens, P. Van Hentenryck, D. Sam-Haroud, and B. Faltings. Algorithms for solving nonlinear constrained and optimization problems: The state of the art. A progress report of the COCONUT project, 2001. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/glopt/coconut/StArt.html`. [124]

[26] Borland together. Available from World Wide Web: `http://www.borland.com/together/`. WWW Page. [9]

[27] J.M. Borwein and A.S.Lewis. *Convex Analysis and Nonlinear Optimization.* Springer, Berlin, 2000. [33]

[28] Hartmut Bossel. *Modellbildung und Simulation.* Vieweg, Braunschweig, 2nd edition, 1994. [3]

[29] Anthony Brooke, David Kendrick, and Alexander Meeraus. *GAMS - A User's Guide (Release 2.25).* Boyd & Fraser Publishing Company, Danvers, Massachusetts, 1992. [iv, 12, 13, 84]

[30] C. Carathéodory. Über den Variabilitätsbereich der Fourierschen Konstanten von positiven harmonischen Funktionen. *Rendiconti del Circolo Mathematico de Palermo*, 32:193–217, 1911. [33]

[31] E. Carrizosa, P. Hansen, and F. Messine. Improving interval analysis bounds by translations. *J. Global Optimization.* to appear. [71]

[32] Chandra Chekuri, Richard Johnson, Rajeev Motwani, B. Natarajan, B. Ramakrishna Rau, and Michael S. Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *International Symposium on Microarchitecture*, pages 58–67, 1996. [84]

[33] H.M. Chen and M.H. van Emden. Adding interval constraints to the Moore–Skelboe global optimization algorithm. In V. Kreinovich, editor, *Extended Abstracts of APIC'95, International Workshop on Applications of Interval Computations*, pages 54–57. Reliable Computing (Supplement), 1995. [78]

[34] D. Chiriaev and G.W. Walster. Interval arithmetic specification. Available from World Wide Web: `http://www.mscs.mu.edu/~globsol/walster-papers.html`. [69]

[35] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987. [78]

[36] COCONUT, continuous constraints – updating the technology. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/glopt/coconut.html`. WWW-Site. [iii, 124]

[37] Condor high throughput computing, version 6.5.5, September 2003. Available from World Wide Web: `http://www.cs.wisc.edu/condor/`. Software. [63]

[38] S.D. Conte and C. de Boor. *Elementary Numerical Analysis.* International Series in Pure and Applied Mathematics. McGraw-Hill, New York, 3rd edition, 1980. [iii]

[39] S. Dallwig, A. Neumaier, and H. Schichl. GLOPT – a program for constrained global optimization. In I. Bomze et al., editor, *Developments in Global Optimization*, pages 19–36. Kluwer, Dordrecht, 1997. [78]

[40] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1967. [i]

[41] G.B. Dantzig. On the significance of solving linear programming problems with some integer variables. *Econometrica*, 28:30–44, 1960. [30]

[42] G.B. Dantzig, S. Johnson, and W. White. A linear programming approach to the chemical equilibrium problem. *Management Science*, 5:38–43, 1958. [20]

[43] W.C. Davidon. Variable metric method for minimization. Report ANL-5990, Argonne National Laboratory, Chicago, 1959. Reprinted (with a new preface) in *SIAM J. Optimization* **1** (1991), 1 - 17. [30]

[44] P. Deuflhard and G. Heindl. Affine invariant convergence theorems for Newton's method and extensions to related methods. *SIAM J. Numer. Anal.*, 16:1–10, 1979. [76, 101]

[45] L.C.W. Dixon and G.P. Szegö. *Towards Global Optimization.* Elsevier, New York, 1975. [60]

[46] Enterprise architect. Available from World Wide Web: `http://www.sparxsystems.com.au/`. WWW Page. [9]

[47] W.R. Esposito and C.A. Floudas. Deterministic global optimization in nonlinear optimal control problems. *J. Global Optimization*, 17:97–126, 2000. [20]

[48] G.W. Walster et al. Extended real intervals and the topological closure of extended real numbers. Technical report, Sun Microsystems, February 2000. [69]

[49] J. Farkas. Die algebraischen Grundlagen der Anwendungen des Fourier'schen Prinzips in der Mechanik. *Math. Naturwiss. Berichte aus Ungarn*, 15:25–40, 1897–9. [39]

