A SCIP Constraint Handler for Piecewise Linear Functions

Masterarbeit

im Studiengang Wirtschaftsmathematik

vorgelegt von:	Tom Walther
Matrikelnummer:	328 144
Erstgutachter:	Prof. Dr. Dr. h.c. mult. Martin Grötschel
Zweitgutachter:	Dr. Benjamin Hiller
Abgabedatum:	18. Januar 2014

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Masterarbeit ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Berlin, den 18. Januar 2014

Tom Walther.

Acknowledgements

The work on this thesis has been conducted at Konrad-Zuse-Zentrum für Informationstechnik, Berlin, where I have gained invaluable insights into real-world mathematics during my time as student assistant.

Most of all, I wish to thank Benjamin Hiller for his constant support, his useful suggestions, critical comments and inspiring ideas. Our regular meetings and discussions helped a lot!

Special thanks to Stefan Heinz for his instant troubleshooting whenever I had any issue with SCIP. Without him, I would have been in desperation more than once.

Finally, I want to thank Prof. Martin Grötschel for supervising this thesis.

Berlin, July 2013.

Tom Walther

Zusammenfassung

Diese Arbeit untersucht verschiedene Modellierungsansätze für stückweise lineare Funktion mit besonderem Hinblick auf deren Anwendung in mathematischen Optimierungsproblemen.

Zu Beginn werden klassische Modelle aus dem Gebiet der gemischt-ganzzahligen Programmierung (*Mixed-Integer Programming*, MIP) vorgestellt und anhand zweier Qualitätsmerkmale analysiert. Es stellt sich heraus, dass alle Ansätze bis auf die wohlbekannte λ-Methode sowohl scharfe als auch lokal ideale Formulierungen liefern.

Der Hauptteil der Arbeit beschäftigt sich jedoch mit einer alternativen Modellierungsvariante im Rahmen des sogenannten *Constraint Programming* (CP). Ziel ist es dabei, eine stückweise lineare Funktionen enthaltende Klasse von Nebenbedingungen unter Ausnutzung ihrer Struktur und ohne die Notwendigkeit zusätzlicher Hilfsvariablen direkt im Lösungsprozess der Optimierungsaufgabe zu behandeln. Dazu wird eine Relaxierung der Nebenbedingung unter Verwendung ihrer konvexen Hülle vorgeschlagen und ein auf selbige zugeschnittenes *Branch-and-Bound*-Lösungsverfahren entwickelt. Dieses wird mitsamt aller seiner Komponenten ausführlich beschrieben und beurteilt. Besonderes Augenmerk liegt auf der Entwicklung geeigneter Strategien für die Auswahl der Branchingvariablen. Dabei werden verschiedene aus der MIP-Theorie bekannte Ansätze auf die vorgestellte Formulierung stückweise linearer Nebenbedingungen übertragen und angepasst.

Eine wesentliche Komponente dieser Arbeit stellt die Implementierung des CP-Ansatzes für den Spezialfall univariater Funktionen als *Constraint Handler* innerhalb der am Konrad-Zuse-Institut Berlin entwickelten Optimierungssoftware SCIP dar. Für die Analyse der Leistungsfähigkeit dieses Constraint Handlers hinsichtlich verschiedener Aspekte wird eine Anwendung aus dem Bereich der Netzwerkoptimierung für Gastransport betrachtet. Im sogenannten Nominierungsvalidierungsproblem werden stückweise lineare Nebenbedingungen als Approximationen für nichtlineare gasphysikalische Zusammenhänge eingesetzt. Die erzielten Rechenergebnisse zeigen, dass die vorgeschlagene CP-Formulierung zu einer Verringerung der benötigten Knoten im Branch-and-Bound-Baum im Vergleich zu den gängigen MIP-Modellen führen kann. Als weitere Erkenntnis ist festzuhalten, dass sich in der Bewertung von Branchingkandidaten eine gemeinsame Evaluation fraktionaler Binärvariablen und stückweise linearer Nebenbedingungen als vorteilhaft gegenüber der Priorisierung eines der beiden Typen erweist. Aufgrund einer fehlenden Funktionalität in SCIP, die eine verlässliche Auswahl der besten Branchingentscheidung nur unter unverhältnismäßig großem Zeitaufwand zulässt, ist es nicht gelungen, Probleminstanzen von praxisrelevanter Komplexität zu lösen.

Contents

List of Figures

List of Tables

1 Introduction

Piecewise linear functions play an important role in many practical applications, either occurring generically or as linearizing approximations to non-linear functions. An example for the former can be found in the field of production planning, where some item might be produced at different scales (e.g., small, medium and large scale) with each of these yielding a distinct constant production cost per item [13]. Then, the revenue as a function of items sold may be described in a piecewise linear manner. Linearizations, on the other hand, come into use whenever non-linear functions are undesirable, for instance, due to their unfavourable computational properties. Mathematical Programming can be named as one of these cases, in which the treatment of non-linear functions is difficult with respect to numerical stability and computational efficiency.

1.1 Piecewise Linear Functions

There exist several ways of defining the concept of a piecewise linear function from which we have chosen a very general one to be presented here:

Definition 1 (Piecewise linear function)**.** *Let $\mathcal{D}$ be a compact domain within $\mathbb{R}^d$. A function $f : \mathcal{D} \to \mathbb{R}$ is called piecewise linear if $\mathcal{D}$ can be partitioned into a finite set of polytopes $\mathcal{P}$ with f being defined as affine function over each $P \in \mathcal{P}$. That is, f can be written as*

$$f(\mathbf{x}) := \{f_P(\mathbf{x})\}_{P\in\mathcal{P}} = \{\mathbf{m}_P\mathbf{x} + n_P\}_{P\in\mathcal{P}}$$

for $\mathbf{x} \in P$ and with $\mathbf{m}_P \in \mathbb{R}^d$ and $n_P \in \mathbb{R}$ for each $P \in \mathcal{P}$.

For our purposes, we will require all piecewise linear functions to be continuous and the partition $\mathcal{P}$ to be a simplicial triangulation which will be defined later. Furthermore, we will distinguish between univariate and multivariate functions.

The main objective of this thesis is the study of mathematical optimization problems that contain piecewise linear dependencies between variables as part of their constraints. In particular, we will discuss and evaluate several modelling and solving approaches. For a practical analysis, we have implemented a constraint handler that is capable of treating a certain class of piecewise linear constraints within the solution framework SCIP that is being developed at Konrad Zuse Institute, Berlin.

1.2 Constraint Programming

As an introduction, we want to give a short overview of the terminology and fundamental concepts of mathematical optimization that we will make use of throughout this

thesis. The starting point for all our considerations is the very general class of constraint programming problems. It consists of optimizing an objective function subject to a finite set of constraints whose type or form is not further specified.

Definition 2 (Constraint program)**.** *Let $f : \mathcal{D} \to \mathbb{R}$ be an objective function defined on a domain $\mathcal{D} \subset \mathbb{R}^d$ and let $\mathfrak{C} := \{C_1..C_m\}$ denote a set of constraints. Then, a constraint program can be defined as*

$$\min\{f(\mathbf{x}) \mid \mathfrak{C}(\mathbf{x}),\ \mathbf{x} \in \mathcal{D}\}. \tag{CP}$$

With $\mathcal{D} := D_1 \times .. \times D_d$, $\mathbf{x}$ can be written as a tuple $(x_1..x_d)$ where each $x_j \in D_j$. For any given $\mathbf{x} \in \mathcal{D}$ and for each constraint $C \in \mathfrak{C}$, it must be possible to decide whether $\mathbf{x}$ satisfies or violates C. We say that some $\mathbf{x} \in \mathcal{D}$ is a feasible solution for (CP) if $\mathbf{x}$ satisfies all constraints in $\mathfrak{C}$. A feasible solution $\mathbf{x}^*$ is optimal if $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all feasible $\mathbf{x}$. In this case, we write $f^* := f(\mathbf{x}^*)$. If no feasible solution exists, the entire problem (CP) is called infeasible. Finally, a problem can be unbounded if there is no lower bound on the objective function for feasible $\mathbf{x}$, which will be denoted by $f^* := -\infty$.

In accordance with Hooker [18], A general solving procedure for (CP) can be described as the combination of three main interacting ingredients:

- The **search** component is responsible for finding the best solution within the domain that satisfies all existing constraints. This may be achieved by branching, i.e., recursively splitting the searched region into smaller pieces, thus creating a branching tree with each node representing a restricted version of the original problem. A primal solution of (CP) is found once that the domain of each variable has been reduced to a single value. By contrast, a subproblem is infeasible and can be discarded if the domain of any variable has become empty.

- **Inference** helps directing the search into the right direction by gathering and exploiting information that arises during the solving process. This information can either be deduced directly from the constraints or from the objective function together with a feasible solution. As a typical example, bound propagation may be performed at every node of the branching tree in order to reduce the variables' domains. Inference plays a crucial role for the performance of constraint programming solvers in practice.

- By **relaxation**, we understand the enlargement of the search space that may facilitate the solving process whenever the feasible region of the problem is, in some manner, difficult to handle. For example, a continuous linear relaxation creates a polyhedron that contains the original search space. The optimal solution of the relaxed problem may then either be feasible (and thus optimal) for the original problem or provide a lower bound on its objective value that may be used to prune the branching tree.

As a special case of (CP) with all constraints and the objective function being linear and all variables being either integral or real-valued, mixed-integer programs (MIP) have become a significant topic of research due to their practical applicability:

Definition 3 (Mixed-integer program)**.** *Let $A \in \mathbb{R}^{m \times d}$ be a matrix of constraint coefficients, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^d$ vectors and $I \subseteq \{1..d\}$ a set of indices. A mixed-integer program can then be stated as*

$$c^* = \min\{c^T \mathbf{x} \mid A\mathbf{x} \leq b,\ \mathbf{x} \in \mathbb{R}^d,\ x_i \in \mathbb{Z} \text{ for all } i \in I\}. \tag{MIP}$$

Mixed-integer problems, too, can be solved by iteratively considering subproblems in a branching tree, i.e., by a branch-and-bound or branch-and-cut method. For each of those subproblems, the linear programming (LP) relaxation is solved and, in case that some integrality condition is violated, subsequently strengthened by valid inequalities that cut off the current LP solution. Therefore, such inequalities are called cutting planes.

Usually, LP relaxations yield stronger bounds than those that the propagation methods of a CP solver can provide. Moreover, highly specific algorithms have been developed in the context of mixed-integer programming, e.g., for computing cutting planes. By contrast, constraint programming allows for greater modelling flexibility such that the structure of a problem can be captured more directly.

1.3 Outline

In chapter 2, we will present several classical methods of modelling piecewise linear functions either using binary variables or special ordered sets, thereby analysing some of their properties such as sharpness and local ideality. Afterwards, chapter 3 deals with issues of piecewise linear functions in the context of mathematical optimization with special focus being laid on various aspects and techniques of constraint programming. That is, we will introduce a convex hull relaxation as a way of directly incorporating constraints involving piecewise linear functions into optimization problems without the necessity of additional variables. Next, in chapter 4, we will explain in detail how the presented ideas and concepts can be applied using SCIP and give some computational results. Finally, we will summarize the insights and give a short outlook.

2 MIP Models for Piecewise Linear Functions

Piecewise linear functions can be modelled via mixed-integer programming formulations. We will thereby distinguish between two classes of functions, so-called separable and non-separable functions, and present several approaches for both of them that can be found in the literature. In general, we assume all functions to be continuous and to be defined over compact domains as in definition 1.

Definition 4 (Separability)**.** *A function $f : \mathbb{R}^d \to \mathbb{R}$ is called (additively) separable if it can be expressed as a sum of univariate functions $f_i : \mathbb{R} \to \mathbb{R}$ for $i = 1..d$, i.e.,*

$$f(\mathbf{x}) = f(x_1..x_d) = \sum_{i=1}^{d} f_i(x_i).$$

Otherwise, the function f is called non-separable.

In the case of separable functions, we can restrict our considerations to MIP models for univariate functions as these can then easily be extended.

2.1 Univariate Piecewise Linear Functions

Considering functions of only one real variable, a continuous piecewise linear function $f : \mathbb{R} \to \mathbb{R}$ can be defined by $n+1$ nodes $\bar{x}_j$ with $j = 0..n$ and the function values $\bar{y}_j := f(\bar{x}_j)$ at these nodes. Hence, the graph of such a function consists of n connected line segments. The points $\{(\bar{x}_j, \bar{y}_j)\}_{j=0}^{n}$ are referred to as support points of f. An example of a univariate piecewise linear function is given in figure 1. All MIP models presented in the following introduce additional variables in order to describe the shape of a piecewise linear function, together with a set of combinatorial constraints. An overview of these models has been provided by Geissler [14].

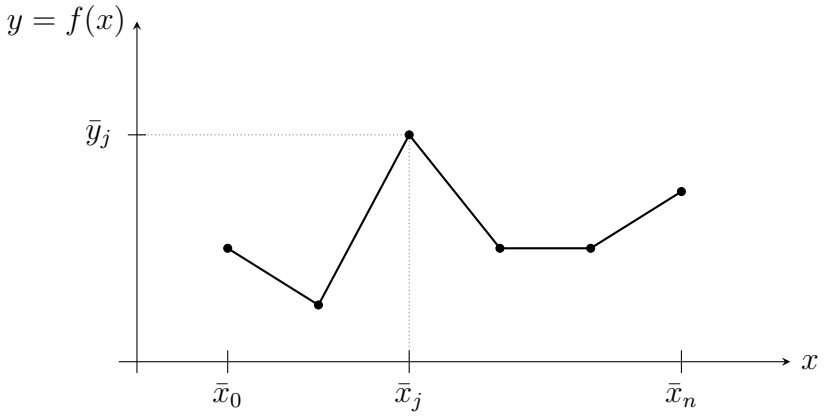


Figure 1: A univariate piecewise linear function.

2.1.1 Convex Combination Method

The most widely known MIP formulation of a piecewise linear function uses convex combinations of two neighbouring nodes to compute the function values for all points that lie in-between those nodes. We therefore introduce a set of multipliers $\Lambda := \{\lambda_0 .. \lambda_n\}$ with each $\lambda_j \in [0, 1]$ corresponding to node $\bar{x}_j$. Due to these multipliers, the formulation is sometimes also referred to as λ-method. Furthermore, one binary variable z_j is assigned to each interval $[\bar{x}_{j-1}, \bar{x}_j]$, $j = 1..n$. In order to simplify the formulation, we add fixed artificial variables $z_0 = z_{n+1} = 0$ and finally arrive at the following MIP model (CC) [11]:

$$x = \sum_{i=0}^{n} \lambda_i \bar{x}_i \tag{1}$$

$$y = \sum_{i=0}^{n} \lambda_i \bar{y}_i \tag{2}$$

$$\sum_{i=0}^{n} \lambda_i = 1 \tag{3}$$

$$\sum_{i=1}^{n} z_i = 1 \tag{4}$$

$$0 \leq \lambda_j \leq z_j + z_{j+1} \qquad j = 0..n \tag{5}$$

$$z_j \in \{0, 1\} \qquad j = 1..n \tag{6}$$

From (4) we can conlude that only one binary variable z_j is allowed to be 1 whereas all others must be equal to 0. Constraints (5) then ensure that if $z_j = 1$, only λ_{j-1} and λ_j can have a positive value, which finally yields $x \in [\bar{x}_{j-1}, \bar{x}_j]$. Because all pairs of subsequent intervals $[\bar{x}_{j-1}, \bar{x}_j]$ and $[\bar{x}_j, \bar{x}_{j+1}]$, $j = 1..n-1$, share a common multiplier λ_j at node $\bar{x}_j$, the given formulation is known as the aggregated version of the convex combination method. Its functioning is illustrated in figure 2.

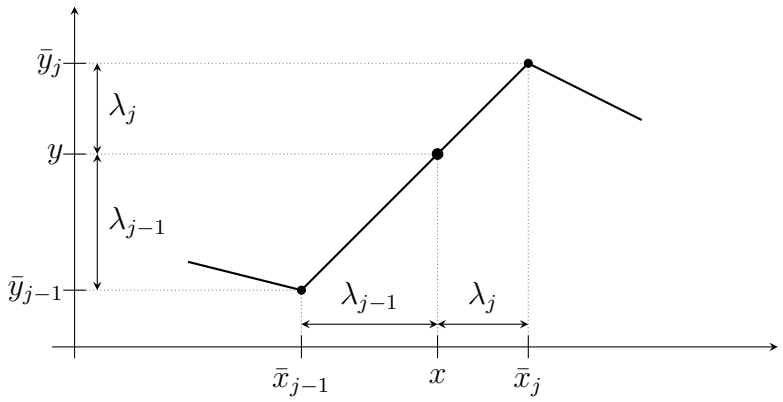


Figure 2: The Convex Combination Method.

By contrast, the so-called disaggregated version uses distinct λ-variables for each interval, i.e., a set of multipliers $\Lambda^* := \{\lambda_1^l, \lambda_1^u .. \lambda_n^l, \lambda_n^u\}$. The MIP model (CC*) resulting from these changes can be formulated as follows:

$$x = \sum_{i=1}^{n} \lambda_i^l \bar{x}_{i-1} + \lambda_i^u \bar{x}_i \tag{1*}$$

$$y = \sum_{i=1}^{n} \lambda_i^l \bar{y}_{i-1} + \lambda_i^u \bar{y}_i \tag{2*}$$

$$\sum_{i=1}^{n} \lambda_i^l + \lambda_i^u = 1 \tag{3*}$$

$$\sum_{i=1}^{n} z_i = 1 \tag{4*}$$

$$\begin{aligned} \lambda_j^l + \lambda_j^u &\leq z_j \\ \lambda_j^l, \lambda_j^u &\geq 0 \end{aligned} \qquad j = 1..n \tag{5*}$$

$$z_j \in \{0,1\} \qquad j = 1..n \tag{6*}$$

It uses considerably more variables than (CC), which, however, is outweighed by a gain in terms of tightness of the formulation, as we will see in detail in section 2.2.

2.1.2 Incremental Method

For any point $x \in [\bar{x}_{j-1}, \bar{x}_j]$, $j = 1..n$, its convex combination representation in $\bar{x}_{j-1}$ and $\bar{x}_j$ can also be written as $x = \bar{x}_{j-1} + (\bar{x}_j - \bar{x}_{j-1})\delta_j$, now using a set of variables $\Delta = \{\delta_1..\delta_n\}$ with all $\delta_j \in [0,1]$ that gives rise to the model's alternative name δ-method. This time, employing $n-1$ binary variables $z_1 .. z_{n-1}$, i.e., one less than previously, leads to the following formulation (INC) [24]:

$$x = \bar{x}_0 + \sum_{i=1}^{n} (\bar{x}_i - \bar{x}_{i-1})\delta_i \tag{7}$$

$$y = \bar{y}_0 + \sum_{i=1}^{n} (\bar{y}_i - \bar{y}_{i-1})\delta_i \tag{8}$$

$$\begin{aligned} &\delta_1 \leq 1 \\ &\delta_j \geq z_j \geq \delta_{j+1} \qquad j = 1..n-1 \\ &\delta_n \geq 0 \end{aligned} \tag{9}$$

$$z_j \in \{0,1\} \qquad j = 1..n-1 \tag{10}$$

Constraints (9) imply that if $z_j = 1$, all variables z_i and δ_i with $1 \leq i \leq j$ will also be set to 1. For having this property these constraints are called filling conditions. For the smallest index j with $z_j = 0$, δ_j is allowed to take any value between 0 and 1, while all variables z_i and δ_i with $i > j$ are forced to 0. From this, again, it follows that $x \in [\bar{x}_{j-1}, \bar{x}_j]$.

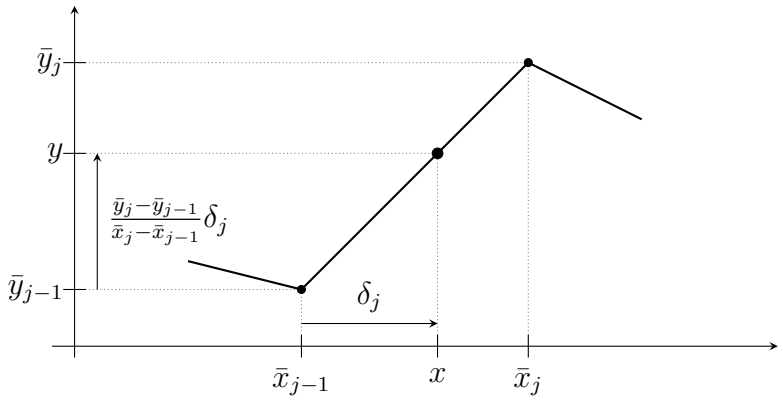


Figure 3: The Incremental Method.

2.1.3 Multiple Choice Method

A slight modification of the incremental model, in which several or even all binary variables may have value 1, leads to another formulation, called multiple choice model (MC) [4], that allows no more than one binary variable to be positive:

$$x = \bar{x}_0 + \sum_{i=1}^{n} [(\bar{x}_{i-1} - \bar{x}_0) z_i + (\bar{x}_i - \bar{x}_{i-1}) \delta_i] \tag{11}$$

$$y = \bar{y}_0 + \sum_{i=1}^{n} [(\bar{y}_{i-1} - \bar{y}_0) z_i + (\bar{y}_i - \bar{y}_{i-1}) \delta_i] \tag{12}$$

$$\sum_{i=1}^{n} z_i \leq 1 \tag{13}$$

$$0 \leq \delta_j \leq z_j \qquad j = 1..n \tag{14}$$

$$z_j \in \{0, 1\} \qquad j = 1..n \tag{15}$$

The restriction mentioned above is ensured by (13). By constraint (14), a variable δ_j, $j = 1..n$, may only take a positive value if $z_j = 1$, thereby determining the position of x within the interval $[\bar{x}_{j-1}, \bar{x}_j]$.

2.1.4 Logarithmic Models

In general, with a set of n binary variables one can express 2^n different states of a system. Conversely, in order to represent n states it would be sufficient to use only $\lceil \log_2 n \rceil$ binary variables. Based on this insight, Vielma and Nemhauser [33] developped the idea of modelling piecewise linear functions as a MIP that only uses a logarithmic number of binary variables in the number of line segments. In their original paper, they concentrated on a logarithmic version of (CC), but the same approach can be applied to other formulations as well.

