CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Reliable characterization of the existence of solutions of parametric systems of equations
Student:	Bc. Libor Vytlačil
Supervisor:	doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

The goal is to develop an algorithm and software to characterize the set of points p for which there is a real-valued solution x to a parametric system of equations F(p, x)=0. The algorithm should be based on floating-point arithmetic and robust wrt rounding errors.

1) Manually compute a few examples applying a solution existence test to systems of non-linear equations based on the topological degree [1].
2) Design an algorithm for the characterization of the existence of solutions of parametric systems of equations. To this end, combine the existence test with an algorithm for computing a sub-paving (i.e., decomposition into Cartesian products of intervals) of the parameter space [3].
3) Implement the resulting algorithm together with a user interface.
4) Test the implementation, e.g., based on systems of equations formed by intersections of ellipsoids and hyper-surfaces.
5) Document the results.

References

[1] Peter Franek, and Stefan Ratschan: Effective Topological Degree Computation Based on Interval Arithmetic, Mathematics of Computation, Volume 84, 2015, pp. 1265-1290.
[2] Peter Franek, Stefan Ratschan, and Piotr Zgliczynski: Quasi-decidability of a Fragment of the First-order Theory of Real Numbers. Journal of Automated Reasoning, Volume 72, 2016, pp. 157-185.
[3] Luc Jaulin et. al.: Applied Interval Analysis, Springer, 2001.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 7, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Reliable characterization of the existence of solutions of parametric systems of equations

Bc. Libor Vytlačil

Supervisor: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

14th February 2018

Acknowledgements

I would like to thank my supervisor doc. Dipl. Ing. Dr. techn. Stefan Ratschan for helpful consultations, his guidance and all the advice he offered me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 14th February 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Vytlačil, Libor. *Reliable characterization of the existence of solutions of parametric systems of equations.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá charakterizací množin bodů $p \in P$, pro které existuje reálné řešení x dané parametrické soustavy rovnic $f_P(x, p) = 0$, kde $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ je spojitá funkce zapsaná ve formě aritmetických výrazů a X je box (součin uzavřených reálných intervalů).

Je navržen a implementován algoritmus typu branch and bound založený na intervalové aritmetice a testu existence řešení pomocí Brouwerova topologického stupně. Implementace umožňuje několik způsobů manipulace s doménou X a další parametrizace své činnosti. Intervalová aritmetika slouží jednak k výpočtu odhadů obrazů funkcí, jednak k zajištění robustnosti vzhledem k zaokrouhlovacím chybám. Implementace je poskytnuta v podobě samostatné aplikace včetně uživatelského rozhraní (textového i grafického).

Na základě výsledků experimentů, které jsou uloženy ve formě skriptů pro snadnou reprodukovatelnost, je naše implementace schopna produkovat kvalitní výsledky a skončit v rozumném čase pro různorodé nelineární soustavy až o třech rovnicích, které mohou obsahovat kombinace elementárních funkcí jako exp, log, sin, cos, arcsin, arccos, n-tá odmocnina, ...

Přispíváme také k dosavadní práci týkající se praktického počítání topologického stupně popisem a použitím vlastní spolehlivé metody pro stanovení, zda pro libovolné boxy $X' \subseteq X$, $P' \subseteq P$ je definován stupeň $\deg(f_p, X', 0)$ a má pro každý parametr $p \in P'$ stejnou hodnotu.

Klíčová slova intervalová aritmetika, topologický stupeň, existence kořene, soustava nelineárních rovnic, dekompozice prostoru parametrů, Gaol, branch and bound

Abstract

This thesis deals with the problem of characterizing the sets of points $p \in P$, for which there exists a real-valued solution x to a given parametric system of equations $f_P(x, p) = 0$, where $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function formed by arithmetical expressions and X is a box (product of closed real intervals).

A branch and bound like algorithm based on interval arithmetics and a solution existence test using Brouwer's topological degree is designed and implemented. It allows several ways of manipulating the domain X and further parametrization of its operation. Interval arithmetics is used for computing estimates of function images, as well as for providing robustness with respect to rounding errors. The implementation is provided in a form of self-contained application including a user interface (both text-based and graphical).

Based on results of the experiments, that are stored in scripts for an easy reproducibility, our implementation is capable of producing quality results and terminate in reasonable time for various non-linear systems up to three equations, possibly containing combinations of elementary functions like exp, log, sin, cos, arcsin, arccos, n-th root, ...

We also enhance previous work related to the practical computation of the topological degree by describing and using a custom sound method of determining, if for arbitrary boxes $X' \subseteq X$, $P' \subseteq P$, the degree $\deg(f_p, X', 0)$ is defined and has the same value for every $p \in P'$.

Keywords interval arithmetics, topological degree, root existence, system of non-linear equations, parameter space decomposition, Gaol, branch and bound

Contents

List of Figures

List of Tables

Introduction

Thesis goals

In engineering practice, there often occur systems of equations containing uncertain parameters. It is then desirable to focus specifically on those parameter values, for which the system has a solution. This requires a method, that can explore the parameter space and localize such values.

In the scope of this thesis, we focus on designing, implementing and experimentally evaluating an algorithm, that allows characterizing points p of such parameter space P. We consider the input equation system to be of the form $f_P(x, p) = 0$, where $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function formed by arithmetic expressions (generally non-linear) and X is a box (i.e. cartesian product of closed real intervals).

Our aim is to provide an implementation based on floating point arithmetic and ensure its robustness with respect to rounding errors by designing the algorithm's computations to make use of interval arithmetic. The use of interval arithmetic also allows us to easily obtain sound estimates of function images.

We should identify the pieces of the algorithm that are likely to have the most influence on the solution quality, and allow them to be parametrized or specialized in specific manners. We want to accompany the implementation with a user interface, preferably with some option of visualizing results.

An important part of our algorithm is a solution existence test, that should be carried out using the properties of Brouwer's topological degree [1, Chap. 1]. To compute the topological degree, we want to make use of an existing algorithm described in [2]. But its application to our problem domain is not fully direct, mainly because we typically need to start the degree computation without actually knowing, whether the degree is defined for the given input. In addition, because the input function is parametrized, we are actually looking for the degree with respect to a whole set of functions. We therefore focus on adjusting the flow of [2] to suite our needs. It is mainly for this reason, that

we also aim to provide our custom implementation of [2] (the other important point is, that we want it to use the same libraries and frameworks as the rest of our implementation).

In the end, we seek to test the correctness of our implementation and write a series of reproducible experiments to test its performance.

Thesis structure

Chapter 1 provides the theoretical background related to the solved problem, focusing mostly on interval arithmetic and topological properties of boxes.

Chapter 2 follows with describing the used algorithm for computing the topological degree [2], supplied by several by-hand computed examples. There, we also show how to adjust its flow to suit our needs and discuss its usage with parametrized functions in a way we came up with.

Chapter 3 is dedicated to the development of our solving algorithm. We start with general ideas followed by formulating its general workflow. After that, possible specialization of its core components, that we designed, are discussed in detail.

Chapter 4 and 5 are aimed at highlighting the relevant points regarding the design and implementation of our application, that encapsulates our main algorithm. The former presents the tools and code libraries we decided to use, the latter talks about the design choices and implementation ideas.

Chapter 6 presents the results of our experimental evaluation and describes the way we tested correctness of certain parts of our whole implementation.

In Appendix A, we explain how to run and use our provided GUI.

Chapter 1

Theoretical background

This chapter introduces notions used in the rest of the next. They are related mainly to interval arithmetic, intervals, boxes and their properties, and the topological degree. Newly introduced terms are typeset in italics. We also provide several by-hand computed examples to better explain some of the ideas and concepts.

1.1 Floating point numbers

Floating point numbers provide a way to represent finite subsets of rational numbers. They can be expressed in the following notation

$$(-1)^s \cdot m \cdot \beta^e,$$

with integer *base* $\beta \geq 2$, fixed length *significand* m, fixed length integer *exponent* e and *sign* $s \in \{0, 1\}$. If the base β is given, such numbers can be simply written as triplets (s, m, e), [3, pp. 13–14].

Floating point numbers representation in computer environments is standardized by IEEE 754 technical standard [4], which specifies their possible decimal and binary formats, arithmetic, interchange encodings, exceptions and their handling, and more.

Given a precision p and an exponent length in bits w, binary formats are encoded as bit strings consisting of three parts.

- Sign bit S.
- Sequence of bits $E = E_1 E_2 \dots E_w$, called biased exponent.
- Sequence of bits $M = M_1 M_2 \dots M_{p-1}$, called trailing significand.

If we define *bias* $b = 2^{w-1} - 1$, most of the triplets (S, M, E) are then uniquely mapped to $x = (s, m, e)$ in the following manner.[1,2]

- If $0 < |E| < 2^w - 1 \wedge M \neq 0$, then normal number $x = (-1)^S \cdot (1 + M) \cdot 2^{|E|-b}$ is represented.
- If $|E| = 0 \wedge M \neq 0$, then subnormal number $x = (-1)^S \cdot (0 + M) \cdot 2^{-b+1}$ is expressed.

The rest of the bit strings are reserved to represent special values.

- Triplets with $E = 0 \wedge M = 0$ represent $+0$ for $S = 0$ and -0 for $S = 1$.
- Triplets with $E = 2^w - 1 \wedge M = 0$ represent $+\infty$ for $S = 0$ and $-\infty$ for $S = 1$.
- Triplets with $E = 2^w - 1 \wedge M \neq 0$ represent values called NaNs (not a number).

Example 1.1. Consider a set F of binary floating-point numbers in the format given by $p = 2, w = 3$. The exponent bias is $b = 2^{w-1} - 1 = 3$. Apart from $\pm 0, \pm\infty$ and NaNs, the set F contains following positive normal numbers

$$\{\frac{1}{4}, \frac{3}{8}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 3, 4, 6, 8, 12\},$$

their negative counterparts and two subnormal numbers $-\frac{1}{8}, \frac{1}{8}$. Some of these numbers are depicted in Figure 1.1. Table 1.1 also shows corresponding binary strings (S, M, E) for several numbers. □

Figure 1.1: Floating point numbers on the real axis. (Example 1.1).

One of the advantages of floating-point representation lies in its universality. The format's precision is given, but it is up to us how we want to spend it, i.e. on working with small numbers with small absolute roundoff errors or on bigger numbers with bigger absolute roundoff errors.