[50] C.L. Feffermann and L.E. Seco. Interval arithmetic in quantum mechanics. In R.B. Kearfott, editor, *Applications of interval computations, Volume 3 of Appl. Optim.*, pages 145–167. Kluwer, Dordrecht, 1995. [64]

[51] M.C. Ferris, G. Pataki, and S. Schmieta. Solving the seymour problem. *Optima*, 66:1–7, 2001. [63]

[52] R. Fletcher. *Practical Methods of Optimization.* Wiley, New York, 1987. [32]

[53] C.A. Floudas. *Nonlinear and Mixed-Integer Optimization: Fundamentals and Applications.* Oxford Univ. Press, Oxford, 1995. [i]

[54] C.A. Floudas. Deterministic global optimization in design, control, and computational chemistry. In L.T. Biegler et al., editor, *Large Scale Optimization with Applications. Part II: Optimal Design and Control*, pages 129–184. Springer, New York, 1997. Available from World Wide Web: `ftp://titan.princeton.edu/papers/floudas/ima.pdf`. [i]

[55] C.A. Floudas. *Deterministic Global Optimization: Theory, Algorithms and Applications.* Kluwer, Dordrecht, 1999. [20]

[56] C.A. Floudas, P.M. Pardalos, C.S. Adjiman, W.R. Esposito, Z.H. Gümüs, S.T. Harding, J.L. Klepeis, C.A. Meyer, and C.A. Schweiger. *Handbook of Test Problems in Local and Global Optimization.* Kluwer, Dordrecht, 1999. Available from World Wide Web: `http://titan.princeton.edu/TestProblems/`. [i, 20, 21, 22, 23]

[57] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, Brooks/Cole Publishing Company, 1993. Available from World Wide Web: `http://www.ampl.com/cm/cs/what/ampl/`. [iv]

[58] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL — A Mathematical Programming Language.* Thomson, second edition, 2003. [13, 84, 182]

[59] Emmanuel Fragnière and Jacek Gondzio. Optimization modeling languages, 1999. Available from World Wide Web: `citeseer.nj.nec.com/320279.html`. [16]

[60] A. Frommer. Proving conjectures by use of interval arithmetic. Preprint, BUGHW-SC 2001/1, University of Wuppertal, 2001. [63]

[61] J.M. Gablonsky and C.T. Kelley. A locally-biased form of the DIRECT algorithm. *J. Global Optimization*, 21:27–37, 2001. [65]

[62] C.-Y. Gau and L. Schrage. Implementation and testing of a branch-and-bound based method for deterministic global optimization. In *Proceedings of the Conference Frontiers in Global Optimization*, Santorini (Greece), June 2003. to appear. [80]

[63] A.M. Geoffrin. An Introduction to Structured Modeling. *Management Science*, 33(5):547–588, 1987. [3]

[64] Helmuth Gericke. *Mathematic in Antike, Orient und Abendland.* Fourier Verlag, Wiesbaden, sixth edition, 2003. [5]

[65] J.W. Gibbs. Graphical methods in thermodynamics of fluids. *Trans. Connecticut Acad.*, 2, 1873. [20]

[66] J.W. Gibbs. A method of geometrical representation of the thermodynamic properties of substances by means of surfaces. *Trans. Connecticut Acad.*, 2, 1873. [20]

[67] P.E. Gill, W. Murray, and M.H. Wright. *Practical Optimization.* Academic Press, London, 1981. [iii]

[68] F.R. Giordano, M.D. Weir, and W.P. Fox. *A First Course in Mathematical Modeling.* Brooks/Cole, 3rd edition, 2002. [3]

[69] GNU lesser general public license. Available from World Wide Web: `http://www.gnu.org/copyleft/lesser.html`. WWW-Document. [124]

[70] L. Granvilliers. Progress in the solving of a circuit design problem. *J. Global Optimization*, 20:155–168, 2001. [-]

[71] A. Griewank and G.F. Corliss. *Automatic Differentiation of Algorithms.* SIAM Publications, Philadelphia, 1991. [84]

[72] T.C. Hales. An overview of the Kepler conjecture. Manuscript, math.MG/9811071 - math.MG/9811078, 1998. [i, 63]

[73] T.C. Hales. Cannonballs and honeycombs. *Notices Amer. Math. Soc.*, 47:440–449, 2000. [63]

[74] E. Hansen. Preconditioning linearized equations. *Computing*, 58:187–196, 1997. [101]