As a start, it is useful to introduce the concept of so-called Special Ordered Sets (SOS) that are characterised by their elements following a particular structure. They were first mentioned by Tomlin and Beale [31] and typically come in two fashions:

- **SOS-1**: Set of variables where at most one variable may be non-zero.
- **SOS-2**: Set of variables where at most two variables may be non-zero. If two variables are non-zero, then they must be adjacent.

Logarithmic formulation of (CC):

It follows from the constraint set of (CC) that the definition of SOS-2 applies to the variables in $\Lambda = \{\lambda_0..\lambda_n\}$. Since the SOS-2 conditions only allow for n different cases of assigning zero or non-zero values to all λ_j, we should be able to model them with $\lceil \log_2 n \rceil$ binary variables.

The basic tool making this possible will be a Gray code, i.e. a dual enconding scheme in which neighbouring code words only differ by one dual digit. To generate this code for each line segment of a piecewise linear function, we use an injective function $c : \{1..n\} \to \{0,1\}^{\lceil \log_2 n \rceil}$ which must furthermore guarantee that, for $k = 1..n-1$, all but one component of $c(k)$ and $c(k+1)$ have the same values.

Assuming that we have found such a function, we now proceed with defining an incidence relation for any number $i \in \{0..n\}$:

$$N(i) := \begin{cases} \{1\} & i = 0 \\ \{n\} & i = n \\ \{i, i+1\} & otherwise \end{cases} \tag{16}$$

To make sure that for every function value of $c(\cdot)$ only those λ-values adjacent to the corresponding line segment are allowed to be positive, we introduce the following index sets for $j = 1..\lceil \log_2 n \rceil$ with $\text{supp}(\mathbf{x}) = \{j : x_j \neq 0\}$ denoting the support of a vector:

$$I^+(j,c) := \{i : j \in \text{supp}(c(k)) \, \forall k \in N(i)\}, \tag{17}$$

$$I^-(j,c) := \{i : j \notin \text{supp}(c(k)) \, \forall k \in N(i)\}. \tag{18}$$

The SOS-2 conditions can now be enforced via the inequalities

$$\sum_{i \in I^+(j,c)} \lambda_i \leq z_j \qquad \forall j = 1..\lceil \log_2 n \rceil, \tag{19}$$

$$\sum_{i \in I^-(j,c)} \lambda_i \leq 1 - z_j \qquad \forall j = 1..\lceil \log_2 n \rceil. \tag{20}$$

The logarithmic formulation of the convex combination model, denoted by (CC_{log}), is then obtained by replacing all inequalities (4)-(5) except the non-negativity constraints on λ_j, $j = 0..n$, by (19)-(20).

As it might not be very intuitive to see that this procedure indeed works properly, we will demonstrate its functioning with the help of an example:

Example 1 (Logarithmic MIP formulation of (CC))**.** *Consider the following domain partition consisting of $n = 5$ intervals and the values of the function $c(\cdot)$ assigned to each of them. Note that we need a 3-bit encoding scheme as $\lceil \log_2 5 \rceil \approx \lceil 2, 32 \rceil = 3$ and that the proposed function values do well comply with the Gray code property.*

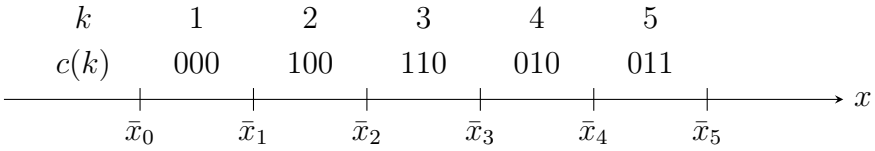


Figure 4: Example of a logarithmic formulation.

We start by explicitely writing down the incidence relation $N(i)$ for all $i = 0..5$ as defined in (16), and, for each $k \in N(i)$, respectively, the support of the vector $c(k)$:

i	$N(i)$	$k \in N(i)$	$c(k)$	$\mathrm{supp}(c(k))$
0	$\{1\}$	1	$(0,0,0)$	$\emptyset$
1	$\{1,2\}$	1	$(0,0,0)$	$\emptyset$
		2	$(1,0,0)$	$\{1\}$
2	$\{2,3\}$	2	$(1,0,0)$	$\{1\}$
		3	$(1,1,0)$	$\{1,2\}$
3	$\{3,4\}$	3	$(1,1,0)$	$\{1,2\}$
		4	$(0,1,0)$	$\{2\}$
4	$\{4,5\}$	4	$(0,1,0)$	$\{2\}$
		5	$(0,1,1)$	$\{2,3\}$
5	$\{5\}$	5	$(0,1,1)$	$\{2,3\}$

Table 1: Incidence relation and support of Gray code vectors.

With all this, we can now construct the index sets $I^+(j,c)$ and $I^-(j,c)$ as in (17) and (18) for all $j = \lceil 1..\log_2 n \rceil$ and, hence, obtain the set of inequalities (19) and (20) representing the SOS-2 conditions:

$$\begin{aligned}
I^+(1,c) &= \{2\} && \rightarrow && \lambda_2 \leq z_1 \\
I^-(1,c) &= \{0,4,5\} && \rightarrow && \lambda_0 + \lambda_4 + \lambda_5 \leq 1 - z_1 \\
I^+(2,c) &= \{3,4,5\} && \rightarrow && \lambda_3 + \lambda_4 + \lambda_5 \leq z_2 \\
I^-(2,c) &= \{0,1\} && \rightarrow && \lambda_0 + \lambda_1 \leq 1 - z_2 \\
I^+(3,c) &= \{5\} && \rightarrow && \lambda_5 \leq z_3 \\
I^-(3,c) &= \{0,1,2,3\} && \rightarrow && \lambda_0 + \lambda_1 + \lambda_2 + \lambda_3 \leq 1 - z_3
\end{aligned}$$

Finally, we can verify that these conditions indeed work as they should by inserting the Gray code values $c(k)$ into the system. Exemplarily, with $(z_1, z_2, z_3) = c(3) = (1,1,0)$ we obtain, as expected,

$$\begin{gathered}
\lambda_2 \leq 1, \quad \lambda_3 \leq 1, \quad \lambda_2 + \lambda_3 \leq 1, \\
\lambda_0 = \lambda_1 = \lambda_4 = \lambda_5 = 0.
\end{gathered}$$

Logarithmic formulation of (MC):

Whereas (CC) was based on the SOS-2 conditions, with (MC) we also have at hand a MIP formulation whose continuous auxiliary variables $\Delta = \{\delta_1..\delta_n\}$ follow the structure of SOS-1. In order to give a logarithmic variant of this model, we keep the idea of Gray encoding but slightly modify the previously defined sets. First, as only one variable δ_i is assigned to each interval, the incidence relation $N(\cdot)$ simplifies to

$$N(i) := \{i\} \qquad i = 1..n, \tag{16*}$$

making itself dispensable for the following considerations. That is, the index sets I^+ and I^- can be reduced to

$$I^+(j,c) := \{i : j \in \operatorname{supp}(c(i))\}, \tag{17*}$$

$$I^-(j,c) := \{i : j \notin \operatorname{supp}(c(i))\}. \tag{18*}$$

With this, we can express the SOS-1 conditions using the inequalities

$$\sum_{i \in I^+(j,c)} \delta_i \leq z_j \qquad \forall j = 1..\lceil \log_2 n \rceil, \tag{19*}$$

$$\sum_{i \in I^-(j,c)} \delta_i \leq 1 - z_j \qquad \forall j = 1..\lceil \log_2 n \rceil. \tag{20*}$$

The logarithmic multiple choice model (MC_{log}) arises by replacing (13)-(14) by (19*)-(20*) and imposing non-negativity constraints on all δ_j, $j = 1..n$.

2.1.5 Continuous Formulations

In all of the formulations mentioned so far, additional binary variables were introduced together with linear constraints in order to model the characteristics of the continuous variables. For example, as we have seen above, some models require the variables to follow the structure of special ordered sets. As an alternative to enforcing the respective combinatorial constraints via binary variables, Tomlin and Beale [31] suggested a constraint programming approach that directly incorporates the SOS conditions into the problem and provides specific branching rules. That is, every time the current LP solution of the branch-and-bound algorithm does not satisfy the SOS requirement, subproblems may be created such that this solution is excluded until, finally, a feasible solution is found. In order to demonstrate the idea, we will now derive continuous SOS-based constraint programming formulations from (CC) and (INC).

Continuous formulation of (CC):

The aggregated version of the convex combination method requires the set of continuous variables $\Lambda = \{\lambda_0..\lambda_n\}$ to be of type SOS-2, which is enforced by the constraints (5). We now eliminate these constraints and all binary variables, and instead state the model that we will call (CC_{SOS}) as follows:

$$x = \sum_{i=0}^{n} \lambda_i \bar{x}_i \tag{21}$$

$$y = \sum_{i=0}^{n} \lambda_i \bar{y}_i \tag{22}$$

$$\sum_{i=0}^{n} \lambda_i = 1 \tag{23}$$

$$\lambda_j \geq 0 \qquad \forall j = 0..n \tag{24}$$

$$\Lambda = \{\lambda_0..\lambda_n\} \qquad \text{is SOS-2} \tag{25}$$

Assuming that we have obtained a solution that does not comply with the SOS-2 conditions at some node of the branch and bound tree, one can create two child nodes by imposing the additional constraints $\sum_{i=0}^{k} \lambda_i = 1$ and $\sum_{i=k}^{n} \lambda_i = 1$, respectively. Thereby, the index k is chosen such that the current solution is made infeasible for both of the resulting subproblems.

Continuous formulation of (INC):

In the incremental model, all but one variable is allowed to take a value other than 0 or 1. Furthermore, if such a variable exists, all preceding variables must be 1 whereas all variables that follow must be set to 0. KEHA ET AL. [23]refer to these requirements on the set $\Delta = \{\delta_1..\delta_n\}$ as SOS-X conditions. By similar means as in the previous case, we obtain the continuous formulation (INC_{SOS}):

$$x = \bar{x}_0 + \sum_{i=1}^{n} (\bar{x}_i - \bar{x}_{i-1})\delta_i \tag{26}$$

$$y = \bar{y}_0 + \sum_{i=1}^{n} (\bar{y}_i - \bar{y}_{i-1})\delta_i \tag{27}$$

$$\delta_j \geq 0 \qquad \forall j = 1..n \tag{28}$$

$$\Delta = \{\delta_1..\delta_n\} \qquad \text{is SOS-X} \tag{29}$$

To illustrate a reasonable branching strategy, we assume that there exists a variable $\delta_j < 1$ and a variable $\delta_k > 0$ for some $k > j$ in the current LP solution, which obviously violates the SOS-X conditions. We can then create two subproblems, adding $\delta_j = 1$ to one of them and $\delta_{j+1} = .. = \delta_n = 0$ to the other.

2.2 Properties of the Models

In this section, we want to study some properties related to the strength of the presented MIP formulations. In order to do so, we first need to give a formal definition of a binary MIP model and apply it to the context of piecewise linear functions:

Definition 5 (Binary MIP model)**.** *A polyhedron $P \subset \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^p \times \mathbb{R}^q$ is called binary MIP model for a set $S \subset \mathbb{R}^d \times \mathbb{R}$, if*

$$(\mathbf{x}, y) \in S \quad \Leftrightarrow \quad \exists\, (\boldsymbol{\mu}, \mathbf{z}) \in \mathbb{R}^p \times \{0,1\}^q \quad \text{s.t.} \quad (\mathbf{x}, y, \boldsymbol{\mu}, \mathbf{z}) \in P. \tag{30}$$

In our case, the set S represents the graph of a function f, i.e., $S := \{(\mathbf{x}, y)\} \subset \mathbb{R}^d \times \mathbb{R}$ for all $\mathbf{x} \in \mathcal{D}$ and $y := f(\mathbf{x})$. As for the analysis of univariate piecewise linear functions with $\mathcal{D} \subset \mathbb{R}$, we get $S = \{(x, y)\} \subset \mathbb{R}^2$. The definition, in other words, states that every point of the function graph must correspond to a feasible point of the MIP model and vice-versa. The continuous auxiliary variables are denoted by $\boldsymbol{\mu} \in \mathbb{R}^p$ whereas the variables $\mathbf{z} \in \mathbb{R}^q$ are binary. Notice that P can also be seen as the LP relaxation of the formulation.

By construction, each of the classical models (CC), (CC*), (INC) and (MC) as well as their logarithmic reformulations comply with the definition of a binary MIP model. We

will begin with studying the strongest property of a MIP formulation with respect to its tightness, called local ideality [27], followed by a study of a weaker property which is referred to as sharpness.

2.2.1 Local Ideality

Definition 6 (Locally ideal MIP formulation)**.** *A MIP formulation is called locally ideal if all extreme points of its LP relaxation satisfy the original integrality constraints.*

By $P_{LP}^{(\cdot)}$ we will denote the feasible region of the LP relaxation of each of the presented non-logarithmic formulations. Thereby, for simplicity, we consider the projections onto the auxiliary variables, since x and y are only used for value assignments and do not occur in any other dependencies. Moreover, instead of aggregating the respective multipliers in sets Λ and Δ, from now we will regard them as vectors $\boldsymbol{\lambda}$ and $\boldsymbol{\delta}$. That is, we obtain

$$
\begin{aligned}
P_{LP}^{(CC)} &= \{(\mathbf{z}, \boldsymbol{\lambda}) \in \mathbb{R}^{2n+1} : (\mathbf{z}, \boldsymbol{\lambda}) \text{ satisfies } (3), (4), (5),\ \mathbf{z} \in [0,1]^n\},\\
P_{LP}^{(CC^*)} &= \{(\mathbf{z}, \boldsymbol{\lambda}) \in \mathbb{R}^{3n} : (\mathbf{z}, \boldsymbol{\lambda}) \text{ satisfies } (3^*),(4^*),(5^*),\ \mathbf{z} \in [0,1]^n\},\\
P_{LP}^{(INC)} &= \{(\mathbf{z}, \boldsymbol{\delta}) \in \mathbb{R}^{2n-1} : (\mathbf{z}, \boldsymbol{\delta}) \text{ satisfies } (9),\ \mathbf{z} \in [0,1]^{n-1}\},\\
P_{LP}^{(MC)} &= \{(\mathbf{z}, \boldsymbol{\delta}) \in \mathbb{R}^{2n} : (\mathbf{z}, \boldsymbol{\delta}) \text{ satisfies } (13), (14),\ \mathbf{z} \in [0,1]^{n-1}\}.
\end{aligned}
$$

With all of the occurring variables being bounded, each of these sets represents a polytope.

In the following, we want to show that (CC*), (INC) and (MC) are locally ideal MIP formulations whereas (CC) is not. Before being able to do so, however, we need to show some preliminary properties based on the notion of total unimodularity.

Definition 7 ((Totally) Unimodular matrix)**.**

- *A square matrix* $\mathbf{M}$ *is unimodular if all its entries are integer and* $\det(\mathbf{M}) = \pm 1$.
- *A matrix* $\mathbf{M}$ *is called totally unimodular if all square non-singular submatrices of* $\mathbf{M}$ *are unimodular.*

In the proofs that follow below, a theorem known as Cramer's Rule will play an important role. As it is a classical result from linear algebra whose proof can be found in many textbooks and which is not of immediate interest here, we will only state it as a lemma.

Lemma 1 (Cramers Rule)**.** *Consider a square matrix* $\mathbf{M} \in \mathbb{R}^{d \times d}$ *having full rank. Then the linear system of equations* $\mathbf{Mv} = \mathbf{b}$ *has a unique solution* $\mathbf{v}^* \in \mathbb{R}^d$ *whose components* v_j^**,* $j = 1..d$*, are given by*

$$v_j^* = \frac{\det(\mathbf{M}^j)}{\det(\mathbf{M})},$$

where $\mathbf{M}^j$ *is the matrix obtained by replacing the* j*-th column in* $\mathbf{M}$ *by the vector* $\mathbf{b}$*.*

With the help of Cramers Rule and using the definition of total unimodularity, we can now derive a very convenient characterization of the extreme points of a polytope defined by a set of linear inequalities:

Lemma 2. *If* $\mathbf{M}$ *is a totally unimodular matrix of dimension* $m \times d$ *and* $\mathbf{b} \in \mathbb{Z}^m$*, then every vertex solution of the system* $\mathbf{Mv} \leq \mathbf{b}$ *is integral.*

Proof. The proof is standard in Linear Algebra. Let $d := \dim(\mathbf{v})$. First of all, we know that every vertex solution $\mathbf{v}^*$ of $\mathbf{Mv} \leq b$ is defined by a set of d linearly independent tight inequalities. Let $\tilde{\mathbf{M}}$ denote the square submatrix of $\mathbf{M}$ corresponding to those inequalities and $\tilde{\mathbf{b}}$ the vector of the respective entries in $\mathbf{b}$, such that it holds $\tilde{\mathbf{M}}\mathbf{v}^* = \tilde{\mathbf{b}}$. As we have assumed $\mathbf{M}$ to be totally unimodular, it follows that $\det(\tilde{\mathbf{M}}) = \pm 1$. With b being an integer vector, applying Cramers Rule finally assures integrality of $\mathbf{v}^*$. □

Lemma 3. *Under the same presumptions as in lemma 2, all vertex solutions of the system* $\{\mathbf{Mv} = \mathbf{b}, \mathbf{v} \geq 0\}$ *are integral.*

Proof. Again, the proof comes from Linear Algebra. We begin with expressing the system equivalently using inequalities only, i.e., as a system $\mathbf{Nv} \leq \mathbf{c}$ with

$$\mathbf{N} = \begin{pmatrix} -\mathbf{I} \\ \mathbf{M} \\ -\mathbf{M} \end{pmatrix} \quad \text{and} \quad \mathbf{c} = \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \\ -\mathbf{b} \end{pmatrix},$$

where $\mathbf{I}$ denotes the identity matrix and $\mathbf{0}$ a zero vector of matching dimension. In analogy to the previous proof, we denote by $\tilde{\mathbf{N}}$ the matrix that consists of all rows in $\mathbf{N}$ corresponding to tight inequalities at some vertex solution of the system. Obviously, $\tilde{\mathbf{N}}$ must contain all rows belonging to $\mathbf{M}$ and $-\mathbf{M}$, plus some rows of $-\mathbf{I}$. Since $\mathbf{M}$ is of rank m, there exists a non-singular square matrix $\tilde{\mathbf{M}}$ of dimension $d \times d$ in $\tilde{\mathbf{N}}$ that fully contains $\mathbf{M}$ and, additionally, a part of $-\mathbf{I}$. From the total unimodularity of $\mathbf{M}$ it follows that $\tilde{\mathbf{M}}$ is unimodular, which, again, yields integrality of all vertices by Cramers Rule. □

In the special case of matrices that only consist of entries 0, +1 or −1, we can find an easy-to-validate sufficient condition for total unimodularity:

Lemma 4. *A matrix* $\mathbf{M}$ *whose entries are exclusively* 0, $+1$ *and* -1 *is totally unimodular if it contains no more than one positive and one negative entry per column.*

Proof. The proof can be found in standard Linear Algebra textbooks. □

With the help of the two previous lemmas, we can now return to showing local ideality of some of the formulations presented above. The ideas for this have been given by Vielma et al. [32] and Padberg [26].

Theorem 1. *The disaggregated convex combination model (CC*), the incremental model (INC) and the multiple choice model (MC) are locally ideal MIP formulations.*

Proof. We restrict ourselves to showing that the result is true for (INC), because the arguments remain the same for the other two formulations.

At first, consider the set of constraints defining $P_{LP}^{(INC)}$ which can be reformulated as a system of linear inequalities of the form $\mathbf{Av} \leq \mathbf{b}$ with

$$\mathbf{v} = \begin{pmatrix} z_1 \\ \vdots \\ z_{n-1} \\ \delta_1 \\ \vdots \\ \delta_n \end{pmatrix}, \quad \mathbf{A} = \left(\begin{array}{cccc|cccc} 0 & \cdots & \cdots & 0 & 1 & 0 & \cdots & \cdots & 0 \\ \hline 1 & 0 & \cdots & 0 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \ddots & \vdots & 0 & -1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & -1 & 0 \\ \hline -1 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & -1 & \ddots & \vdots & \vdots & \ddots & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 0 & \cdots & \cdots & 0 & 1 \\ \hline 0 & \cdots & \cdots & 0 & 0 & \cdots & \cdots & 0 & -1 \end{array}\right), \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}.$$

We observe that the matrix A fulfills the assumptions of Lemma 4, thus it is totally unimodular. Hence, by Lemma 2, every vertex of the LP polytope of (INC) must be integral.

For the proof of (CC*), notice that we can use constraint (3*) to get equality in (5*), i.e., $\lambda_j^l + \lambda_j^u = z_j$ for $j = 1..n$. Then, Lemma 3 applies instead of Lemma 2. □

In contrast to that, it can be shown that the property of local ideality does not hold for the aggregated version of (CC), and, additionally, that (CC) is strictly dominated by (INC) [26]. In order to be able to compare these two models, we will now develop an equivalent formulation for (CC) that uses the same sets of variables as (INC).

In a first step, we use constraints (3) and (4) to eliminate the variables λ_0 and z_n from the model:

$$\lambda_0 = 1 - \sum_{i=1}^{n} \lambda_i$$
$$z_1 = 1 - \sum_{i=2}^{n} z_i$$

Note that with $z_1 \in \{0,1\}$ we immediately get $\sum_{i=2}^{n} z_i \in \{0,1\}$.