[1]Notice how the first binary digit of m is determined by the value of E and thus does not need to be stored. This shows why M is referred to as trailing significand and that only $p - 1$ bits are needed to store p-bit precision.

[2]Storing e biased as $E = e + b$ is done so that both positive and non positive values of e can be stored without the need of using a signed integer representation for E.

S	M	E	s	m	e	$(-1)^s \cdot m \cdot 2^e$
0	0	001	0	1.0	-2	$\frac{1}{4}$
0	0	011	0	1.0	0	1
0	1	011	0	1.1	0	$\frac{3}{2}$
1	1	100	1	1.1	1	-3
0	1	000	0	0.1	-2	$\frac{1}{8}$

Table 1.1: Mapping of binary strings (S, M, E) to floating point numbers from Example 1.1.

1.1.1 Rounding

When performing an operation with floating-point numbers from a given set F, its exact result may not belong to F and needs to be rounded, i.e. substituted by a member of F. A way of selecting such member is called a *rounding mode*. We can think of rounding modes as functions $\mathbb{R} \to F$. Four common rounding modes, based on IEEE 754 [4] specification, are:

- Rounding down (toward $-\infty$); $\delta(x) = \max\{y \in F; y \leq x\}$.
- Rounding up (toward $+\infty$); $\Delta(x) = \min\{y \in F; y \geq x\}$.
- Rounding toward zero; $\sigma(x) = \delta(x)$ for $x \geq 0$ and $\sigma(x) = \Delta(x)$ otherwise.
- Rounding to nearest even; $\varphi(x)$ is the floating-point number closest to x. If two floating-point numbers are equally close to x, the one whose representation has even trailing significand (i.e. its LSB is 0) is chosen.

Example 1.2. Consider the set F of floating-point numbers from the Example 1.1. Then $\delta(\frac{1}{2} + \frac{3}{8}) = \delta(\frac{7}{8}) = \frac{3}{4}$, $\Delta(\frac{7}{8}) = 1$ and $\sigma(\frac{7}{8}) = \frac{3}{4}$. Number $\frac{7}{8}$ is equally distant from $\frac{3}{4}$ and 1, therefore $\varphi(\frac{7}{8}) = 1$, because 1 has even trailing significand (see Figure 1.1).

Similarly, $\delta(-1-\frac{3}{8}) = \delta(-\frac{11}{8}) = -\frac{3}{2}$, $\Delta(-\frac{11}{8}) = -1$, $\sigma(-\frac{11}{8}) = -1$. Closest number to $\varphi(-\frac{11}{8})$ from F is $-\frac{3}{2}$ and so $\varphi(-\frac{11}{8}) = -\frac{3}{2}$. □

The rounding modes Δ and δ are called *directed rounding modes*. They are typically used for implementing sound interval arithmetic. As [3, Sec. 3.5] explains, a rounding mode is typically specified globally for all subsequent floating-point instructions via a control register. Setting it to a different value requires carrying out extra instructions and causes the floating-point pipeline to flush.

Directed rounding modes can also be mimicked programmatically to a certain extent, by computing successors and predecessors of floating-point numbers via bit manipulation, see [5, p. 3–4].

Example 1.3. In C language (assuming C99 support), one possibility of explicitly switching rounding modes is via the floating-point environment, by including the `<fenv.h>` header, see [6]. Current rounding mode can be retrieved via `fegetround`, and can be switched via `fesetround`. Consider the following sample code.

```
#include <fenv.h>
#include <stdio.h>
#pragma STDC FENV_ACCESS ON

int main() {
const int originalRounding = fegetround();
float c = 0.1;

fesetround(FE_UPWARD);
int i;
float d1 = 0.0;
for (i = 0; i < 2000; ++i) {
        d1 += c;
}

fesetround(FE_DOWNWARD);
float d2 = 0.0;
for (i = 0; i < 2000; ++i) {
        d2 += c;
}

printf("d1 = %f\n", d1);
printf("d2 = %f\n", d2);

fesetround(originalRounding);
return 0;
}
```

Compiling this code on Ubuntu 16.04.3 using gcc 5.4.1 with flags `-O2 -std=c99 -lm`, we obtain the output

```
d1 = 200.009155
d2 = 199.991653
```

Also, when disassembling the resulting object file, we find out, that calls to `fesetround` and `fegetround` are not inlined, see below.

```
00000000004005b0 <main>:
4005b0: 53                  push   %rbx
4005b1: 48 83 ec 10         sub    $0x10,%rsp
```

```
4005b5: e8 b6 ff ff ff     callq  400570 <fegetround@plt>
4005ba: bf 00 08 00 00     mov    $0x800,%edi
4005bf: 89 c3              mov    %eax,%ebx
4005c1: e8 9a ff ff ff     callq  400560 <fesetround@plt>
4005c6: 66 0f ef c0        pxor   %xmm0,%xmm0
4005ca: b8 d0 07 00 00     mov    $0x7d0,%eax
...
```

This implies, that this way of rounding mode accessing causes extra overhead associated with performing function calls. □

1.2 Real intervals and boxes

In this section, we define real intervals, boxes and interval functions. Chosen notations listed in this section are mostly based on [7] and [8].

Closed real intervals are sets of the form

$$I = [\underline{x}, \overline{x}] = \{y \in \mathbb{R} \,|\, \underline{x} \leq y \leq \overline{x}\}. \tag{1.1}$$

We refer to them simply as *intervals* through the rest of the text and denote the set of all intervals by $I(\mathbb{R})$. Elements $\underline{x}, \overline{x}$ are called *endpoints* of X and if $\underline{x} = \overline{x}$, then X is a *degenerated interval* and we may denote it by $[\underline{x}]$ instead of $[\underline{x}, \overline{x}]$

A generalization of an interval is a *box* in $\mathbb{R}^n$, which we define for $n \geq 0$ as

$$X = I_1 \times I_2 \times \cdots \times I_n, \; I_i \in I(\mathbb{R}) \tag{1.2}$$

and denote the set of all boxes in $\mathbb{R}^n$ by $I(\mathbb{R}^n)$. Interval I_i is the i-th *component* of X and an *endpoint* of X is any endpoint of its components. Every interval itself is a box in $\mathbb{R}^1$ and vice versa, i.e. $I(\mathbb{R}) = I(\mathbb{R}^1)$.

We also make a use of the notion of *width* defined respectively for an interval $I = [\underline{x}, \overline{x}]$ and a box $X = I_1 \times \cdots \times I_n$ as

$$\omega(I) = \omega([\underline{x}, \overline{x}]) = \overline{x} - \underline{x}, \tag{1.3}$$

$$\omega(X) = \max_i \{\omega(I_i)\}. \tag{1.4}$$

1.2.1 Interval functions

Any function $F\colon I(\mathbb{R}^n) \to I(\mathbb{R}^m)$ is called an *interval function*. The image of a given box $X \in I(\mathbb{R}^n)$ under F is simply denoted by $F(X)$. In addition, for a point $x = (x_1, \ldots, x_n)$ in $\mathbb{R}^n$, we assign the meaning of the notion $F(x)$ to be the same as for $F([x_1] \times \cdots \times [x_n])$.

We are interested in the following three properties interval functions can have. F is said to be *inclusion monotonic*, if

$$\forall X, Y \in I(\mathbb{R}^n).\, X \subseteq Y \implies F(X) \subseteq F(Y), \tag{1.5}$$

and it is called an *interval extension* of function $f\colon D \subseteq \mathbb{R}^n \to \mathbb{R}^m$, if

$$\forall x \in D.\, f(x) = F(x). \tag{1.6}$$

F is further said to be *convergent* on $D \subseteq \mathbb{R}^n$, if for every sequence of boxes X_k from D,

$$\lim_{k\to+\infty} \omega(X_k) = 0 \implies \lim_{k\to+\infty} \omega(F(X_k)) = 0. \tag{1.7}$$

1.3 Interval arithmetic

Interval arithmetic is a system of computation with intervals and boxes via interval functions designed to provide a way of reliable numerical calculation. Inputs are packed into boxes, that bound possible input uncertainities, and all functions used during computation are defined, so that the resulting box is guaranteed to contain the exact punctual value of the result.

For our purposes, we like to look at interval arithmetic as a system, that extends every function f of the form $f\colon D \to \mathbb{R}^m, D \subseteq \mathbb{R}^n$, from the given set of functions Ω, to an interval function $F\colon I(\mathbb{R}^n) \to I(\mathbb{R}^m)$, while ensuring the *inclusion property*

$$\forall X.\, X \in I(\mathbb{R}^n) \implies F(X) \supseteq f(X \cap D). \tag{1.8}$$

A function F with this property is called an *inclusion function* of f.

It makes the most practical sense, when Ω contains standard real arithmetic operations $+, -, *, /$ and real elementary functions, and when their inclusion functions are both

- as tight as possible, i.e. they do not overestimate $f(X \cap D)$ much,
- operational, which informally means that images $F(X)$ can be computed in a finite, bounded number of steps using functions from Ω.

Apart from the reliable way of numerical computations, inclusion functions are then used to perform a reliable test, that a point $x \in \mathbb{R}^n$ does not belong to the image $f(X)$ of a given $X \subseteq D$, using the fact that

$$x \notin F(X) \implies x \notin f(X). \tag{1.9}$$

It is sufficient for an interval function F to be an inclusion function of a punctual function f, if properties (1.5) and (1.6) are satisfied for F and f, see [8, p. 47]. This is also called *The fundamental theorem of interval arithmetic*.

The property (1.6) lets us view the interval computation as a true extension to punctual computations in the sense, that we do not need to make difference between computing with points from $\mathbb{R}^n$ and degenerated intervals from $I(\mathbb{R}^n)$.

Property (1.5) in combination with (1.7) is important for iterative methods, that start off with some box X, and keep splitting it into smaller boxes and evaluating them, in order to prove some property of the original box X. The property (1.7) itself counterweights the overestimation tendency, that interval arithmetic computations might suffer from, as discussed in Subsection 1.3.3. The property (1.5) is then useful, if a given iterative method involves the property (1.9), as showing that $x \notin F(X)$ then implies $x \notin F(Y)$ for every $Y \subseteq X$.

1.3.1 Extending arithmetic operations and elementary functions

Here we use [7] and [8] to show the way standard binary operations $+, -, *, /$ and basic elementary functions are extended, so that their inclusion functions have properties (1.5), (1.6) and (1.7). In addition, presented inclusion functions are *minimal*, meaning that for any input X, they always produce the smallest box containing the image of X under the corresponding punctual function.