[75] E.R. Hansen. Global optimization using interval analysis – the multidimensional case. *Numer. Math.*, 34:247–270, 1980. [67]

[76] E.R. Hansen. *Global Optimization Using Interval Analysis.* Dekker, New York, 1992. [67, 77]

[77] J. Hass, M. Hutchings, and R. Schlafly. The double bubble conjecture. *Electron. Res. Announc. Amer. Math. Soc.*, 1:98–102, 1995. electronic. [63]

[78] S. Heipcke. *Applications of Optimization with Xpress-MP*. Dash Optimization, Blisworth, UK, 2002. [17]

[79] Ch. Helmberg. Semidefinite programming. Available from World Wide Web: `http://www-user.tu-chemnitz.de/~helmberg/semidef.html`. WWW-Site. [-]

[80] D. Henrion and J.B. Lasserre. Detecting global optimality and extracting solutions in GloptiPoly. Manuscript, 2003. [-]

[81] D. Henrion and J.B. Lasserre. GloptiPoly: Global optimization over polynomials with Matlab and SeDuMi. *ACM Trans. Math. Software*, 29:165–194, 2003. [-]

[82] D. Henrion and J.B. Lasserre. Solving global optimization problems over polynomials with GloptiPoly 2.1. In Ch. Bliek et al., editor, *Global Optimization and Constraint Satisfaction*, pages 43–58. Springer, Berlin, 2003. [-]

[83] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989. [78]

[84] P. Van Hentenryck, L. Michel, and F. Benhamou. Newton: constraint programming over non-linear constraints. *Sci. Programming*, 30:83–118, 1997. [78]

[85] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica. A Modeling Language for Global Optimization.* MIT Press, Cambridge, MA, 1997. [13, 18, 72, 73, 78, 93, 101, 108]

[86] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms.* Springer, Berlin, 1993. [33, 42]

[87] J. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM J. Computing*, 2:88–105, 1973. [65]

[88] R. Horst, P.M. Pardalos, and N.V. Thoai. *Introduction to Global Optimization*. Kluwer, Dordrecht, 1995. [-]

[89] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer, Berlin, 2nd edition, 1990. [81]

[90] Human genome research, 2002. Available from World Wide Web: `http://www.science.doe.gov/ober/hug_top.html`. WWW document. [19]

[91] Tony Hürlimann. Computer-based mathematical modeling. Habilitation Thesis, 1997. [6]

[92] Tony Hürlimann. Modeling languages: A new paradigm of programming. *Annals of Operations Research*, on Modeling Languages and Applications, 1998. to appear. [13]

[93] Tony Hürlimann. *Mathematical Modeling and Optimization, An Essay for the Design of Computer-Based Modeling Tools*, volume 31 of *Applied Optimization*. Kluwer Academic Publishers, Dordrecht, 1999. [13]

[94] Tony Hürlimann. LPL : A mathematical modeling language, an introduction. Departement of Informatics, Fribourg, 2002. [17]

[95] W. Huyer and A. Neumaier. Global optimization by multilevel coordinate search. *J. Global Optimization*, 14:331–355, 1999. [ii, 65]

[96] W. Huyer and A. Neumaier. SNOBFIT – stable noisy optimization by branch and fit. Manuscript, 2003. [ii]

[97] *ILOG Solver 5.1*, 2001. [88]

[98] L. Ingber. Simulated annealing: Practice versus theory. *Math. Comput. Modelling*, 18:29–57, 1993. [65]

[99] R. Van Iwaarden. *An improved unconstrained global optimization algorithm*. PhD thesis, Univ. of Colorado at Denver, Denver, CO, May 1986. [101]

[100] E. Janka. A comparison of stochastic methods for global optimization, 2000. Available from World Wide Web: `http://www.mat.univie.ac.at/~vpk/math/gopt_eng.html`. WWW-Document. [ii, 65]

[101] C. Jansson. On self-validating methods for optimization problems. In J. Herzberger, editor, *Topics in validated computation*, pages 381–438. Elsevier, Amsterdam, 1994. [101]

[102] Christian Jansson. A rigorous lower bound for the optimal value of convex optimization problems. *J. of Global Optimization*, 2003. Available from World Wide Web: `http://www.ti3.tu-harburg.de/report/03.1.ps`. to appear. [199]

[103] Y. Jiang, W.R. Smith, and G.R. Chapman. Global optimality conditions and their geometric interpretation for the chemical phase and equilibrium problem. *SIAM J. on Optimization*, 5(4):813–834, 1995. [22]