Next, we consider the following variable substitutions and their respective inverse mappings:

$$\delta_j := \sum_{i=j}^{n} \lambda_i \qquad j = 1..n \qquad \Longleftrightarrow \qquad \begin{cases} \lambda_j = \delta_j - \delta_{j+1} & j = 1..n-1 \\ \lambda_n = \delta_n \end{cases} \tag{31}$$

$$\tilde{z}_j := \sum_{i=j+1}^{n} z_i \qquad j = 1..n-1 \qquad \Longleftrightarrow \qquad \begin{cases} z_j = \tilde{z}_{j-1} - \tilde{z}_j & j = 2..n-1 \\ z_n = \tilde{z}_{n-1} \end{cases} \tag{32}$$

Obviously, all new variables $\tilde{\mathbf{z}} = (\tilde{z}_1..\tilde{z}_{n-1})$ are restricted to be binary. Furthermore, the original z-variables being binary translates into additional constraints from which, however, all but $\tilde{z}_j \geq \tilde{z}_{j+1}$, $j = 1..n-1$, are redundant.

For simplicity, again, we add fixed artificial variables $\tilde{z}_0 = 1$ and $\tilde{z}_n = 0$. Applying all this to (CC) yields a new formulation which we will denote by (CC'):

$$x = \bar{x}_0 + \sum_{i=1}^{n} (\bar{x}_i - \bar{x}_{i-1})\delta_i \tag{33}$$

$$y = \bar{y}_0 + \sum_{i=1}^{n} (\bar{y}_i - \bar{y}_{i-1})\delta_i \tag{34}$$

$$\begin{aligned} \delta_1 &\leq 1 \\ \delta_j &\geq \delta_{j+1} \qquad j = 1..n-1 \\ \delta_n &\geq 0 \end{aligned} \tag{35}$$

$$\tilde{z}_j \geq \tilde{z}_{j+1} \qquad j = 1..n-2 \tag{36}$$

$$\begin{aligned} \delta_1 &\geq \tilde{z}_1 \\ \delta_j - \delta_{j+1} &\leq \tilde{z}_{j-1} - \tilde{z}_{j+1} \qquad j = 1..n-1 \\ \delta_n &\leq \tilde{z}_{n-1} \end{aligned} \tag{37}$$

$$\tilde{z}_j \in \{0,1\} \qquad j = 1..n-1 \tag{38}$$

All variables being explicitly or implicitly bounded, let $P_{LP}^{(CC')}$ be the LP polytope of the reformulated convex combination model, i.e.,

$$P_{LP}^{(CC')} = \{(\tilde{\mathbf{z}}, \boldsymbol{\delta}) \in \mathbb{R}^{2n-1} : (\tilde{\mathbf{z}}, \boldsymbol{\delta}) \text{ satisfies } (35), (36), (37), \tilde{\mathbf{z}} \in [0,1]^{n-1}\}.$$

The (CC') formulation is, by its construction, equivalent to (CC). However, it is based on the same variables as those used in (INC), making a comparison of both models possible. Therefore, in the following, we will simply denote the set of variables $\tilde{\mathbf{z}}$ by $\mathbf{z}$.

Theorem 2. *$P_{LP}^{(INC)} \subset P_{LP}^{(CC')}$. Furthermore, the convex combination model is not locally ideal.*

Proof. The proof follows the steps presented in [26]. Let $(\mathbf{z}, \boldsymbol{\delta}) \in P_{LP}^{(INC)}$. The set of constraints (9) implies that $1 \geq \delta_1 \geq .. \geq \delta_n \geq 0$, which makes it easy to see that $(\mathbf{z}, \boldsymbol{\delta})$ is also contained in $P_{LP}^{(CC')}$. Hence, $P_{LP}^{(INC)} \subset P_{LP}^{(CC')}$.

Next, let $(\mathbf{z}, \boldsymbol{\delta}) \in P_{LP}^{(CC')}$ such that $\mathbf{z} \in [0,1]^{n-1}$. As the constraints (4*) jointly imply $z_1 \geq .. \geq z_{n-1}$, let k be the highest index for which $z_j = 1$, or, if all $z_j = 0$, set $k := 0$. With this, we can derive from the constraints (3*) that $0 \leq \delta_{k+1} \leq 1$, $\delta_{k+2} = .. = \delta_{n-1} = 0$ and, in the case of $k \geq 1$, $\delta_1 = .. = \delta_k = 1$. The variables $(\mathbf{z}, \boldsymbol{\delta})$ being restricted in this manner fulfill the constraints (9) of the incremental formulation, thus $(\mathbf{z}, \boldsymbol{\delta}) \in P_{LP}^{(INC)}$.

At this stage, we know that $P_{LP}^{(CC')}$ fully contains $P_{LP}^{(INC)}$. Apart from this, it does not include any further points with binary values for $\mathbf{z}$. It remains to show that there are non-integral points that belong to $P_{LP}^{(CC')}$ but not to $P_{LP}^{(INC)}$. Consider therefore $(\mathbf{z}, \boldsymbol{\delta})$ given by $z_1 = z_2 = \frac{1}{2}$, $z_3 = .. = z_{n-1} = 0$, $\delta_1 = \frac{1}{2}$ and $\delta_2 = .. = \delta_n = 0$. It can easily be verified that $(\mathbf{z}, \boldsymbol{\delta}) \in P_{LP}^{(CC')}$ and $(\mathbf{z}, \boldsymbol{\delta}) \notin P_{LP}^{(CC')}$. As $P_{LP}^{(CC')}$ is a polytope, this finally tells us that some of its vertices must have fractional values in $\mathbf{z}$. Hence, (CC') is not locally ideal. □

2.2.2 Sharpness

Definition 8 (Sharp MIP formulations)**.** *A MIP model $P \subset \mathbb{R}^{d+1+p+q}$ of a set $S = (\mathbf{x}, y) \subset \mathbb{R}^{d+1}$ is said to be sharp if $P \mid_{(\mathbf{x},y)} = \text{conv}(S)$, i.e., the projection of the polytope P onto the first $d+1$ variables is exactly the convex hull of S.*

The goal of this section that largely builds upon the paper by Vielma et al. [32] is to show that all of the presented MIP formulations are sharp. For most of them, this follows from the fact that their local ideality implies sharpness. Beyond that, we will find an alternative way to prove that the property even holds for the only non-locally ideal model (CC).

As a first step, we can state the following lemma which is an adapted version of a theorem in the article of Jeroslow and Lowe [20].

Lemma 5. *For any closed set $S \subset \mathbb{R}^{d+1}$ and for any MIP model $P \subset \mathbb{R}^{d+1+p+q}$ for S, the projection of P onto the variables $(\mathbf{x}, y)$ contains the convex hull of S.*

Proof. Let $(\bar{\mathbf{x}}, \bar{y}) \in \text{conv}(S)$. Then it can be expressed as a finite convex combination of points $(\mathbf{x}^{(i)}, y^{(i)}) \in S$, i.e.,

$$\exists\, \boldsymbol{\gamma} \in \mathbb{R}^t,\ t < \infty,\ \sum_{i=1}^{t} \gamma_i = 1 \qquad \text{s.t.} \qquad (\bar{\mathbf{x}}, \bar{y}) = \sum_{i=1}^{t} \gamma_i (\mathbf{x}^{(i)}, y^{(i)}).$$

Since P is a MIP model for S, each of these points $(\mathbf{x}^{(i)}, y^{(i)}) \in S$ corresponds to a point in P. That is, for all $i = 1..t$,

$$\exists\, \boldsymbol{\mu}^{(i)} \in \mathbb{R}^p,\ \mathbf{z}^{(i)} \in \{0,1\}^q \qquad \text{s.t.} \qquad (\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\mu}^{(i)}, \mathbf{z}^{(i)}) \in P.$$

From the convexity of P, using the same multipliers $\boldsymbol{\gamma}$ as above and with $\bar{\boldsymbol{\mu}} = \sum_{i=1}^{t} \boldsymbol{\mu}^{(i)}$ and $\bar{\mathbf{z}} = \sum_{i=1}^{t} \mathbf{z}^{(i)}$, we can conclude that

$$\sum_{i=1}^{t} \gamma_i (\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\mu}^{(i)}, \mathbf{z}^{(i)}) = (\bar{\mathbf{x}}, \bar{y}, \bar{\boldsymbol{\mu}}, \bar{\mathbf{z}}) \in P.$$

Hence, $(\bar{\mathbf{x}}, \bar{y}) \in P\,|_{(\mathbf{x},y)}$. □

With this, in order to show that a MIP formulation is sharp, it remains to prove that the opposite inclusion also holds, i.e., $P\,|_{(\mathbf{x},y)} \subset \text{conv}(S)$ [32].

Theorem 3. *All locally ideal MIP formulations are sharp.*

Proof. Let $P \in \mathbb{R}^{d+1+p+q}$ be the polytope defining the MIP formulation of a set $S = (\mathbf{x}, y) \in \mathbb{R}^{d+1}$ according to definition 5 and let $(\bar{\mathbf{x}}, \bar{y}) \in P\,|_{(\mathbf{x},y)}$. Then, there exist $\bar{\boldsymbol{\mu}} \in \mathbb{R}^p$ and $\bar{\mathbf{z}} \in [0,1]^q$ such that $(\bar{\mathbf{x}}, \bar{y}, \bar{\boldsymbol{\mu}}, \bar{\mathbf{z}}) \in P$.

Since P is locally ideal, all its extreme points fulfil the binary integrality constraints of the $\mathbf{z}$-variables. Assuming that P is bounded and, hence, a polytope, as is the case for all of the presented MIP models, any point of P can be expressed as a convex combination of these extreme points. Hence,

$$\exists\, \boldsymbol{\gamma} \in \mathbb{R}^I_+,\ \sum_{i \in I} \gamma_i = 1 \qquad \text{s.t.} \qquad (\bar{\mathbf{x}}, \bar{y}, \bar{\boldsymbol{\mu}}, \bar{\mathbf{z}}) = \sum_{i \in I} \gamma_i\, (\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\mu}^{(i)}, \mathbf{z}^{(i)}),$$

for $|\, I \,| < \infty$ and with $(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\mu}^{(i)}, \mathbf{z}^{(i)})$, $i \in I$, being the extreme points of P, i.e. $\mathbf{z}^{(i)} \in \{0,1\}^q$. By the definition of a binary MIP model, we know that $(\mathbf{x}^{(i)}, y^{(i)}) \in S$ for all $i \in I$. Thus, $(\bar{\mathbf{x}}, \bar{y}) \in \text{conv}(S)$. □

It remains to show that (CC) is sharp as well, which will not be difficult [32]:

Theorem 4. *The convex combination method yields a sharp MIP model.*

Proof. Let $S := \{(x, y)\} \subset \mathbb{R}^2$ be the graph of a piecewise linear function f,

$$P := \{(x, y, \boldsymbol{\lambda}, \mathbf{z}) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{d+1} \times [0, 1]^d : (1) - (5)\}$$

be the polytope of the LP relaxation of (CC) and let

$$\tilde{P} := \{(x, y, \boldsymbol{\lambda}) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{d+1} : (1) - (3),\ \lambda_j \geq 0\, \forall j = 0..n\}$$

be derived from P by simply dropping the variables $\mathbf{z}$ and all of the dependencies in which they occur. Clearly, it holds that $P\,|_{(x,y,\boldsymbol{\lambda})} \subset \tilde{P}$ and $\tilde{P}\,|_{(x,y)} \subset \text{conv}(S)$. Combined together, the last two statements yield $P\,|_{(x,y)} \subset \text{conv}(S)$. □

2.3 Multivariate Non-Separable Piecewise Linear Functions

Until now we have only considered MIP models for univariate piecewise linear functions which can straightforwardly be extended to cover the class of multivariate additively separable functions. In order to handle general multivariate functions, however, we have to adapt the previously presented models using modelling techniques for functions of arbitrary higher dimension. A broad review on the topic of multivariate piecewise linear functions from which we have taken many ideas can be found in the Phd thesis by Geissler [14].

In definition 1 we have specified the domain $\mathcal{D}$ of a piecewise linear function to be the union of finitely many polytopes. For the models that we are going to derive in the following, however, it is necessary to be slightly more restrictive. That is, let $\mathcal{D}$ now be partitioned into a finite number of simplices $S \in \mathcal{S}$ and let $n :=| \mathcal{S} |$. Thereby, a d-simplex is defined as d-dimensional polytope having exactly $d+1$ vertices. We denote the set of vertices of a simplex S by

$$V(S) := \{\bar{\mathbf{x}}_0^S..\bar{\mathbf{x}}_d^S\} \quad \text{with} \quad \bar{\mathbf{x}}_i^S \in \mathbb{R}^d,$$

and, with m being their total number, the set of all vertices of the partition $\mathcal{S}$ by

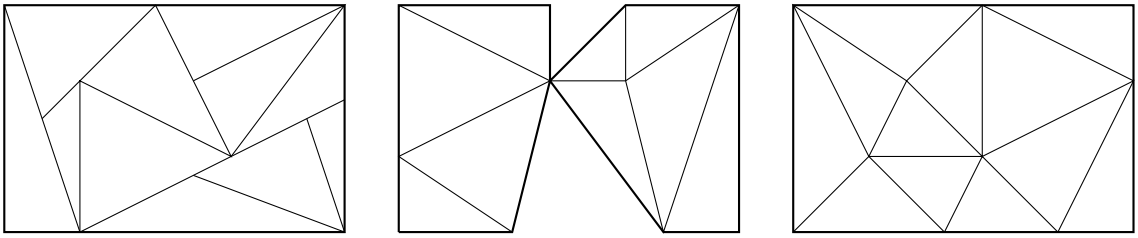
$$V(\mathcal{S}) = \{\bar{\mathbf{x}}_1..\bar{\mathbf{x}}_m\} := \bigcup_{S \in \mathcal{S}} V(S).$$

Furthermore, we assume the simplicial partition $\mathcal{S}$ to be a triangulation, which will be useful for some of the formulations and fundamental for others.

Definition 9 (Triangulation)**.** *A set $\mathcal{S}$ of d-simplices is called triangulation if the following two properties hold:*

1. *For all simplices $S_i, S_j \in \mathcal{S}$, $i \neq j$, the intersection $S_i \cap S_j$ is either a facet of S_i and S_j, a single point or empty.*
2. *For all subsets $\emptyset \neq \mathcal{T} \subset \mathcal{S}$ there exist at least one simplex $T \in \mathcal{T}$ and one simplex $S \in \mathcal{S} \setminus \mathcal{T}$ such that S and T share d common vertices.*

Notice that if $\mathcal{S}$ is a partition of a convex compact domain $\mathcal{D}$, the second condition is automatically fulfilled. Examples of valid and non-valid triangulations in two dimensions are given in 5.



(a) Violation of 1. **(b)** Violation of 2. **(c)** Valid triangulation.

Figure 5: Examples of 2-dimensional triangular partitions. Figures (a) and (b) violate the first and the second condition of definition 9, respectively, whereas (c) shows a valid triangulation.

With these preparations, we are now ready to state the generalized MIP models for the class of multivariate non-separable piecewise linear functions. Thereby, we will proceed in the same order as in section 2.

2.3.1 Generalized Convex Combination Method

In analogy to the univariate case, this method is based on the idea that, for a d-simplex S, every point $\mathbf{x} \in S$ can be written as a convex combination of the $d+1$ vertices of S. Again, we will distinguish between an aggregated and a disaggregated version of the convex combination method. In the aggregated case, exactly one multiplier $\lambda_j \in \Lambda$ is introduced at each vertex $\bar{\mathbf{x}}_j \in v(\mathcal{S})$, $j = 1..m$. By contrast, the disaggregated version uses separate variables for each simplex, i.e., a set of variables $\Lambda^* := \{\Lambda^*_S\}_{S \in \mathcal{S}}$ with $\Lambda^*_S := \{\lambda^S_0..\lambda^S_d\}$ for all $S \in \mathcal{S}$. In addition to that and for both variants alike, each simplex $S \in \mathcal{S}$ is assigned a binary variable z_S.

Beginning with the aggregated convex combination method, we arrive at the following MIP formulation (GCC):

$$\mathbf{x} = \sum_{i=1}^{m} \lambda_i \bar{\mathbf{x}}_i \tag{39}$$

$$y = \sum_{i=1}^{m} \lambda_i \bar{y}_i \tag{40}$$

$$\sum_{i=1}^{m} \lambda_i = 1 \tag{41}$$

$$\sum_{S \in \mathcal{S}} z_S = 1 \tag{42}$$

$$0 \leq \lambda_j \leq \sum_{\{S : \bar{\mathbf{x}}_j \in V(S)\}} z_S \qquad j = 1..m \tag{43}$$

$$z_S \in \{0, 1\} \qquad S \in \mathcal{S} \tag{44}$$

The disaggregated version, denoted by (GCC*), can be formulated as:

$$\mathbf{x} = \sum_{S \in \mathcal{S}} \sum_{v \in V(S)} \lambda_v^S \bar{\mathbf{x}}_v^S \tag{39*}$$

$$y = \sum_{S \in \mathcal{S}} \sum_{v \in V(S)} \lambda_v^S \bar{y}_v^S \tag{40*}$$

$$\sum_{S \in \mathcal{S}} \sum_{v \in V(S)} \lambda_v^S = 1 \tag{41*}$$

$$\sum_{S \in \mathcal{S}} z_S = 1 \tag{42*}$$

$$\sum_{v \in V(S)} \lambda_v^S \leq z_S \qquad S \in \mathcal{S} \tag{43a*}$$

$$\lambda_v^S \geq 0 \qquad \begin{matrix} S \in \mathcal{S} \\ v \in V(S) \end{matrix} \tag{43b*}$$

$$z_S \in \{0, 1\} \qquad S \in \mathcal{S} \tag{44*}$$

Notice that no ordering of the simplices is required for these formulations. As we will see, this will not be the case for the generalizations of (INC) as well as (MC).

2.3.2 Generalized Incremental Model

In the incremental model, we use the fact that every point $\mathbf{x}$ of a simplex S can be expressed as

$$\mathbf{x} = \bar{\mathbf{x}}_0^S + \sum_{j=1}^{d} (\bar{\mathbf{x}}_j^S - \bar{\mathbf{x}}_0^S)\delta_j^S$$

with non-negative multipliers $\Delta^S := \{\delta_1^S..\delta_d^S\}$ that are restricted by $\sum_{i=1}^{d} \delta_i^S \leq 1$. See figure 6 for an illustration.

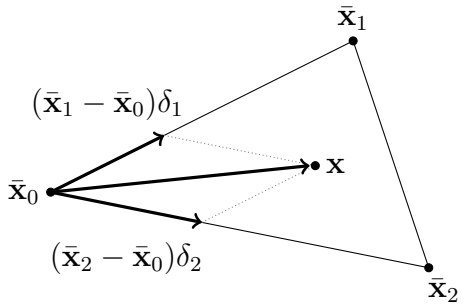


Figure 6: δ-representation of an interior simplex point.

As another main characteristic feature of the incremental model, the so-called filling conditions relied on an ordering of the underlying simplices. This ordering was trivially induced in the one-dimensional case by labelling the subintervals according to their position, whereas for general dimensionality additional considerations become necessary. In order to formulate a generalization of (INC) we need the triangulation to fulfill two properties:

- The set of simplices $\mathcal{S} = \{S_1..S_n\}$ is ordered such that $S_i \cap S_j \neq \emptyset$ for all $i \neq j$.
- The vertices $\bar{\mathbf{x}}_0^{S_j}..\bar{\mathbf{x}}_d^{S_j}$ of each simplex S_j, $j = 1..n-1$, can be labelled such that $\bar{\mathbf{x}}_d^{S_i} = \bar{\mathbf{x}}_0^{S_{i+1}}$, i.e., the first vertex of every simplex corresponds to the last vertex of the previous one.

We assume at this point that we have such a triangulation at hand. In fact, it can be shown that, given an arbitrary triangulation, an ordering fulfilling the required properties can be computed in $\mathcal{O}(n^2 + nd^2)$ time [14].

With this, it is possible to reach the first vertex $\bar{\mathbf{x}}_0^{S_j}$ of any simplex S_j via a path along the edges of the simplices preceding S_j. That is, we can write $\bar{\mathbf{x}}_0^{S_j}$ as

$$\bar{\mathbf{x}}_0^{S_j} = \bar{\mathbf{x}}_0^{S_1} + \sum_{i=1}^{n} (\bar{\mathbf{x}}_d^{S_i} - \bar{\mathbf{x}}_0^{S_k}).$$

Combining this with the representation of a point inside a single simplex given above, we can now state the generalized incremental model that will be denoted by (GINC). Besides the continuous variables $\Delta = \{\Delta^{S_1}..\Delta^{S_n}\}$ with $\Delta^{S_j} = \{\delta_1^{S_j}..\delta_n^{S_j}\}$, one binary variable z_j is used in correspondence to each simplex S_j, $j = 1..n-1$.

$$\mathbf{x} = \bar{\mathbf{x}}_0^{S_1} + \sum_{i=1}^{n}\sum_{k=1}^{d}(\bar{\mathbf{x}}_k^{S_i} - \bar{\mathbf{x}}_0^{S_i})\delta_k^{S_i} \tag{45}$$

$$y = \bar{y}_0^{S_1} + \sum_{i=1}^{n}\sum_{k=1}^{d}(\bar{y}_k^{S_i} - \bar{y}_0^{S_i})\delta_k^{S_i} \tag{46}$$

$$\sum_{k=1}^{d}\delta_k^{S_1} \leq 1 \tag{47}$$

$$\delta_d^{S_j} \geq z_j \geq \sum_{k=1}^{d}\delta_k^{S_{j+1}} \qquad j = 1..n-1 \tag{48}$$

$$\delta_k^{S_j} \geq 0 \qquad \begin{matrix} j = 1..n \\ k = 1..d \end{matrix} \tag{49}$$

$$z_j \in \{0,1\} \qquad j = 1..n-1 \tag{50}$$

Constraints (47)-(49) ensure that if $z_j = 1$ for any j, then the point $\mathbf{x}$ cannot lie inside of simplex S_j. Instead, since $\delta_d^{S_j}$ is forced to be equal to 1, the path leads directly from $\bar{\mathbf{x}}_0^{S_j}$ to $\bar{\mathbf{x}}_d^{S_j} = \bar{\mathbf{x}}_0^{S_{j+1}}$, i.e., to the first vertex of the next simplex. Therefore, we call these constraints generalized filling conditions. Furthermore, the interior of a simplex S_j can only be reached if $z_{j-1} = 1$ and $z_j = 0$. Figure 7 illustrates the idea on which the generalized incremental model relies.