Let X and Y denote intervals $[\underline{x}, \overline{x}]$ and $\left[\underline{y}, \overline{y}\right]$, respectively. Standard operations $+, -, *, /$ are extended as

$$\begin{aligned} X + Y &= \left[\underline{x} + \underline{y}, \overline{x} + \overline{y}\right], \\ X - Y &= \left[\underline{x} - \overline{y}, \overline{x} - \underline{y}\right], \\ X \cdot Y &= \left[\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})\right], \\ X/Y &= X \cdot \left[\frac{1}{\overline{y}}, \frac{1}{\underline{y}}\right], \text{if } 0 \notin Y. \end{aligned} \tag{1.10}$$

If f is a real function in one variable, defined and monotonic on an interval $D \subseteq \mathbb{R}$, it can be extended simply as

$$F(X) = \begin{cases} [f(\underline{x}), f(\overline{x})] & f(\underline{x}) \leq f(\overline{x}), \\ [f(\overline{x}), f(\underline{x})] & f(\underline{x}) > f(\overline{x}). \end{cases} \tag{1.11}$$

In this manner, elementary functions $\exp, \log, \arcsin, \arccos, \arctan$, n-th root and odd power x^n are extended, each on its respective domain.

Non-monotonic functions often need to be dealt with individually. For example, the even power x^n is extended in the following way;

$$X^n = \begin{cases} [(\underline{x})^n, (\overline{x})^n] & (\underline{x})^n \geq 0 \\ [(\overline{x})^n, (\underline{x})^n] & (\overline{x})^n \leq 0 \\ [0, \max\{(\underline{x})^n, (\overline{x})^n\}] & \text{otherwise} \end{cases} \tag{1.12}$$

Another examples are sin and cos functions, where the periodicity is taken into account. The result of $\cos(X)$ is defined to be an interval $[\underline{r}, \overline{r}]$, such that

$$\underline{r} = \begin{cases} -1 & \exists k \in \mathbb{Z} \mid (2k-1)\pi \in X, \\ \min(\cos(\underline{r}), \cos(\overline{r})) & \text{otherwise}, \end{cases} \tag{1.13}$$

$$\overline{r} = \begin{cases} 1 & \exists k \in \mathbb{Z} \mid 2k\pi \in X, \\ \max(\cos(\underline{r}), \cos(\overline{r})) & \text{otherwise}. \end{cases} \tag{1.14}$$

Extending sin is similar to extending cos, see [7, p. 22].

An important point here is, that the presented extensions of each of the discussed functions are computed very cheaply and actually operate only on endpoints of particular input intervals.

Example 1.4. Some basic interval computations performed using inclusion functions defined above are

$$\begin{aligned} [-2,3] + [4,9] &= [2,12]\,, & [-2,3] * [4,9] &= [-18,27]\,, \\ [-2,3] - [4,9] &= [-11,-1]\,, & [-2,3] \,/\, [4,9] &= \left[-\frac{1}{2}, \frac{3}{4}\right], \\ [-3,-2]^2 &= [4,9]\,, & [2,3]^2 &= [4,9]\,, \\ [-2,3]^2 &= [0,9]\,, & [-3,-2]^3 &= [-27,-8]\,, \\ [2,3]^3 &= [8,27]\,, & [-2,3]^3 &= [-8,27]\,, \\ \sin\left(\left[\frac{\pi}{2}, \frac{3\pi}{2}\right]\right) &= [-1,1]\,, & \cos\left(\left[\frac{\pi}{2}, \frac{3\pi}{2}\right]\right) &= [-1,0]\,. \end{aligned}$$

□

1.3.2 Natural inclusion functions

We will deal with equation systems of the form $f = 0, f\colon \mathbb{R}^n \to \mathbb{R}^n$, where each component f_i of f will be a composition of standard arithmetic operations $+, -, *, /$ and elementary functions $x^n, \sin, \cos, \exp, \log, \arcsin, \arccos, \ldots$.

An interval function F_i obtained by taking f_i and replacing each point variable x with an interval variable X and each operator and elementary function by a corresponding convergent inclusion monotonic interval extension function (like the ones from Subsection 1.3.1) is called the *natural inclusion function* of f_i, or just the *natural inclusion* of f_i. Furthermore if the component f_i is composed only from continuous functions, then the obtained interval function F_i is also covergent.

In addition, properties of the interval function $F = (F_1, \ldots, F_n)$ are determined by its components to certain extent. Namely, if all components F_i have any of the properties listed in 1.2.1, then so does F itself. Thanks to this, we also call such function F the *natural inclusion* of f.

Notice, that we did not define natural inclusions as unique entities (for a given punctual function f), because in general, different convergent inclusion monotonic interval extension functions can be used for the replacements. However, when referring to natural inclusions in the rest of text, we will always mean those formed of interval inclusions from the Subsection 1.3.1.

Example 1.5. Consider a function $f(x_1, x_2) = (\sin(x_1) + 1, x_1 + 2x_2)$ defined on $X = \mathbb{R}^2$, and the interval inclusions introduced in Subsection 1.3.1. Then the function

$$F(X_1, X_2) = (\sin(X_1) + 1, X_1 + 2X_2),$$

where $X_1, X_2 \in I(\mathbb{R})$, is the natural inclusion function of f. □

1.3.3 Expression evaluation

When evaluating interval functions, the way they are written as expressions matters, because not all identities known from real arithmetic hold for interval arithmetic. Most notably, instead of the common distributive law, we can only rely on a weaker property called *subdistributivity*, that can be expressed for real intervals X, Y, Z as

$$X \cdot (Y + Z) \subseteq X \cdot Y + X \cdot Z. \tag{1.15}$$

Furthermore, both addition and multiplication have no inverse elements for non-degenerate intervals.[3]

For a given expression, one can expect greater uncertainty of its result the more times the same variable X appears in the expression. Intuitively, interval arithmetic treats each occurrence of X independently of others and is unable to capture the fact that they all bound the same exact value x.

Example 1.6. Identities $x^3 = x \cdot x \cdot x$, $(x + y)^2 = x^2 + 2xy + y^2$ and $\sin 2x = 2 \sin x \cos x$ hold for every $x, y \in \mathbb{R}$, whereas corresponding interval inputs demonstrate the thought of the previous paragraph.

$$[-2, 3]^3 = [-8, 27]$$
$$[-2, 3] * [-2, 3] * [-2, 3] = [-6, 9] * [-2, 3] = [-18, 27]$$

$$([-1, 0] + [0, 1])^2 = [-1, 1]^2 = [0, 1]$$
$$[-1, 0]^2 + [2]\,[-1, 0]\,[0, 1] + [0, 1]^2 = [0, 1] + [-2, 0] + [0, 1] = [-2, 2]$$

$$\sin([2]\,[0, \pi/4]) = \sin[0, \pi/2] = [0, 1]$$
$$[2] \sin[0, \pi/4] \cos[0, \pi/4] = [2] \left[0, \sqrt{2}/2\right] \left[\sqrt{2}/2, 1\right] = \left[0, \sqrt{2}\right]$$

[3] Commutativity and associativity of addition and multiplication still holds in interval arithmetic.

As a part of our main algorithm in Chapter 3, we will generally find ourselves in a situation of having a punctual function f, its natural inclusion F and a box X, such that $0 \notin f(X)$, but $0 \in F(X)$, and will need to keep splitting X into smaller boxes (sub-boxes, see Section 1.4), until for each sub-box X_i, $0 \notin F(X_i)$. Although the convergence property (1.7) gives us a theoretical guarantee of success, the way F is written can still affect the and influence execution time (because of the "speed" of convergence), used memory and even the ability to keep splitting boxes, because it is impossible to represent infinitely small boxes with computer floating-point arithmetic.

1.3.4 Parametrized functions

We also work with functions that are parametrized by some parameter box $P \subseteq \mathbb{R}^m$. If f is such function, we usually add the subscript P to it, that is f_P, to express the parametrization. Such function can be seen as a set of functions $\{f_p \,|\, p \in P\}$, where f_p is a function obtained from f_P, by replacing its parameters with the concrete value p. For every $x \in X$, we define the value $f_P(x)$ as

$$f_P(x) = \bigcup_{p \in P} f(x, p). \tag{1.16}$$

We also need the notion of inclusion function F_P of parametrized function f_P. We simply say, that F_P is an *inclusion function* of f_P, if F_P is an inclusion function of f_p, for every $p \in P$. To construct such inclusion function F_P of f_P, we extend the process of constructing natural incusion function from Subsection 1.3.2.

1. We replace each point variable x_i in the prescription of f_P with interval variable X_i and each operator and elementary function with its corresponding convergent inclusion monotonic interval extension function.

2. We replace each parameter p_j with the j-th component of the parameter box P.

Example 1.7. Consider a function $f_P = (x_1 + p_1, x_1 + p_2 x_2)$ defined on $X = \mathbb{R}^2$, parametrized with $P = [0, 1] \times [0, 2]$. Then the function

$$F_P = (X_1 + [0, 1]\,, X_1 + [0, 2]\, X_2),$$

where $X_1, X_2 \in I(\mathbb{R})$, is an inclusion function of f_P. □

1.3.5 Interval arithmetic implementation

So far, we discussed interval arithmetic as a theoretical concept. Here, we briefly mention core ideas behind its implementation in computer systems.

Representing an interval in a computer system is as simple as storing its two endpoints, and for this purpose, floating-point numbers are typically used.

This of course means, that only a certain finite subset $I(F) \subsetneq I(\mathbb{R})$ can be represented. During a computation with intervals from $I(F)$, we generally may end up with an interval $I \notin I(F)$. To retain the inclusion property, such interval I is substituted by a wider interval J, $I \subsetneq J$, $J \in I(F)$.

This can be achieved by using directed rounding (Subsection 1.1.1). When computing a result of an interval operation or function, the current rounding mode is first set towards $-\infty$ and the lower endpoint of the result is computed. Then, the rounding mode is switched towards $+\infty$, and the upper endpoint of the result is computed. For example, the result $[x, y]$ of the sum $[a, b] + [c, d]$ would be computed in the following steps.

1. Set rounding mode towards $-\infty$.
2. Compute $x = a + c$.
3. Set rounding mode towards $+\infty$.
4. Compute $y = b + d$.