[104] F. John. Extremum problems with inequalities as subsidiary conditions. In J. Moser, editor, *Studies and Essays Presented to R. Courant on his 60th Birthday January 8, 1948, Interscience, New York 1948. Reprinted as pp. 543–560 of: Fritz John, Collected Papers*, volume 2, pages 187–204. Birkhäuser, Boston, 1985. [50]

[105] D.R. Jones, M. Schonlau, and W.J. Welch. Efficient global optimization of expensive black-box functions. *J. Global Optimization*, 13:455–492, 1998. [ii]

[106] W.M. Kahan. A more complete interval arithmetic. Lecture notes for an engineering summer course in numerical analysis, University of Michigan, 1968. [66, 71, 76]

[107] J. Kallrath. *Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis.* Vieweg, 2002. [3, 6]

[108] J. Kallrath, editor. *Modeling Languages in Mathematical Optimization.* Kluwer, Dordrecht, 2003. [iv, 13]

[109] Josef Kallrath, editor. *Modeling Languages in Mathematical Optimization.* Kluwer Academic Publishers, Boston Dordrecht London, 2003. [182]

[110] L.B. Kantorovich. Functional analysis and applied mathematics. *Uspekhi Mat. Nauk*, 3:89–185, 1948. (in Russian). Translated by C.D. Benster, Nat. Bur. Stand. Rep. 1509, Washington, DC (1952). [76]

[111] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, IL, 1939. [50]

[112] R. B. Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996. [75, 84, 101]

[113] R.B. Kearfott. A review of techniques in the verified solution of constrained global optimization problems. In R.B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations*, pages 23–60. Kluwer, Dordrecht, 1996. [101]

[114] R.B. Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer, Dordrecht, 1996. [67]

[115] R.B. Kearfott. Empirical evaluation of innovations in interval branch and bound algorithms for nonlinear algebraic systems. *SIAM J. Sci. Comput.*, 18:574–594, 1997. [101]

[116] R.B. Kearfott. GlobSol: History, composition, and advice on use. In Ch. Bliek et al., editor, *Global Optimization and Constraint Satisfaction*, pages 17–31. Springer, Berlin, 2003. [51, 72, 77]

[117] R.B. Kearfott and K. Du. The cluster problem in multivariate global optimization. *J. Global Opt.*, 5:253–265, 1994. [73]

[118] R.B. Kearfott, C. Hu, and M. Novoa III. A review of preconditioners for the interval Gauss-Seidel method. *Interval Computations*, 1:59–85, 1991. [109]

[119] S. Kirkpatrick, C.D. Geddat, Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983. [-]

[120] M. Kojima, S. Kim, and H. Waki. A general framework for convex relaxation of polynomial optimization problems over cones. *J. Oper. Res. Soc. Japan*, 46:125–144, 2003. [-]

[121] M. Kojima and L. Tuncel. Cones of matrices and successive convex relaxations of nonconvex sets. *SIAM J. Optimization*, 10:750–778, 2000. [-]

[122] L.V. Kolev. Use of interval slopes for the irrational part of factorable functions. *Reliable Computing*, 3:83–93, 1997. [70, 71, 72]

[123] L.V. Kolev and I.P. Nenov. Cheap and tight bounds on the solution set of perturbed systems of nonlinear equations. *Reliable Computing*, 7:399–408, 2001. [99]

[124] J. Kostrowicki and H.A. Scheraga. Some approaches to the multiple-minima problem in protein folding. In P.M. Pardalos et al., editor, *Global Minimization of Nonconvex Energy Functions: Molecular Conformation and Protein Folding*, pages 123–132. Amer. Math. Soc., Providence, RI, 1996. [i, 26]

[125] R. Krawczyk. Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken. *Computing*, 4:187–201, 1969. [71]

[126] R. Krawczyk and A. Neumaier. Interval slopes for rational functions and associated centered forms. *SIAM J. Numer. Anal.*, 22:604–616, 1985. [72]

[127] M. Krein and D. Milman. On the extreme points of regularly convex sets. *Studia Math.*, 9:133–138, 1940. [42]

[128] H.W. Kuhn and A.W. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. Univ. of California Press, Berkeley, California, 1951. [51, 55]

[129] A. Kuntsevich and F. Kappel. SolvOpt. Available from World Wide Web: `http://www.kfunigraz.ac.at/imawww/kuntsevich/solvopt/`. WWW-Document. [106]