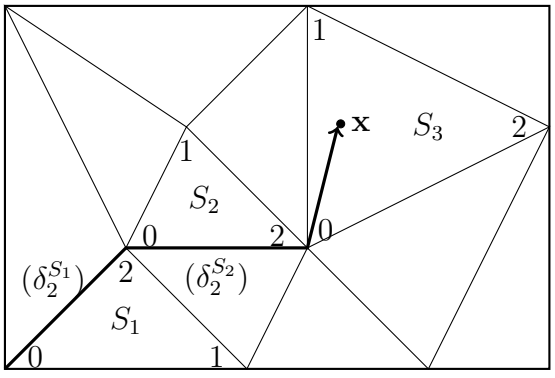


Figure 7: The Generalized Incremental Method. For simplicity, the figure only shows the relevant information. The remaining part of the given triangulation can easily be labelled in such a way that the two requirements on the ordering scheme are met.

2.3.3 Generalized Multiple Choice Model

The generalized multiple choice model (GMC) can be derived from (GINC) by jumping directly to the simplex that contains $\mathbf{x}$ instead of following a path along the edges of the preceding simplices. Hence, this procedure in itself does not require an explicit ordering of the simplices and could be sufficiently described by choosing a global starting point $\bar{\mathsf{x}}_0$ and labelling the vertices of each simplex, but for simplicity we will keep the notation used in (GINC).

$$\bar{\mathbf{x}} = \bar{\mathbf{x}}_0^{S_1} + \sum_{i=1}^{n}[(\bar{\mathbf{x}}_0^{S_i} - \bar{\mathbf{x}}_0^{S_1})z_i + \sum_{k=1}^{d}(\bar{\mathbf{x}}_k^{S_i} - \bar{\mathbf{x}}_0^{S_i})\delta_k^{S_i}] \tag{51}$$

$$y = \bar{y}_0^{S_1} + \sum_{i=1}^{n}[(\bar{y}_0^{S_i} - \bar{y}_0^{S_1})z_i + \sum_{k=1}^{d}(\bar{y}_k^{S_i} - \bar{y}_0^{S_i})\delta_k^{S_i}] \tag{52}$$

$$\sum_{k=1}^{d}\delta_k^{S_j} \leq z_j \qquad j = 1..n \tag{53}$$

$$\delta_k^{S_j} \geq 0 \qquad \begin{matrix} j = 1..n \\ k = 1..d \end{matrix} \tag{54}$$

$$z_j \in \{0,1\} \qquad j = 1..n \tag{55}$$

Constraints (53) couple the δ-variables of each simplex S_j to its respective binary variable z_j, only allowing the point $\mathbf{x}$ to be inside S_j if $z_j = 1$. This idea, which is illustrated in 8, is closely related to the disaggregated convex combination method with the only difference being the representation of a point $\mathbf{x}$.

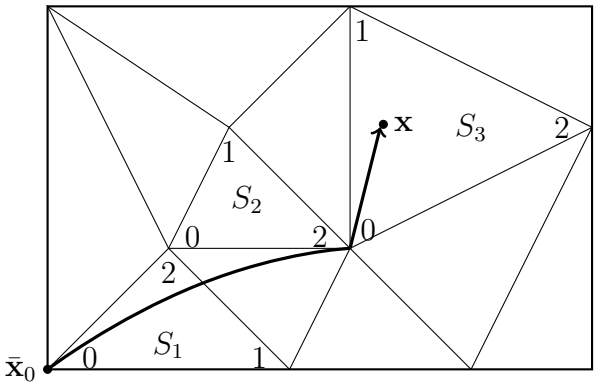


Figure 8: The Generalized Multiple Choice Method.

2.3.4 Generalized Logarithmic Models

As with the MIP models for univariate functions, it is also possible to find formulations having a logarithmic number of binary variables in the multivariate case. This can be achieved by exploiting special structures of the constraint sets of those models. That is, using triangulations with a number of n simplices, all models restrict the values of their variables to conform to one of n different classes, depending on which simplex the point $\mathbf{x}$ lies in. As we have already seen, n states can be encoded using $\lceil \log_2 n \rceil$ binary variables.

Logarithmic formulation of (GCC*):

In the disaggregated version of the generalized convex combination method, exactly one binary variable is allowed to be 1 while all others are 0. Thereby, it is determined which simplex contains $\mathbf{x}$. In the following, we will then speak of this simplex as being active.

The idea behind converting a conventional formulation into a logarithmic one is to introduce an injective function $c : \mathcal{S} \to \{0,1\}^{\lceil \log_2 n \rceil}$. I.e., every simplex is assigned a binary code having enough digits to let it be unique over the whole set $\mathcal{S}$. If, as in the case of (GCC*), there are no other requirements like, e.g., precedence constraints on the simplices and their variables, we need no further restrictions on the function c. An example is shown in figure 9.

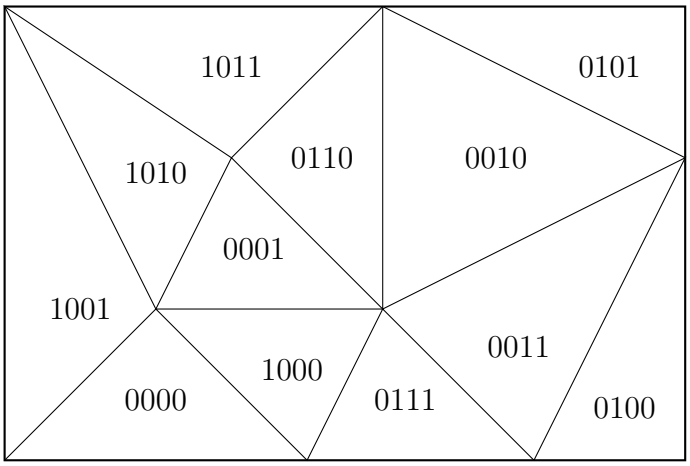


Figure 9: Binary encoding of $n = 12$ simplices in a triangulation by a binary encoding scheme using $\lceil \log_2 12 \rceil = 4$ bits.

According to Vielma and Nemhauser [33], the logarithmic model which we will denote by gCC^*_{log} can be stated as follows:

$$\mathbf{x} = \sum_{S \in \mathcal{S}} \sum_{i=0}^{d} \lambda_i^S \bar{\mathbf{x}}_i^S \tag{56}$$

$$y = \sum_{S \in \mathcal{S}} \sum_{i=0}^{d} \lambda_i^S \bar{y}_i^S \tag{57}$$

$$\sum_{S \in \mathcal{S}} \sum_{i=0}^{d} \lambda_i^S = 1 \tag{58}$$

$$\sum_{S \in \mathcal{S}} \sum_{i=0}^{d} c(S)_j \lambda_i^S \leq z_j \qquad j = 1 \,..\, \lceil \log_2 n \rceil \tag{59}$$

$$\sum_{S \in \mathcal{S}} \sum_{i=0}^{d} (1 - c(S)_j) \lambda_i^S \leq 1 - z_j \qquad j = 1 \,..\, \lceil \log_2 n \rceil \tag{60}$$

$$\lambda_j^S \geq 0 \qquad \begin{matrix} S \in \mathcal{S} \\ j = 0 \,..\, d \end{matrix} \tag{61}$$

$$z_j \in \{0, 1\} \qquad j = 1 \,..\, \lceil \log_2 n \rceil \tag{62}$$

By $c(S)_j$, we mean the j-th component of the function value of c at simplex S. The constraints (59) and (60) guarantee that exactly the simplex whose binary encoding corresponds to the values of $(z_1..z_{\lceil \log_2 n \rceil})$ is made active. This can be seen from the fact that, for every $j = 1 \,..\, \lceil \log_2 n \rceil$, only those λ-variables belonging to simplices whose binary code at bit j equals z_j are allowed to be positive.

2.4 Extension to Lower-Semicontinuous Functions

The formulations introduced until now are all based on the function f being continuous over a compact domain $\mathcal{D}$. However, it is not difficult to extend the underlying ideas to semicontinuous functions. This section is supposed to have a supplementary character only, since we are not going to concentrate further on issues of semicontinuity in the remainder of this thesis. Moreover, we will focus on the class of lower-semicontinuous functions. These are often used in the context of minimization problems in order to make sure that the optimum is attained. See figure 10 for an example of such a function. Analogously, for maximization problems one would prefer to consider upper-semicontinuous functions.

Definition 10 (Lower-semicontinuous function)**.** *A real-valued function f is called lower-semicontinuous at a point $\bar{\mathbf{x}} \in \mathcal{D}$ if, for any small positive number ϵ, $f(\mathbf{x}) > f(\bar{\mathbf{x}}) - \epsilon$ for all $\mathbf{x}$ in some neighborhood of $\bar{\mathbf{x}}$.*

In general, the set of points representing the graph of a semicontinuous function is not closed. Therefore, it can not be modelled as a binary MIP by a polyhedron P according to definition 5. In order to cure this problem, we will switch to modelling the epigraph of the function.

Definition 11 (Epigraph)**.** *The epigraph of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the set of points lying on or above its graph, i.e.,*

$$\text{epi}(f) := \{(\mathbf{x}, y) \mid \mathbf{x} \in \mathbb{R}^n,\ y \in \mathbb{R},\ y \geq f(\mathbf{x}\} \subset \mathbb{R}^{n+1}.$$

Then, the minimization of f on a domain $\mathcal{D}$ is equivalent to minimizing y under the condition $(\mathbf{x}, y) \in \text{epi}(f)$. Figure 10 shows an example of a univariate lower-semicontinuous piecewise linear function and its epigraph.

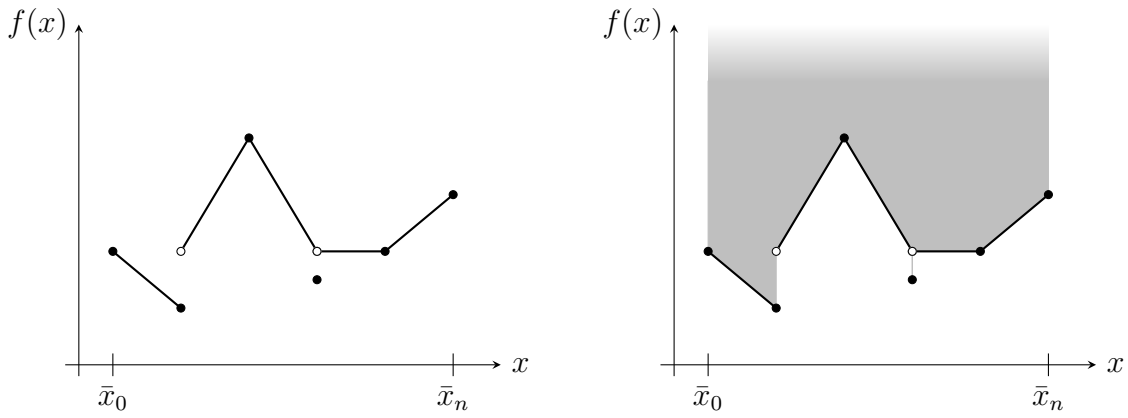


Figure 10: A lower-semicontinuous piecewise linear function and its epigraph.

The main difficulty we face when considering semicontinuous functions is that they are no longer affine on closed subsets of the domain like polytopes or simplices. For example, univariate functions may now only be affine on open or semi-open intervals such as $[\bar{x}_i, \bar{x}_{i+1})$ or $(\bar{x}_i, \bar{x}_{i+1})$, or even on discrete points $\{\bar{x}_i\}$. A simple and natural extension is to work with so-called copolytopes that are, according to Kannan [21], defined by finitely many strict and non-strict inequalities. More formally, a set of points C is referred to as copolytope if it can be written as

$$C = \{\mathbf{x} \in \mathbb{R}^n \mid A_1\mathbf{x} \leq b_1,\ A_2\mathbf{x} < b_2\},$$

where A_1, A_2 are coefficient matrices and b_1, b_2 are vectors of appropriate dimension. We obtain the closure $\bar{C}$ of C by using $A_2\mathbf{x} \leq b_2$ instead of $A_2\mathbf{x} < b_2$. By $V(\bar{C})$, we denote the set of vertices of $\bar{C}$. Note that a copolytope does not necessarily need to be of full dimension d, and, hence, it is possible that $|\, V(\bar{C})\, | \leq d$.

Let $\mathcal{C}$ denote the set of copolytopes on which f is affine, representing a partition of its domain, i.e., $\mathcal{D} = \bigcup_{C \in \mathcal{C}}$. With this, we slightly modify definition 1 such that a semicontinuous function is called piecewise linear if it can be written as

$$f(\mathbf{x}) := \{m_C \mathbf{x} + n_C\}_{C \in \mathcal{C}}$$

with $m_C \in \mathbb{R}^d$ and $n_C \in \mathbb{R}$ for all $C \in \mathcal{C}$.

Working with copolytopes instead of simplices finally allows to modify the standard models such that they become valid for lower-semicontinuous functions in a straightforward manner. Details can be found, for example, in [32].

3 Piecewise Linear Optimization

Until now, we have studied various possibilities of modelling piecewise linear functions without explicitly focussing on issues of optimization. In particular, we have neither set an objective function nor taken into account any constraints on the feasible region other than those directly imposed by the formulation of the piecewise linear function. The largest part of this section will be dedicated to a constraint programming approach for a certain class of univariate piecewise linear constraints, paving the way towards our implementation of a constraint handler in SCIP. Thereafter, we briefly want to point out some issues that would occur if multivariate functions were considered in the same context, which may be taken as an inspiration for further work. At the beginning, however, we want to give a brief overview of the topic of piecewise linear objective functions.

3.1 Piecewise Linear Objective Functions

Using the notation of section 2, let $f : \mathcal{D} \to \mathbb{R}$ be a continuous piecewise linear function on a compact domain $\mathcal{D} \subset \mathbb{R}^d$ and $S := \{(\mathbf{x}, y) \mid \mathbf{x} \in \mathcal{D},\, y = f(\mathbf{x})\}$ the set representing the graph of f. Furthermore, let $P \subset \mathbb{R}^{d+1+p+q}$ be a sharp MIP model for S according to definition 8.

As a first step, we assume that f is to be minimized without any other constraints being present. Then, the following holds [32]:

$$\min_{\mathbf{x} \in D} f(\mathbf{x}) = \min\{y \mid (\mathbf{x}, y) \in S\} \tag{63}$$

$$= \min\{y \mid (\mathbf{x}, y) \in \operatorname{conv}(S)\} \tag{64}$$

$$= \min\{y \mid (\mathbf{x}, y, \boldsymbol{\mu}, \mathbf{z}) \in P\} \tag{65}$$

Hence, under these presumptions, it is sufficient to simply solve the LP relaxation of an arbitrary sharp formulation to obtain the solution to the original MIP. Notice that S being a closed set is required in equality (64) and the sharpness of P is needed for step (65).

If, by contrast, additional constraints are considered, the property of sharpness will no longer be preserved on the restricted domain $X \subset \mathcal{D}$. Provided that X is a convex set, however, the sharpness of a MIP model on the whole domain $\mathcal{D}$ at least helps to obtain a characterization of the LP bound [32]:

$$\min_{\mathbf{x}\in X} f(\mathbf{x}) = \min\{y \mid (\mathbf{x}, y) \in S \cap (X \times \mathbb{R})\} \tag{66}$$

$$\geq \min\{y \mid (\mathbf{x}, y) \in \operatorname{conv}(S) \cap (X \times \mathbb{R})\} \tag{67}$$

$$= \min\{y \mid (\mathbf{x}, y, \boldsymbol{\mu}, \mathbf{z}) \in P,\ \mathbf{x} \in X\} \tag{68}$$

$$= \min_{\mathbf{x}\in X} \operatorname{convenv}_{\mathcal{D}} f(\mathbf{x}) \tag{69}$$

Here, (69) follows from the definition 12 of convex envelopes given below in combination with the sharpness of P. Figure 11 shows an example of a function to be minimized on a restricted domain that illustrates this estimate. For restricted domains, as we can see, it can not be ensured that the minimum of the convex envelope lies on the function graph.

Definition 12 (Convex underestimator and convex envelope)**.** *A convex underestimator of a function $f : \mathcal{D} \to \mathbb{R}$ is a convex function c such that $f(\mathbf{x}) \geq c(\mathbf{x})$ for all $\mathbf{x} \in \mathcal{D}$. The largest convex underestimator of f over $\mathcal{D}$ is called convex envelope of f, denoted by* $\operatorname{convenv}_{\mathcal{D}} f(\mathbf{x})$.

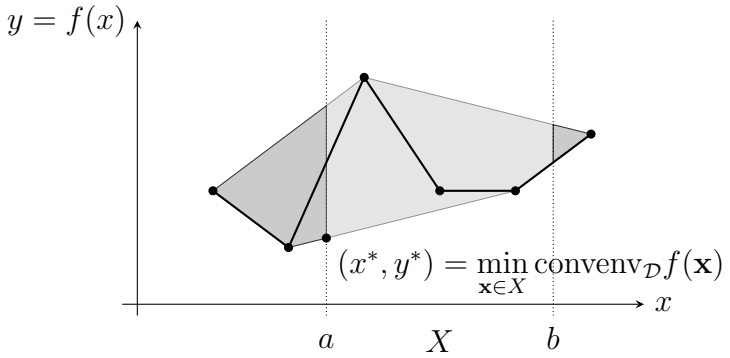


Figure 11: Lower bound for minimization on a restricted domain $X = [a, b]$. The convex envelope is equal to the lower boundary of the convex hull of the function.

In the following, we want to study a special case of such problems that is of particular practical relevance. With the function f being continuous and additively separable and all constraints being linear inequalities, it is referred to as separable piecewise linear optimization problem (SPLOP):

$$\begin{aligned} \min f(\mathbf{x}) &= \sum_{i=1}^{d} f_i(x_i) \\ \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ x_j &\in [l_j, u_j] \qquad\qquad j = 1..d \end{aligned}$$

For each of the univariate functions f_i, $i = 1..d$, let n_i denote the number of its affine segments. If all f_i, $i = 1..n$, are convex, then (SPLOP) can be solved in polynomial time. This is because, under these circumstances and regarding the affine segments of each f_i as parts of functions $f_i^{(k)}$, $k = 1..n_i$, we can establish the equivalence

$$\min \sum_{i=1}^{d} f_i(x_i) \qquad \Leftrightarrow \qquad \begin{cases} \min \sum_{i=1}^{d} y_i & \\ y_j \geq f_j^{(k)}(x_j) & \begin{matrix} j = 1..d \\ k = 1..n_j \end{matrix} \end{cases}.$$

Hence, the convex (SPLOP) can be formulated as a linear program and solved, e.g., by a polynomial interior point method. By contrast, as shown by Keha et al. [22], only one f_i being non-convex makes (SPLOP) NP-hard.

3.2 Univariate Piecewise Linear Constraints

Besides being the objective to be minimized or maximized, piecewise linear functions often occur as constraints of optimization problems and special approaches have been developed to address this topic. In this part, we are going to study a type of constraint that involves a univariate piecewise linear function:

Definition 13 ((Univariate) Piecewise Linear Constraint)**.** *A constraint in the variables $x \in \mathbb{R}$ and $y \in \mathbb{R}$ is called piecewise linear if it is of the form*

$$g(x) \leq ay + b \qquad \text{or} \qquad g(x) = ay + b,$$

where $g : \mathcal{D} \to \mathbb{R}$ with $\mathcal{D} \subset \mathbb{R}$ is a piecewise linear function defined by a finite number of discrete support points $\{(\bar{x}_i, \bar{y}_i)\}_{i=0}^{n} \subset \mathbb{R}^2$, and $a > 0$, $b \in \mathbb{R}$ are parameters.

Notice that a constraint of the type $g(x) \geq ay + b$ can be modelled via the substitutions $y' := -y$ and $g' := -g$ as $g'(x) \leq ay' - b$. The feasible region of a piecewise linear inequality constraint can be described as a polyhedron in the (x, y)-space bounded on one side by an affine transformation of g. In the equality case, the feasible region follows exactly the graph of the piecewise linear function. In both cases, in general, the feasible region is a non-convex set.

By applying one of the approaches presented in chapter 2, piecewise linear constraints can easily be incorporated into a mixed-integer program. We will later use such formulations for the purpose of benchmarking. In this section, however, we are going to study a way of treating piecewise linear constraints that has been described by Hooker [18], based on the idea of working with a convex hull relaxation which avoids the necessity of additional variables.

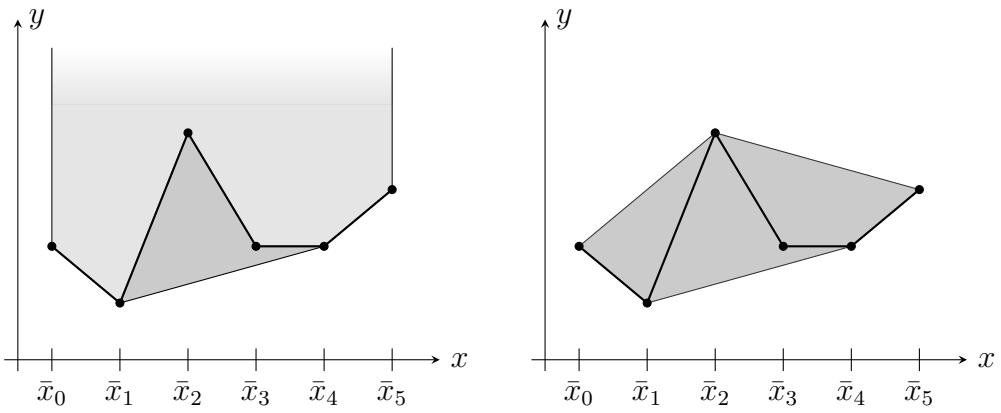
3.2.1 Convex Hull Relaxation

By replacing the graph of the function $\frac{g(x)-b}{a}$ with its (partial) convex hull, valid relaxations for the constraints of the above type can be stated as follows:

$$g(x) \leq ay + b \quad \Rightarrow \quad (x, y) \in \operatorname{epi} \operatorname{convenv}_{\mathcal{D}} \left(\frac{g(x) - b}{a} \right), \tag{70}$$

$$g(x) = ay + b \quad \Rightarrow \quad (x, y) \in \operatorname{conv} \left(\{(\bar{x}_i, \bar{y}_i)\}_{i=0}^{n} \right). \tag{71}$$

Figure 12 illustrates the two cases.