Written symbolically using the notation from Subsection 1.1.1,

$$[x, y] = [\delta(a + b), \Delta(c + d)]\,. \tag{1.17}$$

Such computation requires two switches of the rounding mode (or even three, if we wish to restore the original rounding mode, after the result is obtained). Because switching rounding mode is costly, a good implementation usually allows to significantly reduce the required number of switches in two ways.

First, it makes use of the fact, that $\delta(-x) = -\Delta(x)$, for any $x \in \mathbb{R}$, see Figure 1.2. Computations of lower endpoints can then be carried out using the rounding towards $+\infty$, instead of towards $-\infty$, eliminating the rounding mode switch between the computation of lower and upper endpoint of the result. For example, (1.17) becomes

$$[x, y] = [-\Delta(-(a + b)), \Delta(c + d)]\,.$$

Notice, how the rounding mode switch is replaced by two negations. It is also possible to store an interval $[a, b]$ directly as a pair of numbers $-a, b$ instead of a, b, to reduce the number of negations.

Secondly, it provides a possibility to set the directed rounding mode at the beginning of a series of consecutive interval computations, and restore its original value only once, at the end of the last computation. This is called the *trust rounding mode* (see also [9, p. 10]), because the user is trusted to change the rounding mode appropriately by herself, should she need to mix interval computation with some other operations, that require different rounding mode.

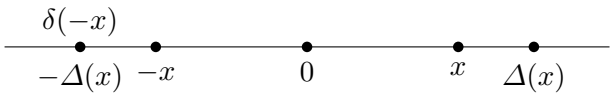


Figure 1.2: Relationship between the directed rounding modes. For $x \in \mathbb{R}$, the number $\Delta(x)$ is by definition the smallest floating-point number greater or equal x. Therefore, $-\Delta(x)$ is the greatest floating-point number smaller or equal $-x$, that is $-\Delta(x) = \delta(-x)$. Similarly, $-\delta(x) = \Delta(-x)$.

1.4 Topological properties of boxes

In this section, we explain some terms used to describe additional properties of boxes, introduced in Section 1.2. They are mostly specialization of general topological terms, but because it is not our goal to dwell deeper into topology, we just stick to boxes, where those terms have reasonable intuitive representation.

1.4.1 Dimension, sub-boxes and orientation

Consider an arbitrary box $B = I_1 \times \cdots \times I_n$. The *dimension* of B is the number k of non-degenerate intervals among $I_1, \ldots, I_n$. We also say, that B is a k-dimensional box, or just k-box. Any box B', that has the same dimension as B and $B' \subseteq B$ is called a *sub-box* of B.

An *orientation* is simply a number $o \in \{-1, 1\}$. If $o = 1$, we also say, that o is the *positive orientation*. Otherwise, o is the *negative orientation*. A pair (B, o) is called an *oriented box*, its *dimension* is the dimension of B. An oriented box (B', o') is an *oriented sub-box* of (B, o), if B' is a sub-box of B and $o = o'$.

Example 1.8. Box $B = [-1, 1] \times [2]$ has dimension 1. Box $C = [0, 1] \times [2]$ is a sub-box of B. Box $D = [0] \times [2]$ is not a sub-box of B, because although $D \subseteq B$, the dimension of D is 0. Oriented box $(C, 1)$ is a sub-box of oriented box $(B, 1)$, but not $(B, -1)$. □

1.4.2 Faces and sub-faces

Let $B = I_1 \times \cdots \times I_n$ be some box. Let $1 \leq j \leq n$ and $[a_j, b_j] = I_j$, and suppose I_j is the k-th non-degenerated component in the sequence $I_1, \ldots, I_n$. Boxes

$$F_j^- = \{(x_1, \ldots, x_n) \,|\, x_j = a_j\},\ F_j^+ = \{(x_1, \ldots, x_n) \,|\, x_j = b_j\} \tag{1.18}$$

are *faces* of B. Further, if we consider an orientation o, then oriented boxes

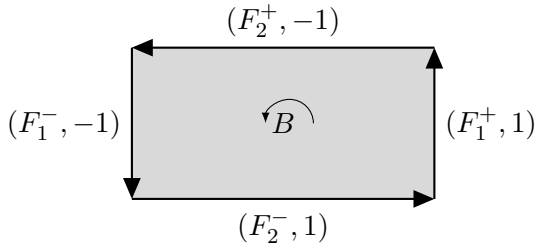
$$(F_j^-, (-1)^k),\ (F_j^+, (-1)^{k+1}) \tag{1.19}$$

are called *oriented sub-faces* of (B, o) and we say, that their orientation is *induced* from the orientation of (B, o). Any (oriented) sub-box of some (oriented) face F is called a sub-face of F. See also Figure 1.3.

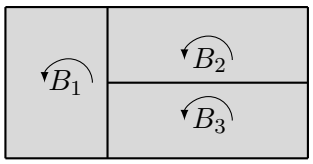
Example 1.9. Box $B = [1, 2] \times [1] \times [3, 4]$ has four faces, namely

$$F_1^- = [1] \times [1] \times [3, 4]\,, \qquad F_1^+ = [2] \times [1] \times [3, 4]\,,$$
$$F_2^- = [1, 2] \times [1] \times [3]\,, \qquad F_1^+ = [1, 2] \times [1] \times [4]\,.$$

If we consider the oriented box $(B, 1)$, then oriented boxes $(F_1^-, -1)$, $(F_1^+, 1)$, $(F_2^-, 1)$ and $(F_1^+, -1)$ are oriented faces of $(B, 1)$. This is sketched in Figure 1.3a. □



(a) Oriented box $(B, 1)$ and its oriented faces, that are depicted as oriented arrows.



(b) Box paving of $(B, 1)$ with three oriented boxes.

Figure 1.3: Boxes, faces and pavings. Here and in further figures, we use oriented arcs to sketch orientations of boxes and oriented arrows to depict their oriented faces and sub-faces. Orientations of such arrows symbolize induced orientations of corresponding faces or sub-faces.

1.4.3 Further notation regarding oriented boxes

In the rest of the text, we assume that any box we work with is oriented. Therefore, we will often refer to any oriented box (B, o) simply as B, and only explicitly write it as the pair (B, o) at points, where the actual value of o is important. We will therefore use phrases like "Let B be an oriented box…", etc, and often even drop the adjective "oriented".

1.4.4 Box paving

Let D be a set of oriented n-boxes, all having the same orientation o. We denote $|D|$ the union of all boxes in D. Set of oriented boxes $\bar{T}$ with the same orientation o is called a *box paving* of D, if

- Intersection of any two boxes from $\bar{T}$ has dimension at most $n - 1$.

- Every box in $\bar{T}$ is a sub-box of some box in D.
- $|\bar{T}| = |D|$.

See Figure 1.3b for a visual example.

1.4.5 Oriented cubical sets

An *oriented cubical set* Ω is a finite set of oriented boxes of the same dimension n and orientation o, such that the intersection of any two different boxes from Ω has dimension at most $n-1$. Values n and o are the *dimension* and *orientation* of Ω, respectively.

The fact that every box in Ω has the same orientation ensures, that whenever two boxes B_1, B_2 have an $(n-1)$-dimensional intersection B_{12}, then the induced orientation o_1 of B_{12} as a sub-face of B_1 is opposite to the induced orientation o_2 of B_{12} as a sub-face of B_2.

For a given oriented cubical set Ω, we denote $|\Omega|$ the union of all boxes in Ω. Any box B can itself be viewed as an oriented cubical set $\{B\}$ and $|B| = B$. Any further terms defined for oriented cubical sets therefore also apply for boxes.

1.4.6 Boundary of an oriented cubical set

The *topological boundary* of an oriented cubical set Ω is purely intuitively the largest set, whose points can be reached both from the inside and from the outside of a box (formalized for example in [10, Chap. 9]). We denote this set by $\partial|\Omega|$

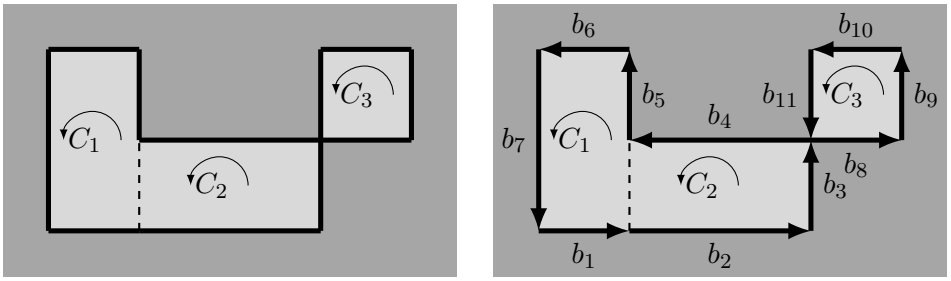
As in [2, pp. 5–6], we also define an *oriented boundary* (or simply *boundary*) of a given oriented cubical set Ω as a set $\partial\Omega$ of oriented boxes of dimension $n-1$, such that

- The intersection of any two boxes from $\partial\Omega$ is at most $n-2$.
- For every $F \in \partial\Omega$ and every $(n-1)$-dimensional sub-box F' of F, there exists exactly one $B \in \Omega$, s.t. F' is an oriented sub-face of B.
- $\partial\Omega$ is maximal, meaning that no other box can be added to it, so that the other two points still hold.

For any boundary $\partial\Omega$, the set $|\partial\Omega|$ always corresponds to the topological boundary $\partial|\Omega|$. See also Figure 1.4.

1.4.7 Bisection of boxes

Consider a box $B = I_1 \times \cdots \times I_n$ and let $I_k = [a, b]$, $a < b$, for some fixed $1 \leq k \leq n$. A *bisection* of B in its component I_k with *bisection ratio* $0 < r < 1$



(a) The topological boundary of Ω. (b) One possible oriented boundary of Ω.

Figure 1.4: Oriented cubical set $\Omega = \{C_1, C_2, C_3\}$. Fig. 1.4a shows its (unique) topological boundary as the thick line segments separating the inside and the outside of $|\Omega|$. Fig. 1.4b shows one possible oriented boundary $\partial\Omega$ of Ω as the set of boxes $b_1, \ldots, b_{11}$, depicted as oriented arrows. Notice, that $|\partial\Omega|$ corresponds to the topological boundary of Ω.

is a process of creating two sub-boxes B_1, B_2 of B,

$$\begin{aligned} B_1 &= I_1 \times \ldots I_{k-1} \times [a, d] \times I_{k+1} \times \cdots \times I_n, \\ B_2 &= I_1 \times \ldots I_{k-1} \times [d, b] \times I_{k+1} \times \cdots \times I_n, \end{aligned} \tag{1.20}$$

where $d = a + r(b - a)$.