[130] J.-L. Lagrange. *Théorie des fonctions analytique*. 1797. (Engl. The theory of analytical functions). [29, 46]

[131] J.-L. Lagrange. *Leçons sur le calcul des fonctions*. 1806. (Engl. Lessons on the calculus of functions). [29, 46]

[132] David Lamb. CASE tool information, 2002. Available from World Wide Web: `http://www.qucis.queensu.ca/Software-Engineering/case.html`. WWW-Document. [9]

[133] A.H. Land and A.G. Doig. An automated method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960. [-]

[134] E. Lee and C. Mavroidis. Solving the geometric design problem of spatial 3r robot manipulators using polynomial homotopy continuation. *J. Mech. Design, Trans. ASME*, 124:652–661, 2002. [i, 19]

[135] E. Lee, C. Mavroidis, and J.P. Merlet. Five precision points synthesis of spatial RRR manipulators using interval analysis. In *Proc. 2002 ASME Mechanisms and Robotics Conference*, pages 1–10, 29–October 2, September 2002. Montreal. Available from World Wide Web: `http://robots.rutgers.edu/Publications.htm`. [i, 19]

[136] Eric Lee, Constantinos Mavroidis, and Jean Pierre Merlet. Five precision points synthesis of spatial RRR manipulators using interval analysis. In *Proceedings of DETC'02, 27th Biennial Mechanisms and Robotics Conference Montreal, Canada*, 2002. [18]

[137] G.W. Leibniz. Nova methodus pro maximis et minimis, itemque tangentibus, quae nec fractas nec irrationales quantitates moratur, et singulare pro illis calculi genus. *Acta eruditorum*, 3, 1684. (Engl. A new method for maxima and minima, as well as tangents, which are neither hindered by rational nor irrational quantities, and an excellent calculus for them). [29]

[138] J.D. Little, K.C. Murty, D.W. Sweeney, and C. Karel. An algorithm for the travelling salesman problem. *Operations Research*, 11:972–989, 1963. [-]

[139] C. Maheshwari, A. Neumaier, and H. Schichl. Convexity and concavity detection. In preparation, 2003. [vi]

[140] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *Int. J. Pure Applied Math.*, 4:379–456, 2003. [71]

[141] O.L. Mangasarian. Pseudo-convex functions. *SIAM Journal on Control*, 3:281–290, 1965. [37]

[142] O.L. Mangasarian. *Nonlinear Programming.* McGraw-Hill New York 1969. Reprinted as: Classics in Applied Mathematics, SIAM, Philadelphia, 1994. [37, 51]

[143] G.P. McCormick. Computability of global solutions to factorable nonconvex programs: Part i – convex underestimating problems. *Math. Programming*, 10:147–175, 1976. [79]

[144] C.M. McDonald and C.A. Floudas. Global optimization for the phase and chemical equilibrium problem: application to the NRTL equation. *Comput. Chem. Eng.*, 19:1111–1139, 1995. [22]

[145] C.M. McDonald and C.A. Floudas. GLOPEQ: A new computational tool for the phase and chemical equilibrium problem. *Computers & Chemical Engineering*, 19(11):1111–1141, 1995. [22, 23, 25]

[146] K.I.M. McKinnon and M. Mongeau. A generic global optimization algorithm for the chemical phase and equilibrium problem. *Journal of Global Optimization*, 12(4):325–351, 1998. [22]

[147] M.M. Meerschaert. *Mathematical Modeling.* Harcourt/Academic Press, 2nd edition, 1999. [3]

[148] H. Minkowski. Allgemeine Lehrsätze über die konvexen Polyeder. *Nachr. Gesellschaft der Wiss. Gottingen*, pages 198–219, 1896. [33]

[149] H. Minkowski. Theorie der Konvexen Körper, insbesondere Begründung ihres Oberflächenbegriffs. In D. Hilbert, A. Speiser, and H. Weyl, editors, *Gesammelte Abhandlungen von Hermann Minkowski, Volume II*, pages 131–229. B.G. Teubner, Leipzig, Berlin, 1911. [35]

[150] M. Mongeau, H. Karsenty, V. Rouz, and J.-B. Hiriart-Urruty. Comparison of public-domain software for black box global optimization. *Optimization Methods Software*, 13:203–226, 2000. [65]

[151] R.E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979. [66, 73]