(a) $g(x) \leq ay + b$ **(b)** $g(x) = ay + b$

Figure 12: Piecewise linear constraints. In figure (a), an inequality constraint is shown, whose feasible region is represented by the area shaded in light gray, whereas its relaxation additionally comprises the dark gray part. For the equality constraint in figure (b), the feasible region is given by its function graph and the convex hull relaxation by the area shaded in dark gray.

As we will see in detail in section 4.3, the convex hull of a two-dimensional set of p points can be computed in $\mathcal{O}(p \log p)$ or, by some output-sensitive method, even in $\mathcal{O}(p \log h)$ time with h being the number of convex hull vertices.

3.2.2 Branching Methods

Using the above LP relaxation, we want to consider the LP solution $(\hat{x}, \hat{y})$ of the optimization problem for the constraint variables that has been obtained at some node in the process of a branch-and-bound algorithm. Assuming that $(\hat{x}, \hat{y})$ violates the original constraint, one way of resolving the infeasibility is branching, i.e., creating subproblems with tighter LP relaxations.

There are several possibilities of how branching can be performed and we want to refer to these as branching methods. First of all, if $\hat{x}$ is identical to some $\bar{x}_k$ belonging to the set of support points, a very intuitive strategy would be to create two child nodes by imposing $x \leq \bar{x}_k$ and $x \geq \bar{x}_k$, respectively. By contrast, if $\hat{x}$ lies inside an interval $(\bar{x}_k, \bar{x}_{k+1})$, the selection of a branching point is not equally obvious such that we have different options to choose from. Throughout the remaining part of this thesis, the following variants will be referenced by the name given in typewriter font:

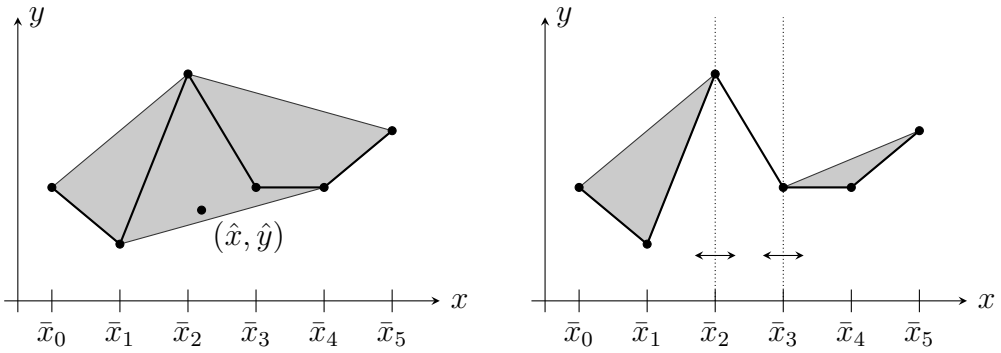
- `2-CHILD-LEFT`: Given a solution value $\hat{x} \in (\bar{x}_k, \bar{x}_{k+1})$, create two child nodes with $x \leq \bar{x}_k$ and $x \geq \bar{x}_k$. As an exception, if $\hat{x} \in (\bar{x}_0, \bar{x}_1)$, choose $\bar{x}_1$ as branching point.
- `2-CHILD-RIGHT`: Similarly, create two child nodes with $x \leq \bar{x}_{k+1}$ and $x \geq \bar{x}_{k+1}$. However, if $\hat{x} \in (\bar{x}_{n-1}, \bar{x}_n)$, choose $\bar{x}_{n-1}$ as branching point.
- `2-CHILD-EXACT`: Add $(\hat{x}, g(\hat{x}))$ to the set of support points of g and create two child nodes with $x \leq \hat{x}$ and $x \geq \hat{x}$. However, if $\hat{x} \in (\bar{x}_0, \bar{x}_1)$ or $\hat{x} \in (\bar{x}_{n-1}, \bar{x}_n)$, better relaxations can be achieved by choosing $\bar{x}_1$ or $\bar{x}_{n-1}$, respectively, as branching point.
- `3-CHILD`: If $k \geq 1$ and $k \leq n-2$, create three child nodes with $x \leq \bar{x}_k$, $x \in [\bar{x}_k, \bar{x}_{k+1}]$ and $x \geq \bar{x}_{k+1}$. Otherwise, i.e., if $\hat{x}$ lies in the left- or rightmost interval, create only two child nodes with $\bar{x}_1$ or $\bar{x}_{n-1}$ being the branching points.

Among the variants that generate two child nodes in each branching decision, only `2-CHILD-EXACT` ensures that the current LP solution is excluded from both subproblems. However, this property is potentially paid for by a successive increase in the number of support points. This can partly be avoided by a combination of the presented methods that, in turn, requires some more computational effort:

- `2-CHILD-FLEX`: Determine whether the current LP solution would be excluded from both resulting subproblems if either `2-CHILD-LEFT` or `2-CHILD-RIGHT` was applied. If this is not the case, perform branching according to `2-CHILD-EXACT`.

Strategy `3-CHILD` automatically guarantees that neither of the generated subproblems contains $(\hat{x}, \hat{y})$. Additionally, it even leads to an exact representation of the constraint over the interval that the solution belongs to. This is illustrated by figure 13.

In addition to branching on the $\mathbf{x}$-variable of a constraint, one might have the idea of taking into account y as branching candidate as well. This, however, besides being a somewhat unintuitive approach, may cause the domain of x in the resulting subproblems to have holes. In order to avoid such complications of the solving process, we refrain from further investigation into this direction.



(a) Convex hull relaxation **(b)** Branching example

Figure 13: Example of `3-CHILD` branching on a piecewise linear equality constraint $g(x) = y$. Since $\hat{x}$ lies between $\bar{x}_2$ and $\bar{x}_3$, the new branches have been created by imposing $x \leq \bar{x}_2$, $x \in [\bar{x}_2, \bar{x}_3]$ and $x \geq \bar{x}_3$, respectively.

3.2.3 Branching Candidate Selection

Another important aspect in the context of branching is the selection of the most suitable branching variable in case that there are several options. Since the solving process may be considerably influenced by the choice of branching variables, the topic of employing a good branching strategy is of great importance. For mixed-integer programs, this question has been intensively studied and a variety of fundamental approaches have evolved. A good summary on these can be found, for example, in [1, 2]. Most of the approaches rely on a quality measure known as score value that allows for ranking possible branching decisions, while differing in the way this score is computed.

In the following, we will present some of these strategies as described in [1] and show how the underlying ideas can be transferred to the continuous variables in piecewise linear constraints.

Most infeasible branching

A very intuitive but, unfortunately, not particularly effective strategy for branching on binary variables is to choose the one that is farthest from integrality. This is motivated by the hope of eliminating as much integrality violation as possible. The score of a branching candidate x_j is computed as

$$s_j^{int} := \min\{\hat{x}_j - \lfloor \hat{x}_j \rfloor, \lceil \hat{x}_j \rceil - \hat{x}_j\}.$$

Computational experiments have shown that most infeasible branching does not perform significantly better than a random selection of branching variables [2]. The same holds true for least infeasible branching in which the variable that is closest to an integral value is selected as branching candidate.

As an analogue procedure for piecewise linear constraints, a distance measure of constraint violation could be designed as follows: First, let d_j^{cons} be the euclidean distance from the constraint variables' LP solution $(\hat{x}_j, \hat{y}_j)$ to the feasible region of the constraint. Similarly, let d_j^{rel} be the distance from $(\hat{x}_j, \hat{y}_j)$ to the boundary of the feasible region of the LP relaxation. With this, a score value in $[0, 1]$ for branching on x_j can be obtained by

$$s_j^{pwl} := \frac{d_j^{cons}}{d_j^{cons} + d_j^{rel}}.$$

Since $s^{int} \in [0, \frac{1}{2}]$, it is advisable to consider $s^{pwl}/2$ instead of s^{pwl} in order to compare scores when branching candidates in a problem can be both integer or piecewise linear constraint variables.

Strong branching

Most and least infeasible branching do not take into account any information related to the objective function of the problem. However, with the goal of keeping the branching tree as small as possible in mind, it makes sense to consider the impact of possible branchings on the dual objective bound. Therefore, the idea of strong branching, as first mentioned by Benichou et al [7], is to tentatively solve the subproblems that would arise if branching was performed on some branching candidate. Then, the most promising variable in the sense of dual bound progress is chosen for the actual branching. Since we are later going to apply strong branching to piecewise linear constraints and also use it as part of another branching strategy, we will go a bit into detail at this place.

In the context of mixed-integer programming and following the notation in [1], let Q be the problem at some node with $\hat{x}_j \notin \mathbb{Z}$ being the (fractional) LP solution of an integer variable x_j and $\hat{c}_Q$ the LP objective value. Furthermore, assuming that x_j is selected as branching variable, let Q_j^- and Q_j^+ be the resulting child problems with restrictions $x_j \leq \lfloor \hat{x}_j \rfloor$ and $x_j \geq \lceil \hat{x}_j \rceil$, respectively. Solving the LP relaxations of Q_j^- and Q_j^+ may then have three different outcomes:

- If both subproblems are infeasible, it can immediately be concluded that Q is also infeasible and, hence, can be cut off from the branch-and-bound tree.
- If either Q_j^- or Q_j^+ is infeasible while the other one is not, the LP solution $(\hat{x}_j, \hat{y}_j)$ can be separated by adding $x_j \geq \lceil \hat{x}_j \rceil$ or $x_j \leq \lfloor \hat{x}_j \rfloor$ to Q. Thus, no branching is required at all in this step.
- If neither Q_j^- nor Q_j^+ is found to be infeasible, we denote by $\hat{c}_{Q_j^-}$ and $\hat{c}_{Q_j^+}$ the respective LP objective values and compute the progress in terms of dual objective as

$$\Delta_j^- := \hat{c}_{Q_j^-} - \hat{c}_Q \qquad \text{and} \qquad \Delta_j^+ := \hat{c}_{Q_j^+} - \hat{c}_Q.$$

Since, from a minimization perspective, the dual objective value of some problem can not decrease when imposing further restrictions, it is ensured that $\Delta_j^- \geq 0$ and $\Delta_j^+ \geq 0$. Finally, the two values are combined to a single score s_j by a scoring function, i.e., $s_j := \text{score}(\Delta_j^-, \Delta_j^+)$. A commonly used example of a scoring function that has proven to work well in practice is the multiplicative form

$$\text{score}(q_1, q_2) = \max\{q_1, \epsilon\} \cdot \max\{q_2, \epsilon\} \tag{72}$$

with a small $\epsilon > 0$ that allows the score to be positive even if there is no objective gain in one of the subproblems [1].

On a local scale, this concept yields the best possible branching variable. Its main drawback, however, is the large computational effort required to solve all the resulting LPs. In part, this can be alleviated by restricting the set of branching candidates by some criterion or by only performing a few simplex iterations on each subproblem.

Strong branching can easily be adapted to piecewise linear constraints by generating child nodes according to one of the above methods and working with the respective convex hull relaxations. However, there are some issues that need to be taken care of. First, as seen above, the branching methods `2-CHILD-LEFT` and `2-CHILD-RIGHT` do not guarantee that the LP solution $(\hat{x}_j, \hat{y}_j)$ is excluded from both subproblems that are generated. Since solving some child node LP relaxation that still contains $(\hat{x}_j, \hat{y}_j)$ would be redundant, it should be checked whether this is the case before performing strong branching. At the same time, it could still make sense to choose x_j as branching variable, for example, if a large objective gain is achieved at the child node that excludes $(\hat{x}_j, \hat{y}_j)$. The second issue with strong branching on piecewise linear constraints is related to the scoring function used to rank branching candidates. In order to be able to apply branching method `3-CHILD` and to compare the scores of several possible branching decisions on an undistorted basis, we propose

$$\text{score}(q_1..q_k) = \left(\prod_{i=1}^{k} \max\{q_i, \epsilon\} \right)^{1/k} \tag{72*}$$

as a generalization of (72), with k being the number of child nodes to be created. Notice that while (72) and (72*) do not yield the same score value for one and the same branching decision in the case of $k = 2$, the ranking of the candidates remains unaffected. As an alternative, one might also think of using (72) with the two minimal values in $\{q_1..q_k\}$ as arguments.

Pseudocost branching

With the goal of keeping the branching tree small and the running time low in mind, yet another perspective on selecting a branching candidate is introduced by keeping record of the success of previous branching decisions. Based on this idea, a method known

as pseudocost branching has been developed for mixed-integer programs. Assuming, again, that some variable x_j has been selected as branching variable in problem Q, let $f_j^- := \hat{x}_j - \lfloor \hat{x}_j \rfloor$ and $f_j^+ := \lceil \hat{x}_j \rceil - \hat{x}_j$. A normalized gain in objective can be computed for both of the resulting subproblems Q_j^- and Q_j^+ as

$$\varsigma_j^- := \frac{\Delta_j^-}{f_j^-} = \frac{\hat{c}_{Q_j^-} - \hat{c}_Q}{\hat{x}_j - \lfloor \hat{x}_j \rfloor} \qquad \text{and} \qquad \varsigma_j^+ := \frac{\Delta_j^+}{f_j^+} = \frac{\hat{c}_{Q_j^+} - \hat{c}_Q}{\lceil \hat{x}_j \rceil - \hat{x}_j}. \tag{73}$$

Then, in the course of the branch-and-bound algorithm and for each integer variable x_j, the averages ψ_j^- and ψ_j^+ of these gains over all problems Q where x_j has been selected as branching variable are calculated. These are referred to as pseudocosts. Evidently, only problems whose child nodes' LP relaxations have been solved and found feasible can be taken into account. With the aid of pseudocosts, the dual bound progress for subsequent branching decisions can then be estimated by $f_j^- \psi_j^-$ and $f_j^+ \psi_j^+$, respectively. Finally, these predictions are passed as arguments to a scoring function as described above.

Pseudocost branching is a computationally fast way to judge branching possibilities with increasing quality as the algorithm learns from its previous decisions. However, since no information on the possible success of branching on any variable is available beforehand and all necessary data can only be acquired throughout the solving process, it relies on some suitable initialization procedure for the values of ψ_j^- and ψ_j^+. In a trivial strategy, one could set some uninitialized pseudocost value for downward or upward branching to the average of all initialized pseudocosts of the same direction, starting with a value of 1 in case that all pseudocosts are uninitialized. The simplicity of doing so is paid for by making rather unfunded decisions in the upper part of the branch-and-bound tree where the quality of the judgement matters most. As an option to cure this drawback, pseudocost branching can be combined with strong branching in different ways. For example, one may apply strong branching up to a certain depth of the branching tree or use strong branching results for the initialization of the pseudocosts. One of the most sophisticated variants of these hybrid branching strategies, known as reliability branching, will be presented below in a separate paragraph.

Before pseudocost branching can be applied to piecewise linear constraints in a similar manner, there are two aspects that need to be revised. At first, the definition of the variable value changes f_j^- and f_j^+ as stated above is obviously no longer meaningful in the presence of continuous variables. Since there is no unique equivalent, Berlotti et al [6] propose a variety of suitable formulations, for example considering the new bound intervals or the Euclidean distance from the parent node LP solution to those of the child nodes. We will assume for now that only two child nodes are to be generated and we distinguish between the two different situations in which the values of f_j^- and f_j^+ are needed:

- For the purpose of updating the pseudocosts in analogy to (73), we have opted for a variant that takes both constraint variables into account by defining

$$f_j^- := \left\| \begin{pmatrix} \hat{x}_j \\ \hat{y}_j \end{pmatrix} - \begin{pmatrix} \hat{x}_j^- \\ \hat{y}_j^- \end{pmatrix} \right\|_2 \qquad \text{and} \qquad f_j^+ := \left\| \begin{pmatrix} \hat{x}_j \\ \hat{y}_j \end{pmatrix} - \begin{pmatrix} \hat{x}_j^+ \\ \hat{y}_j^+ \end{pmatrix} \right\|_2,$$

  with $(\hat{x}_j^-, \hat{y}_j^-)$ and $(\hat{x}_j^+, \hat{y}_j^+)$ denoting the child LP solutions. As already mentioned, it may happen that $(\hat{x}_j, \hat{y}_j)$ is still feasible for one of the child problems when using branching method `2-CHILD-LEFT` or `2-CHILD-RIGHT`. In this case, f_j^- or f_j^+ will be zero such that ς_j^- or ς_j^+ is undefined and no update is possible, making both of these branching methods an inferior choice in the context of pseudocost branching.

- In order to compute the estimated dual bound progress as $f_j^- \psi_j^-$ and $f_j^+ \psi_j^+$, we set f_j^- and f_j^+ to the Euclidean distance from the parent LP solution to the feasible region of the respective child node relaxation. Notice that if `2-CHILD-LEFT` or `2-CHILD-RIGHT` is used, either f_j^- or f_j^+ may again be zero for the same reasons as just explained. This time, however, the implication of the dual bound progress being zero is correct given that the child LP solution remains the same as in the parent node.

As the second issue arising in the context of piecewise linear constraints, we have to take special care of the `3-CHILD` branching method, for which at least three different treatments are conceivable:

- Let Q_j° denote the subproblem at the middle child node after branching on variable x_j in some problem Q and keep track of the success of these middle branches by introducing pseudocost variables ψ_j°. Thereby, similar as above, the variable value change is defined as

$$f_j^\circ := \left\| \begin{pmatrix} \hat{x}_j \\ \hat{y}_j \end{pmatrix} - \begin{pmatrix} \hat{x}_j^\circ \\ \hat{y}_j^\circ \end{pmatrix} \right\|_2.$$

  In this case, however, ψ_j° may contain less information than ψ_j^- and ψ_j^+ since no middle child node is generated every time the solution $\hat{x}_j$ is equal to the x-coordinate of some support point.

- Incorporate the normalized gain in the objective of the middle branch in the computation of ς_j^- and ς_j^+ in case that three child nodes are to be generated, e.g.,

$$\varsigma_j^- := \frac{2}{3} \cdot \frac{\Delta_j^-}{f_j^-} + \frac{1}{3} \cdot \frac{\Delta_j^\circ}{f_j^\circ} \qquad \text{and} \qquad \varsigma_j^+ := \frac{2}{3} \cdot \frac{\Delta_j^+}{f_j^+} + \frac{1}{3} \cdot \frac{\Delta_j^\circ}{f_j^\circ}.$$

- Ignore all pseudocost information of the middle branches and proceed with only considering ψ_j^- and ψ_j^+.

Reliability branching

As a remedy for the problem of uninitialized pseudocosts at the beginning of the branch-and-bound procedure, pseudocost branching is often used in combination with strong branching. Doing so, at the same time, helps to keep within limits the high computational costs that pure strong branching would incur. A very dynamic way of combining the two strategies is referred to as reliability branching, based on a simple principle. As long as some candidate x_j has not been selected as branching variable at least η times, its corresponding pseudocosts are considered unreliable. In this case, strong branching is performed on x_j, otherwise the score value of x_j is computed from the available pseudocosts. Consequently, η is called reliability parameter. In practice, the results of strong branching attempts count towards the reliability of the pseudocosts of some variable even if it is not chosen as branching variable in the end.

3.2.4 Propagation

Using the proposed relaxation together with one of the branching rules suffices for eventually obtaining an optimal solution of an optimization problem with piecewise linear constraints. However, the solving process will be very slow since no further information on the constraint variables is exploited. Therefore, we will now present several ideas about how variable bounds and constraint relaxations can be tightened as a result of propagating relevant information among constraints. Propagation is an essential part of the inference component in the context of constraint programming as introduced in section 1.2.

There are two views onto the topic of propagation that we will take care of: First, information can be deduced from a piecewise linear constraint to be provided to the rest of the model and, second, a constraint may be modified due to newly gathered knowledge from other parts of the model.

As a start, the domain $\mathcal{D}$ of the piecewise linear function g can be used to tighten the bounds of the variable x. More precisely, we can impose $x \geq \bar{x}_0$ and $x \leq \bar{x}_n$ since the constraint is only defined between these two points. In order to obtain restrictions on y, we need to determine $y_{min} := \min\{\bar{y}_0..\bar{y}_n\}$ and $y_{max} := \max\{\bar{y}_0..\bar{y}_n\}$. Then we can derive $y \geq \frac{y_{min}-b}{a}$ from an inequality constraint and, additionally, $y \leq \frac{y_{max}-b}{a}$ if the constraint is an equation. These bounds on y are, of course, implicitly incorporated into the model by the convex hull relaxation itself. Despite that, explicitly updating the variable bounds may be useful if not all convex hull inequalities shall be inserted into the LP from the start and in order to give other constraint handling methods access to this information even before an LP is solved. We will refer to the two propagation steps as `PROPAGATE(x)` and `PROPAGATE(y)`, respectively.

Next, we want to study how a piecewise linear constraint and its relaxation can be modified once a new bound on either x or y becomes known. Given a bound of the type $x \leq c$, there are three different cases that may occur:

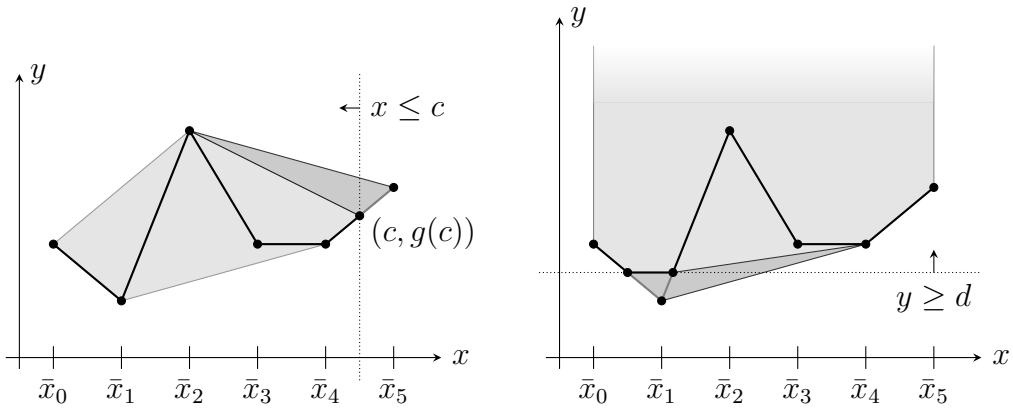
- $c \geq \bar{x}_n$: The new bound is redundant.
- $c < \bar{x}_0$: The current problem is infeasible.
- $c \in [\bar{x}_0, \bar{x}_n)$: The domain of g can be updated to $\mathcal{D} := [\bar{x}_0, c]$ by removing all support points $(\bar{x}_k, \bar{y}_k)$ with $\bar{x}_k > c$ and adding the point $(c, g(c))$ if it is not equal to an already existing one. Then, a tighter convex hull relaxation can be computed for the modified constraint.