1.5 Topological degree

As a part of the algorithm we develop in Chapter 3, we repeatedly need to be able to decide, whether a given continuous $\mathbb{R}^n$-valued function f defined on a n-box X (or, more generally, on an oriented cubical set of n-boxes) has a root in X. To this end, we use the notion and properties of Brouwer's *topological degree* [1, Chap. 1], [2, p. 4].

For a given $\mathbb{R}^n$-valued function f, that is continuous on a given open bounded set $\Omega \subset \mathbb{R}^n$, and a number $d \in \mathbb{R}^n$, the topological degree $\deg(f, \Omega, d)$ is a unique integer of certain properties, whose value depends on the behavior of f on the (topological) boundary of Ω. (The terms *open* and *bounded* set in $\mathbb{R}^n$ are topological properties, see [10, Chap. 5].[4]) and as noted in [2, p. 6], the notion can also be extended to boxes and oriented cubical sets, which we make use of in our work.

This concept is rather formal. However, in the simplest case of $n = 1$ and so called *oriented edges*, which are simply boxes of dimension 1, the topological degree is closely related to Bolzano's intermediate value theorem, which we mention in Subsection 1.5.1.

[4]Intuitively, an open bounded set in $\mathbb{R}^n$ is a generalization of an open interval in $\mathbb{R}$. A set $\mathbb{R}^n$ is open, if it does not contain any of its boundary points, and is bounded, if there exists some $D \in \mathbb{R}$, such that the distance of any two points from the set is less than $D \in \mathbb{R}$.

Further, in the case of $n = 2$ (i.e. in a plane), there is a correspondence between the degree $\deg(f, \Omega, d)$ and the winding number of $f(|\partial\Omega|)$ around d; a notion that has a nice intuitive representation and we describe it in Subsection 1.5.2.

1.5.1 Computing the degree of oriented edges

In the case of oriented edges, the topological degree is closely related to Bolzano's intermediate value theorem [1, p. 8]. Its special case states, that if $f\colon [a, b] \to \mathbb{R}$ is a continuous function and $\operatorname{sgn}(f(a)) \neq \operatorname{sgn}(f(b))$, then there exists $x, a < x < b$, such that $f(x) = 0$.

Now assuming that the interval (box) $[a, b]$ (also denoted by $\overrightarrow{ab}$) has positive orientation and $m = 0$, the degree is defined as

$$\deg(f, [a, b], 0) = \frac{1}{2}(\operatorname{sgn}(b) - \operatorname{sgn}(a)). \tag{1.21}$$

Clearly, the degree is non-zero if and only if $\operatorname{sgn}(f(a)) \neq \operatorname{sgn}(f(b))$, which means that a non-zero value of the degree implies the existence of $x, a < x < b$, such that $f(x) = 0$.

Notice, that to soundly infer the existence of the root, only the information about the behavior of f on the boundary of $[a, b]$ was needed.

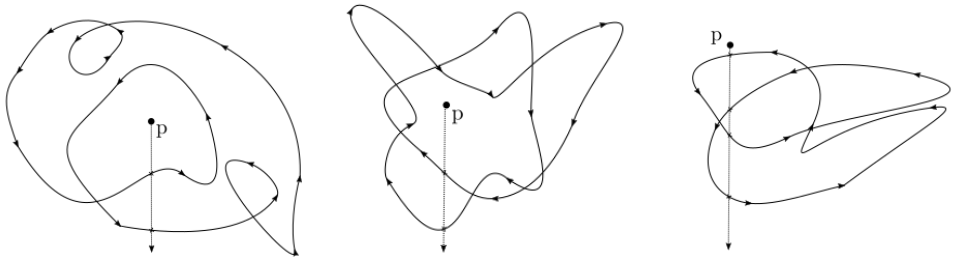
1.5.2 Informal description of the winding number

For a closed oriented curve C in $\mathbb{R}^2$ and a point $d \in \mathbb{R}^2$, that does not lie in C, the *winding number* of C around d is a unique number denoted by $\nu(C, p)$, characterizing the pair (C, d) and representing the number of times the curve C travels around d. The direction of the travel is taken into account. Counterclockwise rotations are considered positive, whereas clockwise rotations are considered negative.

We can visually determine $\nu(C, p)$ by drawing a semi-straight line starting in the origin p and following the direction of the negative y-axis. Each time this semi-straight line is crossed by the curve, we count 1, if the curve crosses from left to right, or -1, if the curve crosses it from right to left. The total count determines $\nu(C, p)$. See Figure 1.5.

The notion of winding number generalizes the idea of determining whether a given point lies inside a given curve, or not. When we have a closed curve that does not cross itself, we can by intuition naturally tell for each point in the plane, whether it lies inside the curve or not. This is because such curve C dividides its complement in a plane into two distinct areas, see Figure 1.6a. Such curves are called *Jordan curves* and their discussed property is the subject of *Jordan curve theorem*. It is a nice example of a theorem that is intuitively clear but it is not trivial to prove formally, see [11].

However, if the curve does cross itself, the idea of points lying inside or outside of the curve starts being unclear. This is where the notion of winding



(a) $\nu(C, p) = 2$. The curve crosses the direction of the negative y-axis from p two times from left to right.

(b) $\nu(C, p) = -2$. The curve crosses the direction of the negative y-axis from p two times from right to left.

(c) $\nu(C, p) = 0$. The curve crosses the direction of the negative y-axis from p two times from left to right and two times from right to left.

Figure 1.5: Visualisation of winding numbers.

number comes in. In general, closed curves can split the plane into multiple areas and the key property here is, that points lying in the same area have the same winding number, see figure 1.6b.

(a) Any closed curve C that does not cross itself (i.e. Jordan curve) splits its complement $\mathbb{R}^2 - |C|$ into two distinct areas.

(b) Non-Jordan curves generally splits their complement $\mathbb{R}^2 - |C|$ into multiple distinct areas. However, any two points in the same area have the same winding number.

Figure 1.6: Closed curves splitting the plane into distinct areas.

1.5.3 Definition and properties of topological degree

For a given $\mathbb{R}^n$-valued function f, that is continuous on a given open bounded set $\Omega \subseteq \mathbb{R}^n$, and a number $d \in \mathbb{R}^n$, the topological degree $\deg(f, \Omega, d)$ can be defined axiomatically, by five axioms listed in [1, pp. 7–8]. For our purposes, the most important of these axioms is so called *solvability*, stating that

$$\deg(f, \Omega, d) \neq 0 \implies \exists x \in \Omega.\, f(x) = d. \tag{1.22}$$

For $d = 0$, this is one of the core ingredient of our algorithm in Chapter 3.

Another noticeable axiom is the one stating, that $\deg(f, \Omega, d)$ is constant on any connected component in $\mathbb{R}^n$, that does not intersect the topological boundary of Ω. We intuitively demonstrated this for winding number in Figure 1.6b.

In our problem domain, we work with boxes and oriented cubical sets. As already mentioned, the notion of topological degree can be extended to them, as noted in [2, p. 6].

In the case when a given function f is differentiable in Ω, and a given point d is regular (meaning, that $\det f'(y) \neq 0$ for every $y \in f^{-1}(d)$), the degree $\deg(f, \Omega, d)$ can be computed as

$$\deg(f, \Omega, d) = \sum_{y \in f^{-1}(d)} \operatorname{sgn} \det f'(y), \tag{1.23}$$

see [2, p. 4] or [1, p. 4].

Although the formula (1.23) is straightforward, it is kinda restrictive, due to its assumptions and the numerical nature stemming from the need of computing derivatives. In Chapter 2, we describe an algorithm for computing topological degree, presented in [2], that specially computes values $\deg(f, X, 0)$, where X is a box or an oriented cubical set and f is a continuous function given in the form of arithmetical expressions. This algorithm requires neither the assumptions of formula (1.23), nor does it rely on computing derivatives or inverses.

1.5.4 Examples

Here, we provide three simple by-hand examples of computing the topological degree via (1.23), whose results are discussed with respect to the solvability property (1.22).

Example 1.10. Consider function $f(x) = 2x$ on $X = [-1, 1]$. For every $q \in X$, we have $f'(q) = 2$, and so the point $d = 0$ is regular. Further, $f^{-1}(0) = \{0\}$ and using (1.23), we get

$$\deg(f, X, 0) = \operatorname{sgn} \det f'(0) = \operatorname{sgn} f'(0) = 2 = 1.$$

Because $\deg(f, X, 0) \neq 0$, the solvability property (1.22) can then be used to deduce, that f has a root in X. □

Example 1.11. Consider function $f(x) = x^2 - 1$ on $X = [-2, 2]$. For $d = 0$, we have $f^{-1}(0) = \{-1, 1\}$ and $f'(-1) = -2$, $f'(1) = 2$. Therefore, point 0 is regular and (1.23) can be used to compute $\deg(f, X, 0)$. We get

$$\deg(f, X, 0) = \operatorname{sgn} \det f'(-1) + \operatorname{sgn} \det f'(1) = -1 + 1 = 0.$$

This shows, that the converse of (1.22) does not hold, because numbers $\pm\sqrt{2}$ are roots of f in X. □

Example 1.12. Consider function $f(x,y) = (x^2 + y^2 - 1, \frac{3}{2}x - y)$ defined on $X = [-1,1]^2$. It has two roots in X, namely $(\frac{2\sqrt{13}}{13}, \frac{3\sqrt{13}}{13})$ and $(-\frac{2\sqrt{13}}{13}, -\frac{3\sqrt{13}}{13})$.