[152] R.E. Moore and C.T. Yang. Interval analysis i. Technical Report Space Div. Report LMSD285875, Lockheed Missiles and Space Co., 1959. Available from World Wide Web: `http://interval.louisiana.edu/Moores_early_papers/Moore_Yang.pdf`. [66, 73]

[153] T.S. Motzkin and E.G. Strauss. Maxima for graphs and a new proof of a theorem of Turan. *Canad. J. Math.*, 17:533–540, 1965. [19]

[154] R.L. Muhanna and R.L. Mullen. Critical issues in the application of interval methods to finite element analysis, 2002. Available from World Wide Web: `http://ecivwww.cwru.edu/civil/rlm/scan2002/SCAN2002_files/v3_document.htm`. Slides, SCAN 2002, [65]

[155] Musser and Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))*, 1989. Available from World Wide Web: `citeseer.nj.nec.com/musser88generic.html`. [126]

[156] I.P. Nenov and D.H. Fylstra. Interval methods for accelerated global search in the Microsoft Excel solver. *Reliable Computing*, 9:143–159, 2003. [-]

[157] NEOS server for optimization. Available from World Wide Web: `http://www-neos.mcs.anl.gov`. [-]

[158] A. Neumaier. The enclosure of solutions of parameter-dependent systems of equations. In R.E. Moore, editor, *Reliability in Computing*, pages 269–286. Acad. Press, San Diego, 1988. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/publist.html\#encl`. [-]

[159] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge Univ. Press, Cambridge, 1990. [67, 70]

[160] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge Univ. Press, Cambridge, 1990. [101, 107, 108, 113]

[161] A. Neumaier. Second-order sufficient optimality conditions for local and global nonlinear programming. *J. Global Optimization*, 9:141–151, 1996. [57]

[162] A. Neumaier. Molecular modeling of proteins and mathematical prediction of protein structure. *SIAM Review*, 39:407–460, 1997. [i, 25, 26]

[163] A. Neumaier. Optimization — theory and algorithms, 2000. Lecture notes, book in preparation. [32, 57, 58]

[164] A. Neumaier. *Introduction to Numerical Analysis*. Cambridge Univ. Press, Cambridge, 2001. [iii]

[165] A. Neumaier. Taylor forms – use and limits. *Reliable Computing*, 9:43–79, 2002. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/papers.html\#taylor`. [71, 77, 99]

[166] A. Neumaier. Constraint satisfaction and global optimization in robotics. Manuscript, 2003. [i]

[167] A. Neumaier. Mathematical model building. In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization*. Kluwer, Dordrecht, 2003. Chapter 3. [6]

[168] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. In A. Iserles, editor, *Acta Numerica 2004*. Cambridge University Press, Cambridge, 2004. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/papers.html##glopt03`. [iii, v, 60, 65, 73, 81]

[169] A. Neumaier and A. Pownuk. FEM structural analysis, 2003. Manuscript, [65]

[170] A. Neumaier and H. Schichl. Sharpening the Karush-John optimality conditions. Manuscript, 2003. [51]

[171] Arnold Neumaier. NOP - A Compact Input Format for Nonlinear Optimization Problems. In I.M. Bomze, T. Csendes, R. Horst, and P.M. Pardalos, editors, *Developments in Global Optimization*, pages 1–18. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997. [13, 18]

[172] Sir I. Newton. *Philosophiae naturalis principia mathematica*. 1687. (Engl. The mathematical principles of natural philosophy). [29]

[173] Sir I. Newton. *Methodus fluxionum et serierum infinitarum*. 1736. (Engl. The method of fluxes and infinite series), written in 1671. [29]

[174] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 1999. [iii, 32]

[175] W. Older and A. Vellino. Constraint arithmetic on real intervals. In F. Benhamou and A. Colmerauer, editors, *Constrained Logic Programming: Selected Research*. MIT Press, Cambrige, MA, 1993. [-]

[176] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*, volume 30 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 2000. [101]

[177] C.H. Papadimitrou. The Euclidean traveling salesman problem is np-complete. *Theoretical Computer Science 4*, 3:237–244, 1977. [ii]

[178] P.M Pardalos and J.B. Rosen. *Constrained Global Optimization: Algorithms and Applications*, volume 268 of *Lecture Notes in Computer Science*. Springer, Berlin, 1987. [-]

[179] P.M. Pardalos and G. Schnitger. Checking local optimality in constrained quadratic programming is NP-hard. *Oper. Res. Lett.*, 7:33–35, 1988. [ii]