In the following, we call this procedure `UPDATE(x)`. If a bound $x \geq c$ becomes known, we can proceed analogously, whereas $x = c$ implies $y \geq \frac{g(c)-b}{a}$ or $y = \frac{g(c)-b}{a}$, depending on the type of the piecewise linear constraint. An example of `UPDATE(x)` is shown below in figure 14a. In general, if a constraint's relaxation has successfully been tightened by this method, it would make sense to update the values of y_{min} and y_{max} with respect to the new set of support points and to call `PROPAGATE(y)` in case they have changed.

For the following considerations on the propagation of y-variable bounds, we want to restrict ourselves to piecewise linear inequality constraints $g(x) \leq ay + b$. By treating equations as two opposing inequalities, the presented ideas can easily be applied to this type of constraint as well. Assuming that a new bound $y \geq d$ becomes known and using the previously computed minimum and maximum y-values y_{min} and y_{max} of the function g, again, we have to distinguish between several cases:

- $d \leq \frac{y_{min}-b}{a}$: The new bound is redundant.
- $d \geq \frac{y_{max}-b}{a}$: The new bound renders the piecewise linear constraint redundant, such that it can be removed from the current subproblem.
- $d \in (\frac{y_{min}-b}{a}, \frac{y_{max}-b}{a})$: The new bound can be incorporated into the constraint in order to obtain a tighter relaxation. Therefore, we update the piecewise linear function g by modifying its set of support points as follows: First, compute all points at which the straight line $y = d$ intersects the graph of the function $\frac{g-b}{a}$. This can be done by considering all of its linear pieces individually. If $(\tilde{x}, \tilde{y})$ is such a point, add $(\tilde{x}, ad+b)$ to the set of support points of g. Then, remove all support points $(\bar{x}_k, \bar{y}_k)$ with $\frac{\bar{y}_k-b}{a} < d$. Finally, if $(\bar{x}_0, \bar{y}_0)$ or $(\bar{x}_n, \bar{y}_n)$ have been eliminated, add $(\bar{x}_0, ad+b)$ or $(\bar{x}_n, ad+b)$, respectively.

This procedure will be named `UPDATE(y)` and an illustration is given in figure 14b.



(a) `UPDATE(x)` **(b)** `UPDATE(y)`

Figure 14: Update of constraints due to variable bound changes. In both pictures, the area shaded in light gray depicts the new relaxation that has been obtained by cutting off the dark gray part from the old one.

If, by contrast, a bound of the form $y \leq d$ arises from the current problem, we can not follow the same steps as in `UPDATE(y)` but have to proceed differently. The reason for this is that such a bound in combination with an inequality constraint $g(x) \leq ay+b$ can cause some x-values within the domain $\mathcal{D}$ to be infeasible, which happens wherever d is smaller than the function value of $\frac{g-b}{a}$. However, we can at least apply a simple strategy for deducing tighter bounds on x or detecting infeasibility of the whole subproblem. It will be referred to as `PROPAGATE(x,y)`.

- $d > \frac{y_{max}-b}{a}$: The new bound does not affect the piecewise linear constraint in the sense that it may not be used to tighten the relaxation.
- $d < \frac{y_{min}-b}{a}$: The current subproblem is infeasible.
- $d \in [\frac{y_{min}-b}{a}, \frac{y_{max}-b}{a}]$: Starting from the leftmost point $\bar{x}_0$ of $\mathcal{D}$ and moving to the right, search the first value ($\tilde{x}_l \in \mathcal{D}$ for which $\frac{g(\tilde{x}_l)-b}{a} \leq d$ is fulfilled. In other words, we want to find $\tilde{x}_l$ such that $\frac{g(x)-b}{a} > d$ for all $x < \tilde{x}_l$. Analogously, determine $\tilde{x}_r$ by starting from the rightmost point $\bar{x}_n$ and moving leftwards. Having done this, we can restrict the domain of the x-variable to $x \in [\tilde{x}_l, \tilde{x}_r]$.

A successful call to `PROPAGATE(x,y)`, as shown in figure 15, might be followed by a constraint data update of type `UPDATE(x)` in order to immediately incorporate the new information into the relaxation. In our implementation of a constraint handler in SCIP, as we will see later, only the three `PROPAGATE()` methods are performed in the propagation step, whereas the `UPDATE()` methods are used for separation.

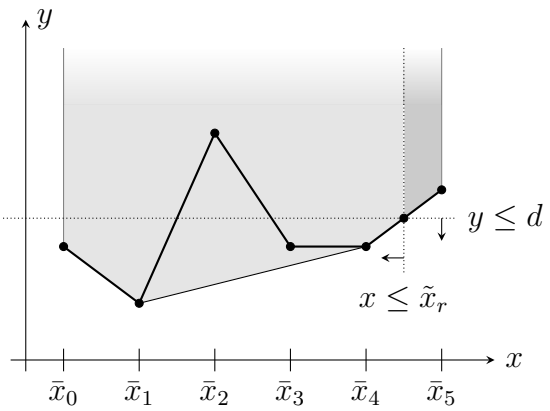


Figure 15: Inference of x-variable bound from y-variable bound according to `PROPAGATE(x,y)`. From $y \leq d$ a new bound $x \leq \tilde{x}_r$ has been deduced, which allows to cut off the dark gray part of the relaxation. The remaining relaxation is shaded in light gray.

3.2.5 Separation

In mixed-integer programming, cut separation has been shown to be a very powerful extension to the basic branch-and-bound procedure. Given a fractional solution of the LP relaxation, the goal is to find further inequalities that cut off this solution from the set of feasible values. Hence, such inequalities are referred to as cutting planes, with the resulting algorithm being known as branch-and-cut method. Repeatedly adding cutting planes to the LP continually improves the dual bound of the problem and may eventually lead to an integral solution. In general, as shown by Gomory [16], it is possible to determine a cutting plane for every non-integral LP solution of a mixed-integer program. However, these cuts sometimes yield numerical issues and are often not the best that one can do. There has been extensive research on the topic of separation that led to a variety of ideas for cutting planes that are adapted to specific problem types, such as, for example, cover cuts for Knapsack problems.

As for the case of piecewise linear constraints, a-priori, it is not possible to generate valid cutting planes. Since we are working with the convex hull relaxation, cutting off an LP solution that is not feasible for the original constraint would invariably mean cutting off part of the feasible region as well. That is, we can only hope to find cutting planes if the current problem bounds on the constraint variables are tighter than those induced by the constraint itself. If this is the case, we proceed similarly as in the methods `UPDATE(x)` and `UPDATE(y)` but without actually modifying the constraint data: Given the current variable domains, the set of support points of the piecewise linear function is tentatively updated such that it incorporates the new bound information. Then, after having computed the convex hull relaxation of the constraint with respect to this set of points, each convex hull segment that differs from the original relaxation is checked for whether it separates the current LP solution. If yes, it is added as a linear inequality to the subproblem.

3.3 Multivariate Piecewise Linear Constraints

When dealing with constraints involving multivariate piecewise linear functions, some of the constraint programming components presented above can easily be generalized while others need more attention. Although our constraint handler for the SCIP framework will not be able to treat multivariate piecewise linear constraints, we still want to present some work related to this topic that might be useful for a future implementation.

Definition 14 ((Multivariate) Piecewise Linear Constraint)**.** *A constraint in the variables $\mathbf{x} \in \mathbb{R}^d$, $d \geq 2$, and $y \in \mathbb{R}$ is called piecewise linear if it is of the form*

$$g(\mathbf{x}) \leq ay + b \qquad \text{or} \qquad g(\mathbf{x}) = ay + b,$$

where the piecewise linear function $g : \mathcal{D} \to \mathbb{R}$ is defined by a finite number of discrete points $\{(\bar{\mathbf{x}}_i, \bar{y}_i))\}_{i=0}^n \subset \mathbb{R}^{d+1}$ on a simplicial triangulation $\mathcal{S}$ of a rectangular domain $\mathcal{D} \subset \mathbb{R}^d$ according to definition 9, $a > 0$ and $b \in \mathbb{R}$.

The convex hull relaxations of multivariate equality and inequality constraints are essentially the same as in the univariate case and we only want to state them again for the sake of completeness:

$$g(\mathbf{x}) \leq ay + b \qquad \Rightarrow \qquad (\mathbf{x}, y) \in \operatorname{epi} \operatorname{convenv}_{\mathcal{D}} \left(\frac{g(\mathbf{x}) - b}{a} \right),$$

$$g(\mathbf{x}) = ay + b \qquad \Rightarrow \qquad (\mathbf{x}, y) \in \operatorname{conv}(\{(\bar{\mathbf{x}}_i, \bar{y}_i)\}_{i=0}^n).$$

In general, given a number of p points in $\mathbb{R}^q$, the computation of their convex hull can be done in $\mathcal{O}(p \log p + p^{\lfloor q/2 \rfloor})$ time. Hence, for $q > 3$, the second term is dominating, which drastically increases the time complexity and even makes it exponential in the dimensionality if q is considered as part of the input. For details, again, see section 4.3. Making use of the fact that convex hulls can be determined faster in low dimensions, separable functions are computationally advantageous to non-separable ones. It can be shown that the convex hull of a multivariate separable piecewise linear function may be expressed using the convex hulls of all its univariate function parts separately [18]. This, of course, extends to the relaxations of piecewise linear constraints. Hence, from now on, let us assume that we are working with non-separable multivariate functions.

If, during the branch-and-bound process, a multivariate piecewise linear constraint is violated by the constraint variables' LP solution $(\hat{\mathbf{x}}, \hat{y})$ at some node and branching becomes necessary, we can not rely on the same methods as described in 3.2.2 for the univariate case. First of all, the $\mathbf{x}$-variable now consists of several components, each of which can be used for branching. Most of the additional complication, however, is introduced by the underlying triangulation $\mathcal{S}$. Instead of relying on the natural ordering of intervals, we now have to handle d-dimensional simplices. For example,

once some branching component x_j has been determined, simply imposing $x_j \leq \hat{x}_j$ and $x_j \geq \hat{x}_j$ in order to create two subproblems would obviously conflict with the simplicial structure of the domain. Generally, the problem comes down to two main tasks that may be sketched as follows, with an illustration given in figure 16 below.

- Compute the intersection of a triangulation $\mathcal{S}$ and a hyperplane H. This can be done by considering each simplex $S \in \mathcal{S}$ separately and will result in a set of intersection points that subsequently need to be incorporated into the triangulation.
- Update a triangulation and restore its simplicial structure with respect to added or removed vertices. For the class of so-called Delaunay triangulations that are often used in practice due to their numerical favorability, such updates can be performed, e.g., by a simple edge-flip algorithm in $\mathcal{O}(n^2)$ or by a more sophisticated randomized incremental algorithm in $\mathcal{O}(n \log n)$ time. For details on these methods we refer to some textbook on Computational Geometry such as [8].

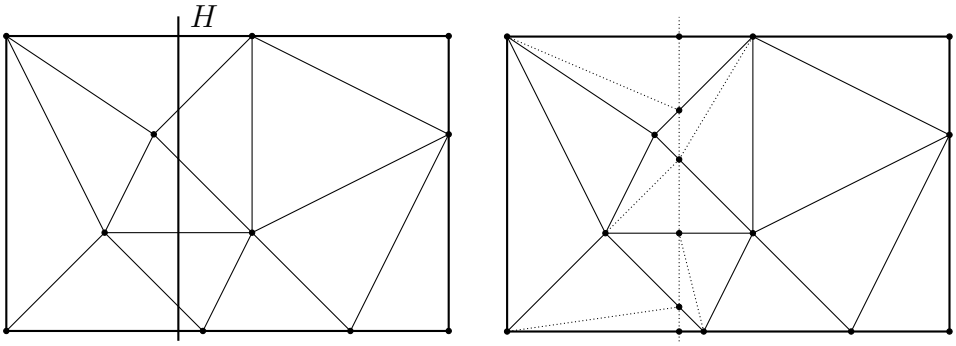


Figure 16: Intersection of a triangulation and a hyperplane.

With the help of this intersection and update procedure, we are now able to generate branching subproblems. It only remains to determine the function values of g at the newly inserted triangulation vertices and to compute the convex hull relaxations on both of the restricted domains. Alternatively, if the triangulation is of a special structure, other branching strategies not relying on a single variable are conceivable. As an example in two dimensions, see the "Union Jack" triangulation described in the paper of Nemhauser and Vielma [33], allowing for branching on diagonals.

By the same update and refinement steps as described above, it is possible to establish an adapted version of the `UPDATE(x)` method. The other topics presented above, i.e., propagation, branching candidate selection and separation, quite naturally extend to multivariate piecewise linear constraints. For a more detailed consideration of multivariate piecewise linear functions, see for example [14].

4 Piecewise Linear Constraints in SCIP

In this chapter, we are going to demonstrate how our convex hull-based approach for handling piecewise linear constraints can be implemented in SCIP, a non-commercial mathematical programming framework developed at the Konrad Zuse Institute in Berlin [19]. SCIP is built upon the concept of constraint integer programming (CIP) that has been developed by Achterberg [1]. By employing ideas and features of both mixed-integer (MIP) and constraint programming (CP), SCIP can be used as a fast and flexible solver for a wide variety of mathematical optimization problems. For all topics related to SCIP as well as an extensive description of the various solving techniques that are applied, we refer the reader to [1]. Meanwhile, we will content ourselves with a brief introduction to the basic concepts of SCIP before describing in detail our implementation work on piecewise linear constraints.

4.1 Basic Concepts of SCIP

SCIP provides the basic infrastructure for solving constraint integer problems by means of branch-and-bound techniques. Thereby, the search for a solution is directed by algorithms contained in various plugins. Some plugins are already included by default in a SCIP distribution, e.g. those required for solving MIPs, whereas in general they can be implemented by the users according to their specific problem requirements.

The most important type of plugin are so-called constraint handlers that are responsible for the representation and treatment of a single class of constraints each. The primary task of a constraint handler is to ensure that a problem solution is feasible with respect to all constraints of its type. However, in order to enhance the solving process, it should contain further means of exchanging information with the SCIP framework, such as presolving and propagation methods, a linear relaxation and branching strategies.

Other plugins that can be provided for SCIP are, for example, presolvers, variable pricers or primal heuristics, depending on the problem structure and the user's preferences. Since the focus of this work is on a constraint handler for piecewise linear functions, we will not go into detail with these other plugin types.

In a typical run of SCIP, several different stages are passed through during the solving process. After some initialization and the problem specification by the user, a working copy of the original problem is created. This copy is referred to as transformed problem and will be subject to modifications in the course of the search procedure, whereas the original problem is kept unchanged. Before the main solving process is started, presolving methods are called with the aim of reducing the size of the problem, e.g., by detecting redundant constraints or variables that might be fixed. A constraint handler may come into action in each of these stages and should provide the relevant methods for its constraint type for all tasks that are implied.

4.2 Implementation of the Constraint Handler

In this section, we will explain the internal structure of a constraint handler along with the specific methods that have been added for the case of piecewise linear constraints of the form

$$l \leq g(x) - y \leq r.$$

The SCIP framework is implemented in C, and, hence, we are also using C as programming language for our constraint handler although it would have been possible to choose C++ as well. A documentation of SCIP is available on its webpage [19]. In the following, when introducing a component of a constraint handler in SCIP, we will make use of this documentation for a general description before explicitly focussing on aspects of our implementation.

4.2.1 Constraint Handler Properties

First of all, some fundamental properties that determine the overall behavior of the constraint handler have to be set. These include

- `CONSHDLR_NAME`: a name by which the constraint handler is addressed,
- `CONSHDLR_DESC`: a short description,

and several others that will be mentioned at the places at which they are used.

4.2.2 Constraint Data

After that, everything that is required for the representation of a single piecewise linear constraint needs to be specified and stored in a data structure called `SCIP_ConsData`. In our case and using SCIP data types, it contains:

`SCIP_VARS*`	`vars`	An array containing the variables occuring in the constraint,
`int`	`nvars`	together with their number. There should be exactly two variables, one taking the role of x and one of y. Hence, although variable names can be arbitrary, we shall refer to the variables as 'x' and 'y' throughout our following explanations.
`SCIP_Real*`	`xvals`	Arrays containing the abscissa and ordinate values of the
`SCIP_Real*`	`yvals`	support points of the piecewise linear function, respectively.
`int`	`npoints`	The values are stored in increasing order with respect to their x-coordinate. The array size must correspond to the total number of points.

SCIP_Real	lhs	The left-hand and right-hand side of the constraint, corresponding to l and r in the above constraint pattern.
SCIP_Real	rhs	
int*	uhull	An array that contains the indices of the support points belonging to the upper part of the convex hull, together with its size. If the upper hull is not required, the value is NULL.
int	uhpoints	
int*	lhull	Analogously, the lower part of the convex hull.
int	lhpoints	

Moreover, SCIP_ConsData contains some auxiliary properties for storing the monotonicity and the curvature of the piecewise linear function as well as its minimum and maximum y-values, such that these information can be accessed whenever required.

4.2.3 Constraint Handler Data

Similarly, all data that are not specific to a single constraint but belong to the constraint handler as a whole, e.g., parameters controlling certain features, can be stored in SCIP_ConshdlrData. Here, we only want to briefly mention the constraint handler data that we used while explaining the meaning of the entries in more detail on occurrence:

SCIP_Bool	addcutsinit	If set to TRUE, add all bounding inequalities of the convex hull relaxation on constraint initialization; otherwise only add them when required/violated.
SCIP_Bool	splitcons	Split constraints into one lower- and one upper-bounded constraint in case that both lhs and rhs are finite.
SCIP_Bool	sepallcons	Try to find separating cuts for all constraints or stop when first cut has been found.
SCIP_Bool	updateconslocal	Tighten the relaxation using local or global variable bounds.
enumType	branchmethod	The branching method to be used.
enumType	branchcandsel	The strategy for branching candidate selection.
SCIP_Real*	pscostdown	Arrays for storing the pseudocosts for downward, middle and upward branches, respectively, for each piecewise linear constraint.
SCIP_Real*	pscostmid	
SCIP_Real*	pscostup	
int*	reliability	An array for storing the degree of reliability of the pseudocosts for each constraint, and the reliability threshold value.
int	relparam	

4.2.4 Fundamental Callback Methods

The interactions between a constraint handler and the solution framework are realized via a variety of callback methods that each provide a certain service to the solver. Some of the callbacks are classified as fundamental such that their implementation is compulsory, whereas all additional callbacks may or may not be included. The fundamental callbacks contain the semantic representation of the constraint class that is considered, enabling SCIP to find a correct optimal solution when constraints of this class are present. However, the solving process may be very slow without additional callbacks being implemented.

CONSCHECK:

The CONSCHECK callback is responsible for determining whether a given primal solution candidate is feasible with respect to all constraints of the constraint handler's type. If this is the case, `SCIP_FEASIBLE` is returned as a result, whereas the violation of at least one constraint yields `SCIP_INFEASIBLE`.

In order to check the feasibility of a solution $(\hat{x}, \hat{y})$ for the variables of a piecewise linear constraint, we start by considering $\hat{x}$. Following the notation of section 2.1 and assuming that the left- and rightmost support point $(\bar{x}_0, \bar{y}_0)$ and $(\bar{x}_n, \bar{y}_n)$ of the piecewise linear function g induce bounds on x, we declare the solution infeasible if $\hat{x} \notin [\bar{x}_0, \bar{x}_n]$. Otherwise, we go through all line segments and determine the index k such that $\hat{x} \in [\bar{x}_{k-1}, \bar{x}_k]$. Then, with

$$\tilde{y} := \frac{\bar{y}_k - \bar{y}_{k-1}}{\bar{x}_k - \bar{x}_{k-1}}(\hat{x} - \bar{x}_{k-1}) + \bar{y}_{k-1}$$

the solution is feasible for the constraint if and only if $\hat{y} \in [\,\tilde{y} - r, \tilde{y} - l\,]$.

The priority of calls to CONSCHECK relative to other constraint handlers' check methods can be determined using the constraint handler property `CONSHDLR_CHECKPRIORITY`.

CONSENFOLP:

With a solution of the LP relaxation of the problem at hand, the CONSENFOLP method can be called. Similar to CONSCHECK, its first task is to investigate whether the solution is feasible for all constraints. If so, `SCIP_FEASIBLE` is returned and nothing else happens. If some constraint is violated, however, the callback may not only return `SCIP_INFEASIBLE` but provide means to resolve the infeasibility.

In our setting, if some constraint itself is not satisfied, we check whether the solution $(\hat{x}, \hat{y})$ of the constraint variables lies within the convex hull of the constraint's feasible region. This will automatically be the case if the constraint handler property `addcutsinit` has been set to `TRUE`, since, then, all convex hull inequalities have already been included in the LP by the CONSINITLP callback. Otherwise, one or more violated bounding inequalities are computed and added as separating cuts to the LP.

On the other hand, i.e., if $(\hat{x}, \hat{y})$ violates the constraint but satisfies its convex hull relaxation, the constraint is marked as a branching candidate since no other means for resolving the infeasibility can be provided. Once that all constraints have been checked and no separating cuts have been found, this list of branching candidates is evaluated in order to determine the branching to be initiated. This happens according to the branching rule given by the `branchcandsel` parameter, with the following options being possible:

- `RANDOM`: A random selection of the branching candidate using the built-in C `rand()` function.