However, as in the previous example, $\deg(f, X, 0) = 0$, because

$$\begin{aligned}\deg(f, X, 0) &= \sum_{(x,y)\in f^{-1}(0)} \operatorname{sgn}\det\begin{pmatrix}\frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y}\\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y}\end{pmatrix} = \sum_{(x,y)\in f^{-1}(0)} \operatorname{sgn}\det\begin{pmatrix}2x & 2y\\ \frac{3}{2} & -1\end{pmatrix}\\ &= \sum_{(x,y)\in f^{-1}(0)} \operatorname{sgn}(-2x - 3y) = -1 + 1 = 0.\end{aligned}$$

And so we se again, that the converse of (1.22) does not hold. □

1.5.5 Parametrized functions

We also work with functions that are parametrized by some parameter box $P \subseteq \mathbb{R}^m$. From Subsection 1.3.4, we now that such functions f_P can be viewed a set of functions $\{f_p \mid p \in P\}$, where f_p is a function obtained from f, by replacing its parameters with a concrete value p.

Now, if a parametrized function f_P is given, and for every two $p, q \in P$, both degrees $\deg(f_p, X, 0)$, $\deg(f_q, X, 0)$ exists and are equal, then we denote this one value as $\deg(f_P, X, 0)$.

Chapter 2

Computing the topological degree

Here we describe the algorithm for computing the topological degree presented in [2], which we studied and decided to make use of in our main algorithm in Chapter 3. We provide a description of the algorithm's workflow, sketching out some of the theoretical results its actions are based on, and referring the reader to particular parts of [2] for all the formal details and proofs. Further, we provide a few example computations done by hand to demonstrate the workflow in a clearer way.

An implementation of the algorithm is available in [12]. It is based on floating point arithmetic. For reasons summarized in Section 2.5, we provide our custom implementation (also based on floating point arithmetic), treating [12] as a reference one, especially when testing, if our own implementation gives correct results, see Section 6.3. To better suit our needs of the design of our solving algorithm (Chapter 3), we make a few adjustments to the workflow of [2] and its implementation [12]. This is explicitly stated further in the text and it is nothing that would violate the correctness of [2].

We also present our custom sound method of determining, if for arbitrary boxes $X' \subseteq X$, $P' \subseteq P$, the degree $\deg(f_p, X', 0)$ is defined and has the same value for every $p \in P'$. This is further used as a part of our main algorithm.

2.1 Overview

Let us start with a general overview of the algorithm [2] and then focus on the details in further sections.

For a given oriented n-box B, or more generally, an oriented cubical set of n-boxes and a continuous $\mathbb{R}^n$-valued function f defined on B, that is nowhere zero on the topological boundary $\partial|B|$ of B, the discussed algorithm computes the degree $\deg(f, B, 0)$ in two phases.

In the first phase, the algorithm searches for an oriented boundary ∂B of B, such that at least one component of the image of each element of ∂B under f does not contain zero. This part is numerical, it requires evaluating subsets of $\partial|B|$ by f. A practical implementation substitutes f with a convergent inclusion monotonic interval extension function F and performs these evaluations using interval arithmetic, which simplifies them and ensures the correctness of the result despite the underlying floating-point computations performed. Section 2.2 is devoted to this phase.

The second phase is combinatorial and requires no further numerical evaluations. It computes the degree recursively from the result provided by the numerical phase. This is the core, the main part of the algorithm and it is described in Section 2.3.

2.2 The numerical phase

The goal of the first phase is to find an oriented boundary $\partial B = \{b_1, \ldots, b_k\}$ of B, such that for each b_j, $j \in \{1, \ldots, k\}$, at least one component f_i of f has a constant sign on b_j—that is, f_i is either strictly positive, or strictly negative in b_j. Such oriented boundary, along with the sign information assigned to each box b_j, then forms the result of this phase. It is called a *sign covering* and it is introduced in [2, pp. 7–8, 11].

We tend to use the notion of sign covering quite often both in the text, as well as in our implementation, and so we rephrase the definition in the following subsection for completeness.

2.2.1 Definition of a sign covering

A d-dimensional *sign vector* is a vector consisting of elements from $\{+, -, 0\}$. We denote the l-th component of a sign vector v as v_l.

If $S = \{B_1, \ldots, B_p\}$ is a set of oriented $(d-1)$-boxes, then a *sign covering* of S is defined as a list of pairs $[(B_1, v_1), \ldots, (B_p, v_p)]$, where v_i is a d-dimensional sign vector. The *dimension* of a sign covering of S is the dimension of any box in S.

A sign covering of S is said to be *sufficient*, if each sign vector v_i contains at least one non-zero element, and is said to be *with respect to* (written shortly as wrt.) a function

$$f\colon \left(\bigcup_{i \in \{1,\ldots,p\}} B_i \right) \subseteq D \to \mathbb{R}^d,$$

if for every $i \in \{1, \ldots, p\}$ and $j \in \{1, \ldots, d\}$, such that $v_j \neq 0$, the component f_j of f has a constant sign v_j on B_i. We follow by a simple example.

Example 2.1. Let $B_1 = [-2, -1]$, $B_2 = [-1, 1]$ and $B_3 = [1, 2]$. Then

$$[(B_1, \left(\begin{smallmatrix} - \\ + \end{smallmatrix}\right)), (B_2, \left(\begin{smallmatrix} 0 \\ + \end{smallmatrix}\right)), (B_3, \left(\begin{smallmatrix} + \\ + \end{smallmatrix}\right))]$$

is a sufficient sign covering of $\{B_1, B_2, B_3\}$ wrt. function $f(x) = (x, \exp(x))$. Its dimension is 1. □

Now we can say, that the goal of the numerical phase is to take the input n-box or oriented cubical set B and the function f defined on B, that is nowhere zero on $\partial|B|$, and compute a sufficient sign covering of some oriented boundary of B wrt. f. This is then passed to the combinatorial phase of the algorithm.

2.2.2 Sign covering computations

There is a slight difference of how our implementation and the reference one [12] performs sign covering computations during the numerical phase of the algorithm. The basic flow is similar for both and we focus at it first. The difference is emphasized in the next subsection.

Supposing an input function f and a box or an oriented cubical set B, we always start with some arbitrary initial oriented boundary ∂B of B. If B is a single box, then ∂B is typically the set of all oriented faces of B. Every box b in ∂B is then evaluated by an interval inclusion function F of f and gets associated a sign vector v, such that its i-th component v_i is determined by the value of $F_i(b)$ as follows (F_i is the i-th component of F).

$$v_i = \begin{cases} - & \text{if } \overline{F_i(b)} < 0, \\ + & \text{if } \underline{F_i(b)} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

This creates an initial sign covering, which is wrt. F and therefore also wrt. f. It may, however, not be sufficient.

Any box b, that has a zero vector 0 associated (that is any box, whose image under F contains zero) is then bisected into a pair of smaller boxes b_1, b_2. In our implementation, the bisection is always done along the longest component of b[5], with a ratio given in advance as a parameter. To our knowledge, the reference implementation [12] also performs the bisection along the longest component, but has the ratio hardcoded to 0.5.

Box b is then replaced by b_1, b_2. This creates a new oriented boundary of B, which we often refer to as a *refined* oriented boundary. Boxes b_1, b_2 are then reevaluated by F and new pairs of boxes and sign vectors, say (b_1, v_1) and (b_2, v_2) are created, replacing the original pair $(b, 0)$. This forms a sign covering of the refined oriented boundary. This process then continues, until a sufficient sign covering is obtained.

[5] If $b = I_1 \times \cdots \times I_n$ has two or more components of the maximal width, then b is bisected along the one with lower index.

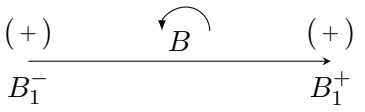
As long as the input contract of f not being zero anywhere on $\partial|B|$ is satisfied, the existence of a sign covering of the desired properties is guaranteed, so the process terminates in a finite number of steps.[6]

We now apply these steps to compute a few examples by hand and present them. For simplicity, we do not consider any interval inclusion function F and simply perform all evaluations by f itself.

Example 2.2. Consider the function $f(x) = \exp(\mathrm{abs}(x))$ on $B = [-2, 2]$, where the orientation of B is 1. As Figure 2.1 shows, we consider the set oriented faces $S = \{B_1^-, B_1^+\}$, which is an oriented boundary of B, and compute a sign covering of S wrt. f using the rule (2.1). We get

$$L = [(B_1^-, +), (B_1^+, +)],$$

which is already sufficient and is therefore the desired result. □



box	or.
$B_1^- = [-2]$	-1
$B_1^+ = [2]$	1

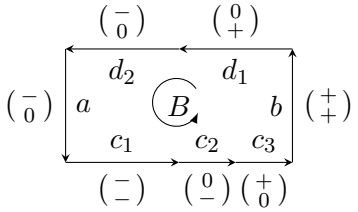
Figure 2.1: Example of a sufficient sign covering of an oriented boundary $\{B_1^-, B_1^+\}$ of $B = ([-2, 2], 1)$ with respect to $f(x) = \exp(\mathrm{abs}(x))$. Boxes B_1^- and B_1^+ are the oriented faces of B.

Example 2.3. Let $f(x) = (2x^2 - y^2 - 2, x^2 + y^2 - 4)$ be defined on the box $B = [0, 2] \times [1, 2]$ with orientation 1. Let B_1^-, B_1^+, B_2^-, B_2^+ denote the four oriented faces of B. No sufficient sign covering wrt. f of the oriented boundary $\{B_1^-, B_1^+, B_2^-, B_2^+\}$ of B exists, because no component of f has a constant sign on B_2^- and B_2^+ as the following shows.

$$f(B_2^-) = f([0, 2] \times [1]) = ([-3, 5], [-3, 1]),$$
$$f(B_2^+) = f([0, 2] \times [2]) = ([-6, 2], [0, 4]).$$

So in this case, boxes B_2^- and B_2^+ get zero vectors associated by (2.1) and the initial boundary needs to be refined by bisecting these into sub-boxes. Figure 2.2 shows one possible refined oriented boundary. The corresponding sign covering is sufficient and contains 7 boxes. □

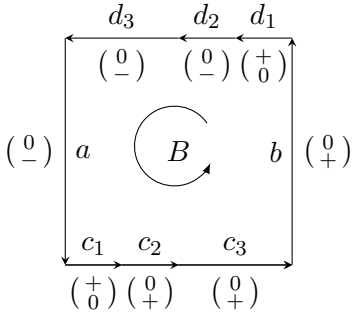
[6]There is of course a chance, that although a sign covering of the desired properties exists, the implementation may not find because of the limited precision of floating-point numbers. For our purposes, we do not have to consider this case, however, because of the way we adjust the input contract of the computation, see Subsection 2.2.4.



box	or.
$a = [0] \times [1, 2]$	-1
$b = [2] \times [1, 2]$	1
$c_1 = [0, 1] \times [1]$	1
$c_2 = \left[1, \frac{3}{2}\right] \times [1]$	1
$c_3 = \left[\frac{3}{2}, 2\right] \times [1]$	1
$d_1 = [1, 2] \times [2]$	-1
$d_2 = [0, 1] \times [2]$	-1

Figure 2.2: Example of a sufficient sign covering of an oriented boundary of $B = ([0, 2] \times [1, 2], 1)$ wrt. $f(x, y) = (2x^2 - y^2 - 2, x^2 + y^2 - 4)$ from Example 2.3. The depicted sign covering is made of 7 boxes: $a, b, c_1, c_2, c_3, d_1, d_2$. Boxes c_1, c_2, c_3 are sub-faces of B_2^-, d_1, d_2 are sub-faces of B_2^+ and $a = B_1^-$, $b = B_1^+$.