[180] P.A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Math. Programming B*, 96:293–320, 2003. [-]

[181] J.D. Pintér. *LGO – A Model Development System for Continuous Global Optimization. User s Guide.* Pintér Consulting Services, Inc., Halifax, NS. [-]

[182] J.D. Pintér. *Global Optimization in Action.* Kluwer, Dordrecht, 1996. [-]

[183] Python. Available from World Wide Web: `http://www.python.org`. WWW-Site. [126]

[184] Rational rose. Available from World Wide Web: `http://www-3.ibm.com/software/awdtools/developer/plus`. WWW Page. [9]

[185] H. Renon and J.M. Prausnitz. Local compositions in thermodynamic excess functions for liquid mixtures. *AIChE (American Institute of Chemical Engineers) J.*, 14:135, 1968. [22]

[186] F. Richards. The protein folding problem. *Scientific American*, 264:54–63, January 2001. [26]

[187] R.T. Rockafellar. *Convex Analysis.* Princeton University Press, Princeton, 1970. [33]

[188] R.T. Rockafellar and R.J.-B. Wets. *Variational Analysis.* Springer, Berlin, 1998. [33]

[189] S.M. Rump. Expansion and estimation of the range of nonlinear functions. *Math. Comp.*, 65:1503–1512, 1996. [72]

[190] S.M. Rump. INTLAB – INTerval LABoratory. In T. Csendes, editor, *Developments in reliable computing*, pages 77–104. Kluwer, Dordrecht, 1999. Available from World Wide Web: `http://www.ti3.tu-harburg.de/rump/intlab/index.html`. [72]

[191] H.S. Ryoo and N.V. Sahinidis. A branch-and-reduce approach to global optimization. *J. Global Optimization*, 8:107–139, 1996. [-]

[192] N.V. Sahinidis. BARON. branch and reduce optimization navigator. user's manual. Available from World Wide Web: `http://archimedes.scs.uiuc.edu/baron/baron.html`. WWW-Document. [-]

[193] N.V. Sahinidis. BARON: A general purpose global optimization software package. *J. Global Optimization*, 8:201–205, 1996. [30]

[194] N.V. Sahinidis. Global optimization and constraint satisfaction: The branch-and-reduce approach. In Ch. Bliek et al., editor, *Global Optimization and Constraint Satisfaction*, pages 1–16. Springer, Berlin, 2003. [-]

[195] D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1&2):85–118, Sep 1996. [88, 93]

[196] D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1:85–118, 1996. [178]

[197] H. Schichl. The COCONUT API version 2.32, reference manual. Technical Report Appendix to ”Specification of new and improved representations”, Deliverable D5v2, the COCONUT project, November 2003. Available from World Wide Web: `http://www.mat.univie.ac.at/coconut-environment`. [125, 170]

[198] H. Schichl. VDBL (Vienna Database Library) version 1.0, reference manual. Technical Report Appendix to ”Upgraded State of the Art Techniques implemented as Modules”, Deliverable D13, the COCONUT project, July 2003. Available from World Wide Web: `http://www.mat.univie.ac.at/coconut-environment`. [125]

[199] H. Schichl. VGTL (Vienna Graph Template Library) version 1.0, reference manual. Technical Report Appendix to ”Upgraded State of the Art Techniques implemented as Modules”, Deliverable D13, the COCONUT project, July 2003. Available from World Wide Web: `http://www.mat.univie.ac.at/coconut-environment`. Version 1.1, October 2003. [125, 133]

[200] H. Schichl and A. Neumaier. Exclusion regions for systems of equations. *SIAM J. Numer. Anal.*, 2003. to appear. [72, 73, 101, 106, 110, 170]

[201] H. Schichl and A. Neumaier. Global optimization on directed acyclic graphs, 2003. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/papers.html`. Manuscript. [84, 173, 174]

[202] Hermann Schichl and Arnold Neumaier. Exclusion regions for systems of equations. *SIAM J. Num. Analysis*, 2003. to appear. [177]

[203] Hermann Schichl, Arnold Neumaier, and Stefan Dallwig. The NOP-2 modeling language. *Ann. Oper. Research*, 104:281–312, 2001. [13, 18]

[204] Linus Schrage. *Optimization Modeling with LINGO.* LINDO Systems, Inc., Chicago, Il, 1999. [13]

[205] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, Chichester, 1986. [41, 51]