- `MOSTINFEAS`: Most infeasible branching following the description in section 3.2.3. In order to obtain the values d^{cons} and d^{rel}, we need to compute the distance from the LP solution to every line segment of the piecewise linear function and the convex hull relaxation and find the respective minimum. Thereby, the distance from a point $(\hat{x}, \hat{y})$ to some line segment with end points $(\bar{x}_j, \bar{y}_j)$ and $(\bar{x}_k, \bar{y}_k)$ is determined by a projection method: Let

$$t := \frac{\begin{pmatrix} \hat{x} - \bar{x}_j \\ \hat{y} - \bar{y}_j \end{pmatrix} \cdot \begin{pmatrix} \bar{x}_k - \bar{x}_j \\ \bar{y}_k - \bar{y}_j \end{pmatrix}}{\left\| \begin{pmatrix} \bar{x}_k - \bar{x}_j \\ \bar{y}_k - \bar{y}_j \end{pmatrix} \right\|^2}.$$

  If $t \leq 0$, the distance is $\left\| \left(\begin{smallmatrix} \hat{x} - \bar{x}_k \\ \hat{y} - \bar{y}_k \end{smallmatrix} \right) \right\|$, while it is $\left\| \left(\begin{smallmatrix} \hat{x} - \bar{x}_k \\ \hat{y} - \bar{y}_k \end{smallmatrix} \right) \right\|$ in the case $t \geq 1$. Finally, if $t \in (0, 1)$, with

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} := \begin{pmatrix} \bar{x}_j \\ \bar{y}_j \end{pmatrix} + t \cdot \begin{pmatrix} \bar{x}_k - \bar{x}_j \\ \bar{y}_k - \bar{y}_j \end{pmatrix}$$

  being the projection of $(\hat{x}, \hat{y})$ onto the line segment, the distance amounts to $\left\| \left(\begin{smallmatrix} \hat{x} - \tilde{x} \\ \hat{y} - \tilde{y} \end{smallmatrix} \right) \right\|$. For computational efficiency, it is possible to work with squared distances, since this does not affect the ranking of the branching candidates.

- `LEASTINFEAS`: Least infeasible branching in analogy to `MOSTINFEAS`.

- `RELPSCOST`: Reliability pseudocost branching whose behaviour can be specified via the constraint handler property `relparam` representing the reliability parameter η introduced in section 3.2.3. Until the required reliability is reached, strong branching is performed in order to initialize the pseudocosts.

- `STRONG`: Strong branching, as introduced earlier, relies on solving the LP relaxations of the subproblems to be generated. For branching on binary variables, SCIP allows to tentatively change the problem bounds and solve the resulting LPs efficiently by using the previously computed basis in what is called a warm start in LP theory. Thus, it is usually possible to arrive at a new optimum after relatively few iterations. However, SCIP does not provide the functionality to temporarily insert cuts into the LP as would be required in the context of piecewise linear constraints. As a workaround for this problem, we can only create a copy of the entire SCIP instance at the parent node, insert the tightened convex hull cuts and solve the problem from scratch. This understandably takes an enormous amount of time, since it has to be done at every branching decision and for each branching candidate, and thus is a crucial point for the overall performance of the constraint handler as we will see later.

The branching method to be applied can be specified by the `branchmethod` parameter of the constraint handler, with possible values as presented in 3.2.2. Furthermore, every child node that is generated is given a score value that indicates its importance of being processed to the SCIP framework. In order to compute this so-called node selection priority, we use built-in SCIP functions.

In the case of more than one type of constraint being present in the problem, the usual way of controlling the sequence in which the participating constraint handlers are called is through indicating a priority value for each of them. The priority belonging to the CONSENFOLP callback is stored in the `CONSHDLR_ENFOPRIORITY` constraint handler property. However, following this principle, when it comes to a branching decision, the branching candidates found by the piecewise linear constraint handler will not be judged jointly with candidates provided by other constraint handlers. For example, if the model contains binary variables and as long as at least one of them has a fractional LP value, branching to enforce their integrality would always be performed before branching on piecewise linear constraints or vice-versa. This lets our constraint programming approach be at a disadvantage compared to pure MIP models based on one of the formulations given in section 2, in which all branching candidates are considered simultaneously. In order to circumvent this restriction and to allow for an integrated evaluation of binary and piecewise linear branching candidates, we have introduced the parameter `branchlpcands`. If set to `TRUE`, it enables our constraint handler to jointly treat both candidate sets and select the overall best branching option.

At the end of the CONSENFOLP callback, depending on what measures have been taken, an appropriate SCIP result is returned. Precisely, `SCIP_CUTOFF` is used in case that infeasibility has been detected for the current node, e.g., by strong branching. Otherwise, if the domain has been reduced or if the solution has been separated, the result

is set to SCIP_REDUCEDDOM or SCIP_SEPARATED, respectively. If none of these apply and child nodes have been generated, SCIP_BRANCHED is returned. Finally, in case that no constraint has been found to be violated, the result can be set to SCIP_FEASIBLE.

CONSENFOPS:

In some cases, e.g., if there were numerical difficulties in the solving process, no LP solution is available for the current subproblem. In this situation, SCIP is able to resort to a pseudo solution that arises from solving the LP relaxation without any constraints but the variable bounds. Basically, the CONSENFOPS callback accomplishes the same tasks with the pseudo solution as CONSENFOLP does with the LP solutions. That is, it checks the solution against the existing constraints and, if required, tries to resolve infeasibilities. However, it may not add cutting planes to the LP and return SCIP_SEPARATED.

Since it has almost never been necessary for our test instances to call the CONSENFOPS callback, we have only provided it with some minimal working functionality. If given a pseudo solution, it simply returns SCIP_FEASIBLE if all constraints are satisfied and SCIP_INFEASIBLE, otherwise.

CONSLOCK:

The last of the fundamental callbacks, CONSLOCK, provides information to SCIP about whether the feasibility of a constraint may potentially be compromised when the value of any of the variables involved is increased or decreased.

As for piecewise linear constraints of the above type, in general, infeasibility may be caused by modifying x or y in either direction. However, there are a number of cases that allow for a more precise statement:

- If $l = -\infty$, then y can be increased arbitrarily.

  If $r = +\infty$, then y can be decreased arbitrarily.

- If g is monotonically increasing and $l = -\infty$, then x can be increased arbitrarily within the bounds induced by the constraint, i.e., as long as $x \leq \bar{x}_n$.

  If g is monotonically decreasing and $r = +\infty$, then x can be decreased arbitrarily within the bounds induced by the constraint, i.e., as long as $x \geq \bar{x}_0$.

The information about the monotonicity of the piecewise linear function g is stored in the monotonicity property of the constraint data. It may take the values MONO_INC, MONO_DEC, MONO_NONE and MONO_UNKNOWN. In the latter case, the callback tries to determine the monotonicity before setting the variable locks.

4.2.5 Additional Callback Methods

In the following, we are going to present a selection of the additional callbacks provided by SCIP that are of use in the context of piecewise linear constraints.

CONSTRANS:

If a constraint handler is capable of modifying the constraint data during preprocessing or the solving process, the original constraint needs to be copied in advance in order to keep an instance of the initial model in memory.

In our example, the CONSTRANS callback has been implemented since the data of a piecewise linear constraint that represents the mathematical formulation of the problem, i.e., the set of support points and the convex hull information, may be subject to modifications. In particular, updates are performed in the CONSSEPALP callback due to bound changes of the constraint variables.

CONSINITLP:

The CONSINITLP callback is responsible for making preparations with respect to the LP to be solved. Primarily, this means adding the LP relaxations of all constraints.

As described above, we have introduced the constraint handler property `addcutsinit` to control whether or not to insert all bounding inequalities of each constraint's convex hull relaxation.

CONSSEPALP:

With a valid LP solution at hand, the CONSSEPALP callback tries to find linear inequalities that separate this solution from the set of feasible points.

In our implementation, the separation procedure is mainly controlled by the constraint handler parameter `updateconslocal`:

- If set to `TRUE`, given a constraint that is violated by the LP solution, we first check whether the local bounds on the constraint variables are tighter than those induced by the constraint itself. If not, as explained in section 3.2.5, the relaxation already is as strong as possible and, hence, there is no chance for cutting planes to be found. Otherwise, we update the relaxation according to the local variable bounds by calling `UPDATE(x)` and/or `UPDATE(y)`, whichever of the two applies. Thereby, it is often not necessary to recompute the constraint's convex hull from scratch but only at regions where it has changed. While doing so, we verify whether the solution has been separated by one of the new convex hull inequalities. However, as a price for obtaining the locally best relaxation, the constraint data must be copied in advance such that the constraint remains unchanged at all other nodes of the branch-and-bound tree. In large problem instances, this may lead to a critical increase in memory usage.

- By contrast, if the parameter is set to `FALSE`, only the weaker global bounds on the constraint variables are used in `UPDATE(x)` and `UPDATE(y)`, thus avoiding the necessity of copying the constraint data while at the same time strengthening the relaxation at all branch-and-bound nodes. Potentially, this already entails a separating cut. If not, we can still use the local variable bounds to derive a cutting plane by tentatively updating the set of support points and the convex hull relaxation without actually modifying the constraint data.

It is easy to see that `UPDATE(y)` can only be applied to a constraint if either $l = -\infty$ or $r = +\infty$, since the `SCIP_ConsData` structure as defined above does not allow the lower and upper boundary of a constraint's feasible region to be modified independently from each other.

By the second parameter related to separation, `separateallcons`, it can be specified whether all constraints should be checked for cutting planes or whether the search should be stopped as soon as the first cut has been found. The callback returns `SCIP_SEPARATED` in the case that some separating cut could be generated and `SCIP_DIDNOTFIND`, otherwise. The frequency of calls to CONSSEPALP can be adjusted via the constraint handler option `CONSHDLR_SEPAFREQ`, whereas the priority relative to other constraint handlers' separation methods is indicated by `CONSHDLR_SEPAPRIORITY`.

CONSPROP:

The CONSPROP callback is one of the most powerful means of a constraint handler with respect to the enhancement of the solving process. Its task is the propagation of information on variable bounds induced by a single constraint at each node, thereby possibly tightening some variable's domain (return value `SCIP_REDUCEDDOM`) or detecting that the current subproblem is infeasible (return value `SCIP_CUTOFF`).

In order to implement the propagation methods `PROPAGATE(x)` and `PROPAGATE(y)`, given a piecewise linear constraint, our goal is to find a rectangular box that contains its feasible region in the plane. That is, we set $x^c_{lb} := \bar{x}_0$ and $x^c_{ub} := \bar{x}_n$, assuming that the support points of g are ordered according to their x-coordinate. As for the values stored in `SCIP_ConsData`, this is ensured by our implementation at the moment the constraint object is created. Furthermore, let $y^c_{lb} := \min\{\bar{y}_i \mid i = 1..n\} - r$ and $y^c_{ub} := \max\{\bar{y}_i \mid i = 1..n\} - l$. Notice that both y^c_{lb} and y^c_{ub} may be infinite depending on the value of r and l, respectively. Then, we can use a built-in SCIP function that compares these bounds with the local bounds of the current subproblem and, if possible or necessary, automatically tightens the domains accordingly or states infeasibility. If no infeasibility has been detected, the next step is a call to `PROPAGATE(x,y)`.

Similar to the constraint enforcement and separation callbacks, the frequency of calls to CONSPROP can be controlled using the `CONSHDLR_PROPFREQ` constraint handler property.

CONSPRESOL:

In the CONSPRESOL callback, presolving methods can be called that are supposed to help making the problem easier to solve. This may include, for example, inferring domain reductions for variables, deleting redundant constraints or upgrading constraints to more specific types. Presolving is usually done in rounds, thus allowing for a successive decrease of the problem complexity in each round. In our case, we have implemented the following presolving steps:

- **Propagation:** Trying to strengthen the variable bounds, propagation is applied as described in CONSPROP. This may already yield global infeasibility.
- **Constraint upgrade:** If some piecewise linear constraint consists of one support point only, we simply remove it from the set of constraints since all necessary information for the correctness of the model has been provided by the previous propagation. Moreover, constraints that have two support points are upgraded to the more specific type of linear constraints and subsequently be treated by their own constraint handler. The same holds true if there are multiple support points that are all collinear. The reason for doing so is that most constraint-specific methods for general piecewise linear constraints do not yield any advantage compared to the more specific ones that are included in the linear constraint handler.
- **Probing:** Another powerful presolving feature that has been developed in the context of mixed-integer programming is so-called probing. Originally, the concept simply was to tentatively fix some binary variable z to either of its possible values 0 and 1 and, each time, perform propagation in the entire model. Then, the newly obtained variable bounds from both cases are compared, hoping that they allow for the inference of useful information that is valid for the initial problem without z being fixed. For example, if the problem is infeasible for $z = j$, $j \in \{0, 1\}$, z can immediately be set to $1 - j$. Furthermore, new bounds on some other variable y can be deduced as follows:

$$\left.\begin{array}{lll} z = 0 & \Rightarrow & y \in [y^0_{lb}, y^0_{ub}] \\ z = 1 & \Rightarrow & y \in [y^1_{lb}, y^1_{ub}] \end{array}\right\} \quad y \in [\min\{y^0_{lb}, y^1_{lb}\}, \max\{y^0_{ub}, y^1_{ub}\}].$$

  These ideas can easily be transferred to piecewise linear constraints in order to implement a probing procedure on the x-variable. Therefore, we consider the intervals induced by the support points $\{\bar{x}_i\}_{i=1}^n$ from left to right and, for an interval $j \in \{1..n\}$, introduce the bounds $x \in [\bar{x}_{j-1}, \bar{x}_j]$. Then, if propagation states infeasibility, we can remove the interval from the constraint as long as no feasible interval has been found until this point. Of course, we proceed the same way from right to left, thus pruning the constraint from both sides. For all intervals that have not been declared infeasible, we compare the respective domains of the other variables and for each one choose the minimum lower and maximum upper bound as new globally valid bounds.

- **Convexity upgrade:** If the feasible region of a piecewise linear constraint is convex, the constraint is identical to its relaxation. Thus, it can be removed after having added all convex hull facets as linear constraints to the model.

The maximum number of presolving rounds that a constraint handler participates in can be specified by the `MAXPREROUNDS` parameter, also allowing to entirely disable the CONSPRESOL callback.

CONSINITPRE:

The CONSINITPRE callback is called before CONSPRESOL and may be used for the initialization of the preprocessing data. Furthermore, since it is executed even if presolving has been turned off, all constraint modifications that are necessary for the solving steps to follow can be done here.

In our case, we use this callback for splitting a constraint into a lower- and an upper-bounded part if the parameter `splitcons` has been set to `TRUE` and both $l > -\infty$ and $r < +\infty$. While removing the original constraint, two new constraints

$$l \leq g(x) - y \leq +\infty \qquad \text{and} \qquad -\infty \leq g(x) - y \leq r,$$

are generated. At the cost of increasing the problem size, this allows for exploiting the constraint structure in a more flexible way at some points of the solving process.

CONSINITSOL:

This callback is executed after the presolving step and before the beginning of the actual solving process. It may be used to initialize branch-and-bound specific data.

In CONSINITSOL, our constraint handler allocates memory for storing reliability pseudocost branching information in case that the parameter `branchcandsel` has been assigned the value `RELPSCOST`.

CONSPRINT:

The CONSPRINT method is responsible for generating an output string that represents a constraint of the constraint handler's type in CIP format. In our case, such a string generically looks as follows, containing all information that is necessary for uniquely defining a piecewise linear constraint:

```
<consname> :  lhs <= pwl(<x>)[(x0,y0) ..  (xn,yn)] - <y> <= rhs
```

CONSPARSE:

The CONSPARSE callback is the counterpart method of CONSPRINT for parsing a CIP formatted input string that specifies a piecewise linear constraint.

4.3 Convex Hull Computation

An efficient computation of the convex hull of a given set of points plays a crucial role in the proposed solution framework since it needs to be done repeatedly in the process of determining constraint relaxations. There has been extensive work on convex hull algorithms for several decades with major achievements in terms of running time and storage optimization. In the following section, we will concentrate on methods for computing planar convex hulls as required in our constraint handler.

The two-dimensional convex hull problem can be stated as the task of finding a convex hull representation of a given set P of n points in the plane. A very intuitive representation would be a clockwise or counterclockwise enumeration of the vertices of the convex hull, but we could, e.g., also think of a set of inequalities defining intersecting half spaces. By a reduction to the problem of sorting real numbers, it can be shown that $\Omega(n \log n)$ is a lower bound on the time complexity of the planar convex hull problem [30]. That is, no algorithm can be faster than that in the worst case.

Graham's Search:

The most famous approach for efficiently computing a planar convex hull is called Graham's Search and runs in $\mathcal{O}(n \log n)$ time [17]. Thus, it achieves the lower time complexity bound. The algorithm works incrementally, meaning that all points of P are considered one after another while updating the convex hull at each step. Andrew [3] proposed a slight modification of Graham's Search that we will present in algorithm 1, based on separately constructing the lower and the upper part of the convex hull.

Algorithm 1 Graham's Search.

Input: A set P of n points in the (x, y)-plane.
Output: A list $\mathcal{C}$ of the vertices of the convex hull in clockwise order.
1: Sort and relabel all points $p \in P$ increasingly according to their x-coordinate, using y-coordinates as tie-breakers. This yields a sequence $p_1..p_n$.
2: Create and initialize two ordered lists, $\mathcal{U} := (p_1, p_2)$ for the upper part of the convex hull and $\mathcal{L} := (p_n, p_{n-1})$ for the lower part.
3: **for** $i = 3 \rightarrow n$ **do**
4: $\quad \mathcal{U} := \mathcal{U} + p_i$
5: $\quad$ **while** $\mid \mathcal{U} \mid > 2$ **and** the last three points in $\mathcal{U}$ do not form a right turn **do**
6: $\quad\quad$ Delete the second-to-last point in $\mathcal{U}$.
7: $\quad$ **end while**
8: $\quad \mathcal{L} := \mathcal{L} + p_{n-i+1}$
9: $\quad$ **while** $\mid \mathcal{L} \mid > 2$ **and** the last three points in $\mathcal{L}$ do not form a right turn **do**
10: $\quad\quad$ Delete the second-to-last point in $\mathcal{L}$.
11: $\quad$ **end while**
12: **end for**
13: Remove p_1 and p_n from $\mathcal{L}$.
14: **return** $\mathcal{C} := \mathcal{U} + \mathcal{L}$.

Notice that the operator '+' is used to append a new element to an ordered list and to join two such lists together. Figuratively spoken, the algorithm exploits the fact that a clockwise walk along the edges of a convex polygon yields a right turn at every vertex. For a more detailed description, see for example [8].

It remains to specify how to test whether a given sequence of three points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and $p_3 = (x_3, y_3)$ makes a right turn or not. This can be done by evaluating the sign of the determinant

$$D := \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

If D is negative, then the points form a right turn, i.e., they follow a clockwise orientation. If it is equal to zero, then p_1, p_2 and p_3 are collinear. Otherwise, they form a left turn. However, the operation of computing a determinant is known to be numerically unstable in the sense that it might yield the wrong sign for some critical input. This could result in the output representing a non-convex set or even in a part of the correct convex hull being erroneously cut off. Alternatively, one could use a trigonometric approach by defining

$$D := \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right) - \arctan\left(\frac{y_3 - y_1}{x_3 - x_1}\right).$$

The interpretation of the sign of D is similar as above. For a correct computation of arctan, many programming languages provide numerically stable implementations such as the `atan2` function in C.

Sorting the points in increasing order of their x-coordinates can be performed in $\mathcal{O}(n \log n)$ time, using, e.g., HeapSort. The outer for-loop is executed $n-2$ times. In each of those iterations, both inner while-loops are executed at least once. If they are executed more than once, then one point is deleted from the respective list at each iteration. Since every point can only be deleted once, the loops take a total of $\mathcal{O}(n)$ time. This finally proves the total time complexity of Graham's Search to be $\mathcal{O}(n \log n)$.

With regard to our SCIP constraint handler, Graham's Search is a very suitable option for computing a constraint's convex hull relaxation. The support points of the piecewise linear function g may be given to the CONSPARSE callback in arbitrary order as problem input. However, since many of the implemented methods rely on the ordering of the points according to the x-coordinate, they will only be stored in `SCIP_ConsData` after having been sorted. Thus, the sorting step, if at all necessary, is performed independently from convex hull computation and will be done no more than once for each constraint. Then, on the basis of a sorted set of points, Graham's Search allows for determining all convex hulls in linear time, as seen above. Whenever only a part of the convex hull needs to be computed, e.g., if not the entire hull is affected in `UPDATE(x)`, we only apply Graham's Search to the support points that are concerned.

Dynamic convex hull computation:

Graham's Search is a static method in a way that it computes the convex hull of a set of points P that is fully known from the beginning. However, it provides no means for updating the result if, later on, points are added to or removed from P, apart from recomputing the convex hull from scratch each time P has been modified. Since the relaxations of the piecewise linear constraints that are to be dealt with receive constant refinement during the solution process, for example due to calls to `UPDATE(x)` or `UPDATE(y)`, it might therefore be worthwhile to consider methods of dynamic convex hull computation as part of future work. At this place, we will only briefly review some of the existing approaches related to this topic.

We want to begin with a special case of the dynamic convex hull problem, known as its on-line variant, that only allows for the insertion but does not support the deletion of points. Using Graham's Search, an update of the convex hull due to the insertion of a single point would take $\mathcal{O}(n)$ time which is not the best that one can do. By changing the data structure for representing the convex hull to a height-balanced binary search tree, Preparata showed that the time complexity can be improved to $\mathcal{O}(\log n)$ [29].

Going one step further, Overmars and van Leeuwen proposed an algorithm that is capable of handling both point insertions and deletions in $\mathcal{O}(\log^2 n)$ time per transaction, making it a fully dynamic maintenance procedure for convex hulls [25]. Thereby, again, a search tree is used for storing the points.