Example 2.4. Consider the function $f(x) = (x^2 + y^2 - 1, \frac{3}{2}x - y)$ on the box $B = [-1, 1] \times [-1, 1]$ with orientation 1. As in Example 2.3, no sufficient sign covering of the oriented boundary $\{B_1^-, B_1^+, B_2^-, B_2^+\}$ of B exists, because no component of f has a constant sign on B_2^- and B_2^+. So again, some refined boundary of B needs to be considered, like the one in Figure 2.3. The corresponding sign covering contains 8 boxes. □



box	or.
$a = [-1] \times [-1, 1]$	-1
$b = [1] \times [-1, 1]$	1
$c_1 = \left[-1, -\frac{1}{2}\right] \times [-1]$	1
$c_2 = \left[-\frac{1}{2}, 0\right] \times [-1]$	1
$c_3 = [0, 1] \times [-1]$	1
$d_1 = \left[\frac{1}{2}, 1\right] \times [1]$	-1
$d_2 = \left[0, \frac{1}{2}\right] \times [1]$	-1
$d_3 = [-1, 0] \times [1]$	-1

Figure 2.3: Example of a sufficient sign covering of an oriented boundary of $B = ([-1, 1] \times [-1, 1], 1)$ wrt. $f(x, y) = (x^2 + y^2 - 1, \frac{3}{2}x - y)$ from Example 2.4. The depicted sign covering is made of 8 boxes: $a, b, c_1, c_2, c_3, d_1, d_2, d_3$. Boxes c_1, c_2, c_3 are sub-faces of B_2^-, d_1, d_2, d_3 are sub-faces of B_2^+, $a = B_1^-$, $b = B_1^+$.

2.2.3 Adjusting the input contract

When the reference implementation [12] computes a sign covering of an oriented boundary of B wrt. f, it relies on the input contract, that $0 \notin f(\partial|B|)$. If this condition is not satisfied by the input, then no sufficient sign covering of

the desired properties exists, so the computation gets stuck in an infinite loop, refining oriented boundaries of B, in the way described in Subsection 2.2.2, over and over again.

This behavior is not suitable for our needs. For the purposes of our solving algorithm discussed in Chapter 3, we rather want to simply drop the condition $0 \notin f(\partial|B|)$ and replace it with some parameter, that tells "how hard" to try when looking for a sufficient sign covering before giving up and terminating.

We solve this by adding a stop condition to the process from Section 2.2.2. It has the form of a number $ref \geq 0$, called a *refinement threshold*, such that no box of the width smaller or equal to *ref* is bisected further. When no box in a currently computed sign covering can be bisected further, we terminate the computation and return the currently computed sign covering. This is guaranteed to be wrt. f, but it might not be sufficient.

Now what does this mean for the whole process of computing $\deg(f, B, 0)$? The algorithm, as specified in [2], requires the resulting sign covering from the numerical phase to be sufficient so it can be passed to the combinatorial phase. Because in our case we cannot always guarantee that the sign covering will be sufficient, we need adjust the algorithm's workflow by sort of separating the two algorithm phases as depicted in Figure 2.4

In words, when performing the sign covering computation during the numerical phase of computing the degree, we first need to check, if the obtained sign covering is sufficient. If it is, then we can send it further to the combinatorial phase, otherwise, we must return some reserved value indicating a failure.

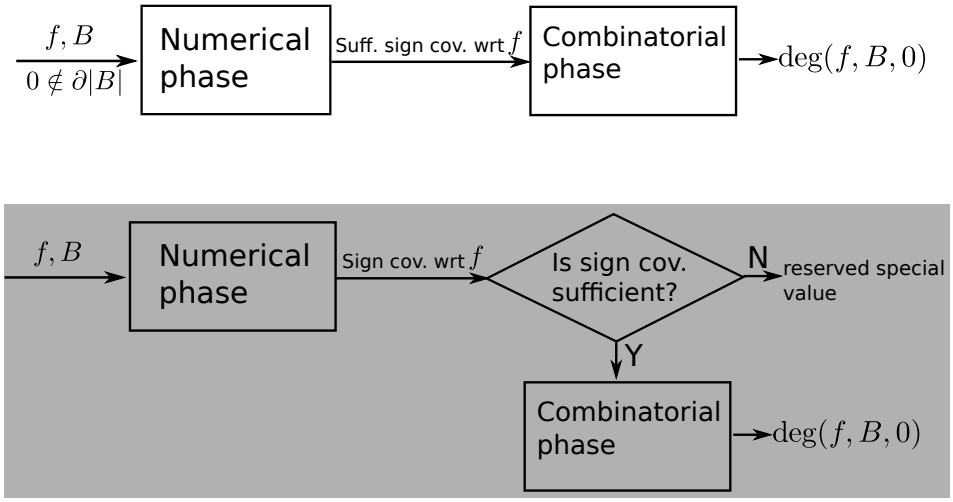


Figure 2.4: Adjusted workflow of the algorithm for computing the topological degree. The basic difference between the reference implementation [12] (white background) of the algorithm [2] and our own implementation (gray background) is shown. See Subsection 2.2.3.

2.2.4 Pseudocode for sign covering computation

Here we formulate the steps for computing a sign covering discussed in Subsections 2.2.2 and 2.2.3 into a pseudocode, so that we can reference it from our solving algorithm in Chapter 3.

We introduce a function SIGNCOVERING, that takes a given set *boxes* of boxes and computes a sign covering of *boxes* wrt. a given interval inclusion function F of a given input function f. It does so by using the helper function SIGNVECTOR, that computes one pair of a box and sign vector in the constructed sign covering according to the rule (2.1). It then keeps refining the set (that is, it repeatedly bisects its boxes into sub-boxes) and updating the sign covering accordingly, until it is either sufficient or the width of each box becomes smaller or equal to a given refinement threshold *ref*. The pseudocode is in Algorithm 1.

Notice, that we formulated it more generally than just for the computation of a sufficient sign covering of some oriented boundary ∂B of a box or an oriented cubical set B wrt. f. But it clearly can be used for this purpose, because its output contract ensures, that as long as *boxes* is an oriented boundary of B, then the returned sign covering *sc* is a sign covering of some oriented boundary of B as well (possibly other, refined one). This follows from the properties of box bisection (Subsection 1.4.7).

The resulting sign covering *sc* may not be sufficient and from Subsection 2.2.3 (also see Figure 2.4) we know, that we need a way to check it. This is done by the function ISSUFFICIENT. Its pseudocode is trivial and we include it in Algorithm 2 mainly for completeness.

Algorithm 1 Construction of a sign covering of a set of boxes with respect to a given function, using its interval inclusion function.

Input: Set *boxes* of boxes of dimension $n-1$; Implementation F of incl. func. $f\colon D \subseteq \mathbb{R}^n \to \mathbb{R}^n$; Refinement threshold $ref \geq 0$.; Bisection ratio r with default value 0.5.

Output: Sign covering *sc* of a set *out* wrt. f, such that every $B' \in out$ is a sub-box of some box $B \in boxes$, $\omega(B') \geq ref$ and $|boxes| = |out|$.

```
1:  function SIGNCOVERING(boxes, F, ref, r = 0.5)
2:      sc ← empty sign covering, st ← empty stack of boxes
3:      push every box in boxes onto st
4:      while st is not empty do
5:          B ← pop the top box from st
6:          sv ← SIGNVECTOR(B, F)
7:          if sv contains a non-zero element or ω(B) ≤ ref then
8:              append (B, sv) to sc          ▷ sv might be possibly a zero vector
9:          else                              ▷ sv is a zero vector and ω(B) > ref
10:             (B1, B2) ← bisect B in its longest component with ratio r
11:             push B1, B2 onto st
```

```
12:         end if
13:     end while
14:     return sc
15: end function
16: function signVector(B, F)
17:     range ← F(B)
18:     for i ∈ {1, ..., n} do
19:         r_i ← i-th component of range
20:         v_i ← if 0 ∈ r_i then 0 else sign of r_i
21:     end for
22:     return (v_1, ..., v_n)
23: end function
```

Algorithm 2 Determining sufficiency of a sign covering.

Input: Sign covering *sc* of some set *boxes* of boxes.
Output: True, if *sc* is sufficient. Otherwise false.

```
1: function signCovering(sc)
2:     for each (B, sv) ∈ sc do
3:         if sv is zero vector then return false
4:     end for
5:     return true
6: end function
```

The correctness of Algorithm 1 with respect to the output requirement $|boxes| = |out|$ is based on the way box bisection is defined. Namely, if a box X is bisected into X_1, X_2 (regardless of the ratio or along which component), then $X = X_1 \cup X_2$.

2.3 The combinatorial phase

The second phase of computing the topological degree, called the combinatorial phase, requires no additional numerical evaluations. It accepts the sign covering *sc* computed in the numerical phase, which is a sign covering of an oriented boundary of the input n-box or oriented cubical set B of n-boxes wrt. the given $\mathbb{R}^n$-valued function f defined on B.

It then recursively extracts necessary information from *sc* and constructs a new sign covering sc' wrt. a particular function g constructed from f by omitting one of its components, until it is left with a sign covering of 0-boxes and with a function of only 1 component.

We denote the result of one level of this recursion step, that rebuilds some sign covering L into L', as $\textsc{deg}(L)$. If L happens to be *sc*, then

$$\textsc{deg}(L) = \deg(f, B, 0),$$

which is the contract of the top level of this recursion.