[206] C.A. Schweiger, A. Rojnuckarin, and C.A. Floudas. *MINOPT: A Software Package for Mixed-Integer Nonlinear Optimization.* Dept. of Chemical Engineering, Princeton University, Princeton, NJ 08544, June 1996. [13]

[207] O. Shcherbina, A. Neumaier, Djamila Sam-Haroud, Xuan-Ha Vu, and Tuan-Viet Nguyen. Benchmarking global optimization and constraint satisfaction codes. In Ch. Bliek et al., editor, *Global Optimization and Constraint Satisfaction*, pages 211–222. Springer, Berlin, 2003. Available from World Wide Web: `http://www.mat.univie.ac.at/~neum/papers.html\#bench`. [ii]

[208] Z. Shen and A. Neumaier. The Krawczyk operator and Kantorovich’s theorem. *J. Math. Anal. Appl.*, 149:437–443, 1990. [70, 72, 77, 101]

[209] M. Silaghi, D. Sam-Haroud, and B. Faltings. Search techniques for non-linear CSPS with inequalities. In *Proceedings of the 14th Canadian Conference on AI*, 2001. [178]

[210] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Search techniques for non-linear csps with inequalities. In *Proceedings of the 14th Canadian Conference on AI*, 2001. [188]

[211] M. Sipser. *Introduction to the theory of computation.* PWS Publ. Comp., 1997. [ii]

[212] S. Skelboe. Computation of rational interval functions. *BIT*, 14:87–95, 1974. Available from World Wide Web: `http://www.diku.dk/~stig/CompRatIntv.pdf`. [73]

[213] S. Smale. Mathematical problems for the next century. In V. Arnold, M. Atiyah, P. Lax, and B. Mazur, editors, *Frontiers and Perspectives 2000.* Amer. Math. Soc., Providence, RI, 2000. [64]

[214] W.R. Smith and R.W. Missen. *Chemical Reaction Equilibrium Analysis: Theory and Algorithms.* Wiley & Sons, Chichester, 1982. [20]

[215] I. Sommerville. *Software Engineering.* Addison-Wesley, Reading, MA, 6th edition, 2000. [9]

[216] P. Spellucci. An SQP method for general nonlinear programs using only equality constrained subproblems. *Mathematical Programming*, 82:413–448, 1998. [51, 177, 191]

[217] A.A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard, 1994. Available from World Wide Web: `citeseer.nj.nec.com/stepanov95standard.html`. [126]

[218] J. Stoer and C.Witzgall. *Convexity and Optimization in Finite Dimensions I.* Springer, Berlin, 1970. [33]

[219] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 3rd edition, 1997. [124]

[220] M. Tawarmalani and N.V. Sahinidis. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications.* Kluwer, Dordrecht, 2002. [80]

[221] J.A. Trangenstein. Customized minimization techniques for phase equilibrium computations in reservoir simulation. *Chemical Engineering Science*, 42(12):2847–2863, 1987. [22]

[222] W. Tucker. A rigorous ODE solver and Smale's 14th problem. *Found. Comput. Math.*, 2:53–117, 2002. [64]

[223] H. Tuy. D.C. optimization: Theory, methods and algorithms. In R. Horst and P.M. Pardalos, editors, *Handbook of Global Optimization*, pages 149–216. Kluwer, Dordrecht, 1995. [81]

[224] Xuan-Ha Vu, Djamila Sam-Haroud, and Marius-Calin Silaghi. Approximation techniques for non-linear problems with continuum of solutions. In *Proceedings of The 5th International Symposium on Abstraction, Reformulation and Approximation (SARA'2002)*, pages 224–241, Canada, August 2002. [188]

[225] Xuan-Ha Vu, Djamila Sam-Haroud, and Marius-Calin Silaghi. Numerical constraint satisfaction problems with non-isolated solutions. In *1st International Workshop on Constraint Satisfaction and Global Optimization (COCOS'2002)*, France, October 2002. [188]

[226] S. Willard. *General Topology.* Addison-Wesley, Reading, MA, 1970. [34]

[227] H.P. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons, Chichester, 3rd edition, 1993. [3]

[228] H.P. Williams. *Model Solving in Mathematical Programming.* Wiley, Chichester, 4th edition, 1999. [6]

[229] R.J. Wilson. *Introduction to Graph Theory.* Addison-Wesley, Reading, MA, 4th edition, 1997. [62]