Convex hulls in higher dimensions:

In arbitrary dimension d, the time complexity of computing the convex hull of a set P of n points can be shown to be at least $\mathcal{O}(n \log n + n^{\lfloor \frac{d}{2} \rfloor})$. For $d = 3$, similar to the planar case, this yields $\mathcal{O}(n \log n)$ which can be achieved, for example by an algorithm proposed by Chan [9]. This algorithm, moreover, has the property of being output-sensitive, meaning that its time complexity can be stated more precisely as $\mathcal{O}(n \log h)$ where h denotes the number of vertices in the convex hull.

There exist several methods for computing convex hulls in higher dimensions $d > 3$, including randomized algorithms like, for example, the one presented by Clarkson and Shor [10]. Pursuing part of their ideas, Barber, Dobkin and Huhdanpaa developped a higher-dimensional variant of the renowned planar QuickHull algorithm [5].

4.4 Application in Gas Network Optimization

In order to study the computational properties of our constraint handler for piecewise linear functions and to compare it with the classical MIP approaches, we have used test instances of so-called nomination validation problems that occur in the context of gas network optimization: Given a gas transmission network that consists of active, controllable elements (e.g. valves and compressor stations) and passive pipelines, and

given a nomination that describes the in- and outflow of gas at each node, the question is whether the network can be operated in a way such that all physical, technical and legal restrictions are fulfilled [28]. While this definition by itself only indicates the search for a feasible solution, adding an objective function yields a true optimization problem. For example, one could choose to minimize the pressure loss within the network or to find a network configuration that incurs minimum costs.

A gas network can be modelled as a directed graph $G = (V, E)$ with each node in V being either entry, exit or intermediate point, whereas all active and passive elements are represented by arcs. For all nodes $v \in V$, pressure variables $p_v \in \mathbb{R}^+$ are introduced that are lower- and upper-bounded due to technical matters. Furthermore, each arc $e \in E$ is assigned a flow variable q_e with the help of which simple flow-conservation constraints can be imposed at each vertex.

The flow of gas through a pipeline $e = (v, w) \in E$ is determined by the difference in pressure at its end points v and w. There exists a non-linear relation between flow and pressure in a pipe that can be described by the so-called WEYMOUTH equation

$$q_e \cdot |q_e| = \alpha_e \cdot (p_v^2 - \beta_e \cdot p_w^2),$$

where α_e and β_e are pipe specific constants depending on the diameter, length and altitude difference of the pipe. Using pressure square variables $\pi_v := p_v^2$ and $\pi_w := p_w^2$, the Weymouth equation can be equivalently expressed as

$$q_e \cdot |q_e| = \alpha_e \cdot (\pi_v - \beta_e \cdot \pi_w) \qquad \Leftrightarrow \qquad \begin{cases} z = \alpha_e \cdot (\pi_v - \pi_w) \\ q_e \cdot |q_e| = z \end{cases} .$$

Hence, considering the pressure square relation as well as the second part of the split Weymouth equation, we are now facing two different non-linear dependencies. Further and more complicated non-linearity is introduced by the models for the remaining network elements. See [28] for details.

In order to deal with non-linear functions, we will work with their piecewise linear approximations. That is, for some relation $y = f(x)$ with f being non-linear, we consider restrictions of the form

$$-\epsilon \leq g(x) - y \leq \epsilon,$$

where g is a piecewise linear approximation of f and $\epsilon \geq 0$ some approximation error. Obviously, this complies with the type of constraint that our SCIP constraint handler is able to process. Notice that we have performed all computational studies on existing data and that the approximation procedure has not been a topic of this thesis. Figure 17 shows an exemplary piecewise linear approximation of the non-linear part of the Weymouth equation.

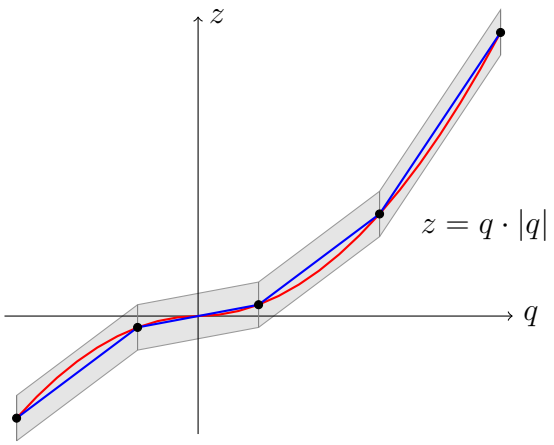


Figure 17: Piecewise linear approximation of a non-linear function. The area shaded in gray represents the feasible region of the resulting constraint.

4.5 Computational Results

In this section, we finally want to present some computational results of running our piecewise linear constraint handler as part of the SCIP framework and put them into perspective with those obtained by the classical MIP formulations introduced in chapter 2.1.

For our tests, we are using a small network that consists of 2 entry and 38 exit nodes, 40 pipes and 2 compressor stations and control valves, respectively. Compared to real-world gas networks, its structure is very simple as can be seen from figure 18.

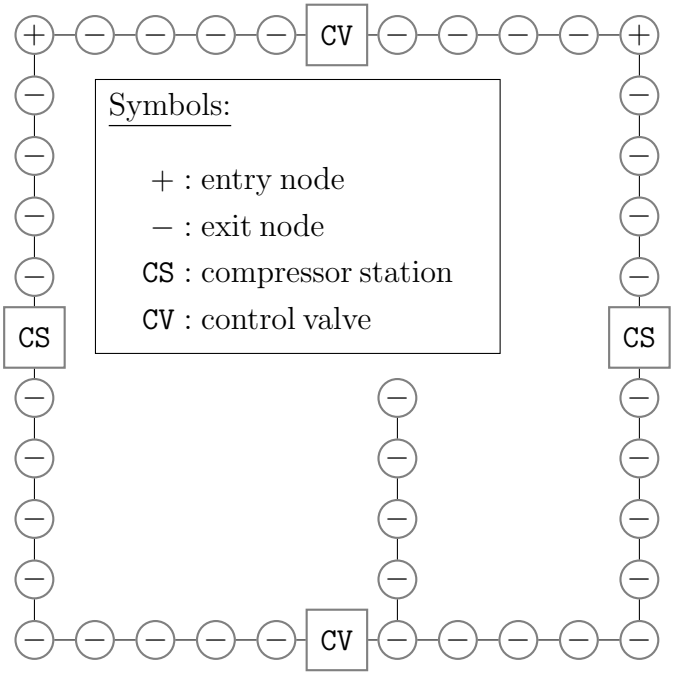


Figure 18: Example network.

The test set that we are working with contains 15 different scenarios, i.e., specific flow nominations at all nodes. For the generation of the mathematical formulations, we have relied on the LAMATTO++ framework for modelling and solving mixed-integer non-linear programming problems on networks, developed at Erlangen-Nürnberg University [12]. The theory on how the piecewise linear approximations of the non-linear functions are computed can be found in [14, 15]. When translated into a mathematical model and after some internal LAMATTO++ preprocessing that exploits certain network structures to reduce the modelling complexity, each of the considered scenarios yields 40 piecewise linear constraints originating from the approximation of the Weymouth equation for all pipes. On average, the approximations consist of 36 support nodes. As objective function, we have opted for a pressure loss minimization.

First of all, in table 2 we want to compare the average number of variables and constraints in our models. Throughout this section, let our constraint programming approach using convex hull relaxations be denoted by (PWL), while the MIP models will be referred to by their abbreviations as introduced.

	(INC)	(CC)	(MC)	(INC$_{log}$)	(PWL)
Variables	2419	2571	2507	2703	742
- binary	883	971	971	260	104
- continuous	1536	1600	1536	2443	638
Constraints	3939	2589	3168	1874	1443
- PWL	2536	1186	1765	471	40
- others	1403	1403	1403	1403	1403

Table 2: Mean number of variables and constraints in test instances.

As expected, all non-logarithmic formulations have roughly the same number of variables, whereas (INC$_{log}$) uses significantly less binaries. However, this goes along with an increase in continuous variables. With its direct modelling of piecewise linear constraints, our model needs the fewest variables. Above all, binary variables are only used for modelling discrete decisions as part of the nomination validation problem and not as auxiliary variables. Similarly, many additional constraints are needed for representing piecewise linear functions in the MIP models, which is not the case for (PWL).

Next, we want to study the running times of the MIP instances. SCIP is very well suited for solving pure mixed-integer programs without further constraint types, since it provides sophisticated methods for all aspects of a branch-and-bound procedure. In particular, it features a very powerful presolving process. Table 3 lists the statistics of running the MIP instances, both with presolving enabled and disabled.

	(INC)	(CC)	(MC)	(INC$_{log}$)
(1) Presolving enabled (unlimited rounds)				
total solving time [s]	7.0	19.8	29.9	11.4
total solving nodes	25.5	592.4	345.5	749.6
presolving time [s]	2.5	1.2	4.0	0.2
- binary vars fixed [%]	77.6	16.9	45.6	22.3
- constraints removed [%]	79.1	34.2	47.5	31.3
(2) Presolving disabled				
total solving time [s]	14.9	17.2	21.8	8.9
total solving nodes	320.8	717.4	522.2	574.5

Table 3: Computational results of MIP test instances. The values shown are averages over all test instances, with the geometric mean being used for solving time and nodes in order to weaken the influence of outliers.

It can be seen from the table that the (INC) formulation and its logarithmic variant clearly yield the best results with respect to total running time. Surprisingly, (INC$_{log}$) thereby does not generate less branch-and-bound nodes than (CC) and (MC). The comparatively bad performance of (CC) may be attributed to the formulation being not locally ideal, whereas there is no obvious reason for (MC) being the slowest model. The major insight, however, is that presolving seems to work extraordinary well for (INC), reducing the size of the models by more than $\frac{3}{4}$. This leads to a drastic decrease of the required solving nodes compared to the other formulations and largely contributes to the low amount of time that is spent after presolving was finished. By contrast and quite unexpectedly, presolving slightly negatively influences the average running times of (CC) and (MC) despite reducing the number of nodes, which is outweighed by an increase in the number of required LP iterations.

We now want to proceed with stating results obtained by our piecewise linear constraint handler. Since there are multiple options in `SCIP_Conshdlrdata` that allow for controlling different aspects of the solving process, we can not experiment with every thinkable combination of these but try to test them as independent parameters. The following parameter values will be considered as default settings that are used if not explicitly stated otherwise:

```
addcutsinit       TRUE
splitcons         TRUE
sepallcons        FALSE
updateconslocal   TRUE
branchmethod      2-CHILD-EXACT
branchcandsel     MOSTINFEAS
MAXPREROUNDS      10
```

The largest impact on the outcome, as we have found, is caused by the rule for selecting a branching variable. In table 4 we will distinguish between two cases:

- `FRACPRIO`: If there are fractional binary variables at some node, always let the integrality constraint handler create a branching on one of them before piecewise linear constraints are considered.
- `JOINT`: Jointly consider fractional variables and violated piecewise linear constraints as branching candidates and score them according to comparable branching rules.

We will omit the case of prioritizing piecewise linear constraints since some of the binary variables represent true network decisions like, for example, opening or closing a valve. It does not make sense to take these decisions only after each piecewise linear constraints has been satisfied.

	FRACPRIO		JOINT	
	time [s]	nodes	time [s]	nodes
RANDOM	2.9	46.1	2.8	68.5
MOSTINFEAS	2.9	47.8	2.8	44.0
LEASTINFEAS	2.9	48.6	3.1	91.6
STRONG	51.4	37.1	133.7	27.9
RELPSCOST-3	14.1	37.5	34.5	32.3
RELPSCOST-6	24.6	37.2	42.6	29.9

Table 4: Computational results of (PWL) test instances by branching rule. Again, the geometric mean has been used. In `RELPSCOST-X` branching, the value of `X` denotes the reliability parameter η.

As we can see, the best results in terms of running time have been obtained by the branching rules `RANDOM`, `MOSTINFEAS` and `LEASTINFEAS` with their average solving times being nearly equal. Furthermore, the instances were solved faster than by any MIP formulation. There is, however, a problem of scale, since none of these three branching rules yield acceptable results when applied to more complex real-world instances. They all result in huge branch-and-bound trees without finding good solutions. Hence, in order to keep the number of solving nodes as small as possible, we need to rely on some strong branching approach. The values in the table show that the size of the branching tree can indeed be reduced by using `STRONG` or `RELPSCOST-X` branching. Moreover, we can see that a joint strong or reliability branching evaluation of fractional variables and piecewise linear constraints is advantageous to a separate consideration based on prioritizing either one above the other. In our tests, when there were both types of branching candidates available at some node, the ratio of joint strong branch-

ing choosing a fractional variable to choosing a piecewise linear constraint was about 53.1% to 46.9%. The main drawback that, until now, prevents us from solving larger instances is the lacking infrastructure of the SCIP framework for performing strong branching on piecewise linear constraints. As described above, SCIP is not capable of performing warm-started LP iterations after a tentative cut insertion, which would be required for computing the convex hull relaxations of potential child nodes. The detour that we have implemented excessively slows down the solving process even in our relatively small test instances. In part, this can be remedied by reliability pseudocost branching that yields comparable results to strong branching in less time, without, however, leading to satisfactory results in real-world examples.

Finally, in table 5 we want to show the impact of running the test instances with parameter values different from the default settings. The most interesting insight is that the branching method `3-CHILD` yields a decrease in both running time and number of solving nodes compared to `2-CHILD-EXACT`. By contrast, the solution process is slower and needs considerably more nodes when either `2-CHILD-LEFT` or `2-CHILD-RIGHT` is used, which may be ascribed to the fact that both of these methods do not guarantee the LP solution to be excluded from all of the generated subproblems. Likewise, all other parameter value changes do not lead to an enhancement of the solving process, with a particularly bad performance being caused by not splitting the constraints into their upper- and lower bounded parts.

parameter	value	time [s]	change	nodes	change
	`DEFAULT`	2.8	-	44.0	-
`addcutsinit`	`FALSE`	2.9	+1.7%	46.9	+6.6%
`splitcons`	`FALSE`	3.2	+12.9%	53.8	+22.2%
`sepallcons`	`TRUE`	2.8	-0.8%	46.5	+5.8%
`updateconslocal`	`FALSE`	2.8	-1.3%	54.6	+24.1%
`branchmethod`	`2-CHILD-LEFT`	3.1	+11.0%	59.0	+34.0%
	`2-CHILD-RIGHT`	2.9	+2.8%	58.5	+32.9%
	`3-CHILD`	2.6	-7.8%	40.7	-7.4%

Table 5: Computational results of (PWL) parameter tests. The percental changes are computed with respect to the default settings given above.

5 Summary

In this thesis, we have studied different modelling approaches for piecewise linear functions, in particular with respect to their occurrence in mathematical optimization problems. Reviewing the quality of the classical MIP models from the literature, we have seen that all but the disaggregated convex combination method yield both sharp and locally ideal formulations. In the main part, we have presented an alternative approach for modelling constraints involving piecewise linear functions, based on ideas of constraint programming and using a convex hull relaxation. For the purpose of developing a branch-and-bound procedure specifically adapted to our constraint formulation, all required components such as propagation, branching and separation have been discussed. In particular, we have focussed on adapting well-known MIP strategies for the selection of branching variables to our type of piecewise linear constraints.

For the special case of univariate piecewise linear functions, we have implemented a constraint handler within the SCIP optimization framework. In order to test its performance with respect to various aspects, we have considered an application of piecewise linear constraints in gas network optimization where they are used as approximations to non-linear functions. It could be shown that, besides drastically decreasing the number of variables and constraints in the model, our CP approach may potentially reduce the number of nodes in the branch-and-bound tree compared to the standard MIP formulations. However, due to some unavailable SCIP functionality that would be required for taking reliable branching decisions in a reasonable amount of time, it has not been possible for us to solve large real-world problem instances. Nevertheless, we have gained the insight that it is preferable to jointly consider piecewise linear and fractional binary branching candidates instead of prioritizing one above the other, since either type may yield the locally best decision in terms of dual bound progress.

As a topic for future work, one might consider extending our constraint handler to more general classes of constraints, e.g., involving multivariate or semicontinuous piecewise linear functions. This would necessitate more complex data structures for storing the constraint information as well as modified constraint programming techniques for the branch-and-bound procedure. Concerning the development of the SCIP framework, it would be very useful to provide more flexible means for applying the idea of strong branching to a wider set of constraint types.

References

[1] Tobias Achterberg. "Constraint Integer Programming". PHD thesis. Berlin, Germany: Technische Universität Berlin, July 2007.

[2] Tobias Achterberg, Thorsten Koch, and Alexander Martin. "Branchin Rules Revisited". In: *Operations Research Letters* 33 (2005), pp. 45–54.

[3] A. M. Andrew. "Another Efficient Algorithm for Convex Hulls in Two Dimensions". In: *Information Processing Letters* 9 (1979), pp. 216–219.

[4] A. Balakrishnan and S. C. Graves. "A Composite Algorithm for a Concave-Cost Network Flow Problem". In: *Networks. An International Journal* 19 (2 1989), pp. 175–202.

[5] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The Quickhull Algorithm for Convex Hulls". In: *ACM Transactions on Mathematical Software* 22.4 (1996), pp. 469–483.

[6] Pietro Belotti et al. "Branching and Bounds Tightening Techniques for Non-Convex MINLP". In: *Optimization Methods and Software* 24.4-5 (2009), pp. 597–634.

[7] M. Benichou et al. "Experiments in Mixed-Integer Linear Programming". In: *Mathematical Programming* 1.1 (1971), pp. 76–94.

[8] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Berlin-Heidelberg: Springer, 2008.

[9] Timothy M. Chan. "Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions". In: *Discrete and Computational Geometry* 16 (1996), pp. 361–368.

[10] K. L. Clarkson and P. W. Shor. "Applications of Random Sampling in Computational Geometry II". In: *Discrete Computational Geometry* 4 (1989), pp. 387–421.

[11] George B. Dantzig. "On the Significance of Solving Linear Programming Problems with Some Integer Variables". In: *Econometrica. Journal of the Econometric Society* 28 (1960), pp. 30–44.

[12] Friedrich-Alexander Universität Erlangen-Nürnberg. *LAMATTO++ - A Framework for Modeling and Solving Mixed-Integer Nonlinear Programming Problems on Networks*. Oct. 2012. URL: `http://www.mso.math.fau.de/edom/projects/lamatto.html`.

[13] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. 2nd ed. Duxbury, 2002.

[14] Björn Geissler. "Towards Globally Optimal Solutions for MINLPs by Discretization Techniques". PHD thesis. Erlangen-Nürnberg, Germany: Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2011.

[15] Björn Geissler et al. “Using Piecewise Linear Functions for Solving MINLPs”. In: *Mixed Integer Nonlinear Programming*. Ed. by Jon Lee and Sven Leyffer. The IMA Volumes in Mathematics and its Applications. New York, USA: Springer, 2012, pp. 287–314.

[16] Ralph E. Gomory. “Outline of an Algorithm for Integer Solutions to Linear Programs”. In: *Bulletin of the American Mathematical Society* 64.5 (1958), pp. 275–278.

[17] Ronald L. Graham. “An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set”. In: *Information Processing Letters* 1 (1972), pp. 132–133.

[18] John N. Hooker. “Semicontinuous Piecewise Linear Functions”. In: *Integrated Methods for Optimization*. New York, USA: Springer, 2007, pp. 259–260.

[19] Konrad-Zuse-Zentrum für Informationstechnik Berlin. *SCIP - Solving Constraint Integer Programs*. June 2013. URL: `http://scip.zib.de/`.

[20] Robert G. Jeroslow and James K. Lowe. “Modeling with Integer Variables”. In: *Mathematical Programming Studies* 22 (1984), pp. 167–184.

[21] Ravi Kannan. “Lattice Translates of a Polytope and the Frobenius Problem”. In: *Combinatorica* 12 (1992), pp. 161–177.

[22] Ahmet B. Keha, Ismael R. de Farias Jr., and George L. Nemhauser. “A Branch-and-Cut Algorithm Without Binary Variables for Nonconvex Piecewise Linear Optimization”. In: *Operations Research* 54 (2006), pp. 847–858.

[23] Ahmet B. Keha, Ismael R. de Farias Jr., and George L. Nemhauser. “Models for Representing Piecewise Linear Cost Functions”. In: *Operations Research Letters* 32 (2004), pp. 44–48.

[24] H. M. Markowitz and A. S. Manne. “On the Solution of Discrete Programming Problems”. In: *Econometrica. Journal of the Econometric Society* 25 (1957), pp. 84–110.

[25] Mark H. Overmars and Jan van Leeuwen. “Maintainance of Configurations in the Plane”. In: *Journal of Computer and System Sciences* 23 (1981), pp. 166–204.

[26] Manfred Padberg. “Approximating Separable Nonlinear Functions via Mixed Zero-One Programs”. In: *Operations Research Letters* 27 (2000), pp. 1–5.

[27] Manfred Padberg and M. Rijal. *Location, Scheduling, Design and Integer Programming*. Boston: Kluwer Academic Publishers, 1996.

[28] Marc E. Pfetsch et al. “Validation of Nominations in Gas Network Optimization: Models, Methods and Solutions”. In: *ZIB-Report* 41 (2012).

[29] Franco P. Preparata. “An Optimal Real-Time Algorithm for Planar Convex Hulls”. In: *Communications of the ACM* 22 (1979), pp. 402–405.

[30] Michael I. Shamos. “Computational Geometry”. PHD thesis. New Haven, Connecticut, USA: Yale University, May 1978.

[31] John A. Tomlin and Martin L. Beale. “Special Facilities in a General Mathematical Programming System for Non-Convex Problems Using Ordered Sets of Variables”. In: *Proceedings 5th IFORS Conference*. Ed. by J. Lawrence. London, UK: Tavistock Publications, 1970, 447–454.

[32] Juan Pablo Vielma, Shabbir Ahmed, and George L. Nemhauser. “Mixed-Integer Models for Nonseparable Piecewise-Linear Optimization: Unifying Framework and Extensions”. In: *Operations Research* 58 (2010), pp. 303–315.

[33] Juan Pablo Vielma and George L. Nemhauser. “Modelling Disjunctive Constraints with a Logarithmic Number of Binary Variables and Constraints”. In: *Mathematical Programming* 128 (2011), pp. 49–72.