This process has a rich theoretical background. We therefore think, that it is best to first present the step-by-step workflow of DEG(L) in Algorithm 3 and then follow by some example computations before sketching the theoretical background. There is a note we would like to mention.

The original [2, pp. 13–14] does not mention, how exactly boxes in Step 3b of Algorithm 3 should be split. It does mention the goal of such bisections, but leaves a systematic way of performing them up to concrete implementations. We use a similar approach as the reference implementation [12], described separately in Subsection 2.3.1 in order not to clutter the flow of Algorithm 3 too much.

Algorithm 3 Combinatorial phase of algorithm [2] (rephrased to better match our notation.)

Input: Sign covering L of dim. m wrt. $\mathbb{R}^{m+1}$-valued function f defined on all boxes in L.

Output: Integer d. If L is a sufficient sign covering of some oriented boundary ∂B of B wrt. f, then $d = \deg(f, B, 0)$.

function DEG(L)

1. If L is empty, set $d = 0$ and return. If it is a sign covering of 0-boxes, then set
$$d = \frac{1}{2} \sum_{(b,v) \in L} \text{orientation}(b) \cdot v \tag{2.2}$$
and return. Otherwise, continue with the next step.
2. Choose a sign $s \in \{+, -\}$ together with an index $1 \leq l \leq m + 1$. The sign covering L is partitioned into two sets, L_{sel}, that contains all the pairs (a, v) from L, such that $v_l = s$, and $L_{non} = L - L_{sel}$.
3. Create an empty sign covering L'. For each box a in L_{sel}, perform the following.
   a) Create the set *bounds* of all oriented faces of a.
   b) Iterate over the boxes in *bounds*. Each time you encounter a box b, that has an $(m-1)$-dimensional intersection with some box from L_{non}, yet at the same time is not its subset, bisect this box into sub-boxes, replace it with the sub-boxes and reevaluate them.

      The goal is to perform the bisections in such way, that eventually, every box b' in *bounds* is either a subset of some box from L_{non} or its intersection with every box from L_{non} has dimension less than $(m-1)$. See Subsection 2.3.1 for more details.
   c) Take every box b in *bounds*, which is a subset of some element c in L_{non} and append the pair (b, v) to L', where v is a sign vector

associated to c (in the sign covering L) with omitted l-th component.

4. Set $d = s \cdot (-1)^{l+1} \cdot \text{DEG}(L')$ and return.

end function

2.3.1 Bisecting boxes in the set *bounds*

Let us return to Step 3 of Algorithm 3. The original [2, pp. 13–14] does not state an exact systematic way of how to bisect boxes in the set *bounds* and leaves this decision to concrete implementations.

We decided to use an approach similar to the one used by the reference implementation [12] and here we describe it.

In the following, we use the notation from Algorithm 3. Let $b \in$ *bounds* and $c \in L_{non}$. Both b and c are m-boxes. The dimension of the box b is $m-1$ and the dimension of the box c is m. Suppose, that b is not a subset of c, yet they have an $(m-1)$-dimensional intersection.

The bisection is performed as follows (you may also refer to Figure 2.5 for a visual aid). If $b = I_1 \times \cdots \times I_m$ and $c = J_1 \times \cdots \times J_m$, choose the lowest index $k \in \{1, \ldots, m\}$, such that $I_k \not\subseteq J_k$. Because of the assumptions, that the dimension of b is $m-1$, the dimension of c is m and b has a $(m-1)$-dimensional intersection with c, it holds that

- either $\underline{I_k} < \underline{J_k}$ (and then also $\underline{J_k} < \overline{I_k} \leq \overline{J_k}$; this situation is depicted in Figure 2.5),
- or $\overline{I_k} > \overline{J_k}$ (and then $\overline{J_k} > \underline{I_k} \geq \underline{J_k}$).

In the former case, b is bisected into

$$\begin{aligned} b_1 &= I_1 \times \cdots \times I_{k-1} \times \left[\underline{I_k}, \underline{J_k}\right] \times I_{k+1} \times \cdots \times I_n, \\ b_2 &= I_1 \times \cdots \times I_{k-1} \times \left[\underline{J_k}, \overline{I_k}\right] \times I_{k+1} \times \cdots \times I_n, \end{aligned} \tag{2.3}$$

and in the latter case, b is bisected into

$$\begin{aligned} b_1 &= I_1 \times \cdots \times I_{k-1} \times \left[\underline{I_k}, \overline{J_k}\right] \times I_{k+1} \times \cdots \times I_n, \\ b_2 &= I_1 \times \cdots \times I_{k-1} \times \left[\overline{J_k}, \overline{I_k}\right] \times I_{k+1} \times \cdots \times I_n. \end{aligned} \tag{2.4}$$

Notice, that in the former case, box b_1 no longer has a $(m-1)$-dimensional intersection with c, because the intersection of its k-th component $\left[\underline{I_k}, \underline{J_k}\right]$ with J_k is the degenerated interval $\left[\underline{J_k}\right]$. In addition, the k-th component $\left[\underline{J_k}, \overline{I_k}\right]$ of b_2 is a subset of J_k, because $\overline{I_k} \leq \overline{J_k}$. Similarly for the latter case of bisection.

Bisection results b_1, b_2 then replace b in the set *bounds* and are then processed in the same way as b. This whole process ensures, that the original

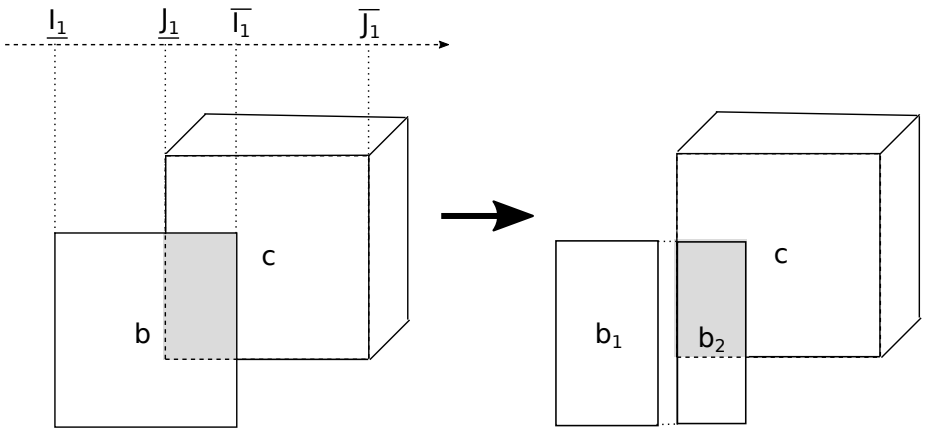


Figure 2.5: Bisection of a box b in the *bounds* set. Follow the text of Subsection 2.3.1. Here, both $b \in$ *bounds*, $c \in L_{non}$ are 3-boxes, the dimension of b is 2 and the dimension of c is 3. They have a 2-dimensional intersection (grey color in the fig.), but b is not a subset of c. The first component of b is I_1 and the first component of c is J_1. Boxes b_1, b_2 are the bisection results and b_1 no longer has a 2-dimensional intersection with c.

box b eventually gets replaced by boxes, that can be divided into two categories. Boxes in the first category will be subsets of some boxes from L_{non}, while boxes in the second category will not have an $(m-1)$-dimensional intersection with any box from L_{non}.

2.3.2 Example computations

Here we present a few by hand computed examples, in which we perform the combinatorial phase and compute topological degrees. These examples use the inputs and results from the examples from Subsection 2.2.2.

Example 2.5. The result of Example 2.2 was a sign covering L of an oriented boundary of $B = [-1, 1]$ with orientation 1 wrt. $f(x) = \exp(\text{abs}(x))$, depicted in Figure 2.1.

Boxes in this covering are of dimension 0, therefore no recursive reduction is performed and the combinatorial phase just computes the formula 2.2 and returns its value. So in total,

$$\deg(f, B, 0) = \text{DEG}(L) = 1 \cdot 1 + (-1) \cdot 1 = 0. \qquad \square$$

Example 2.6. Figure 2.2 displays a sign covering L of an oriented boundary of $B = [0, 2] \times [1, 2]$ with orientation 1, wrt. function

$$f(x) = (2x^2 - y^2 - 2, x^2 + y^2 - 4).$$

This sign covering was the result of Example 2.3.

Figure 2.6 then visually depicts, how a new sign covering L' is constructed from L in $\text{DEG}(L)$

For the sign $s = -$ and index $l = 1$, boxes a, c_1, d_2 are selected into L_{sel}. For each of these boxes, the respective set *bounds* of oriented faces is then constructed and examined. We come across six boxes in total, namely A^-, A^+, C^-, C^+, D^-, D^+. Two of them, C^+ and D^- are subsets of non-selected boxes, c_2 and d_1, respectively. From C^+, D^- and the sign vectors of c_2, d_1, a new sign covering L' is constructed. It contains two pairs, $((C^+, 1), (-))$ and $((D^+, -1), (+))$.

L' becomes the input for the next recursive call. Because it is already a sign covering of 0-boxes, this call is the bottom level of the recursion and it returns

$$\text{DEG}(L') = \frac{-1 \cdot 1 + 1 \cdot (-1)}{2} = -1,$$

and altogether

$$\deg(f, B, 0) = \text{DEG}(L) = -(-1)^{1+1} \cdot \text{DEG}(L') = -1 \cdot (-1) = 1. \qquad \square$$

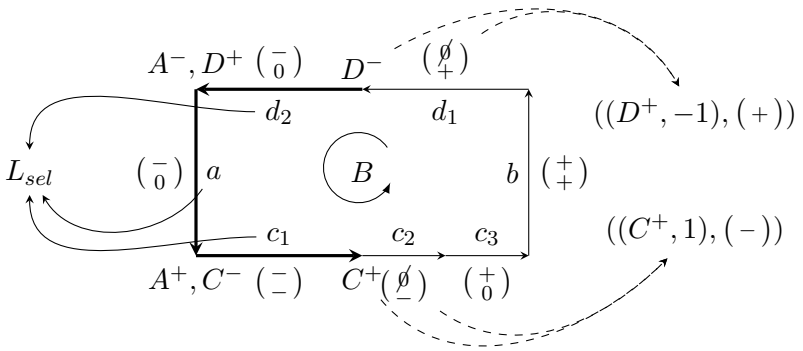


Figure 2.6: One step of combinatorial for the input from Example 2.6. Orientation of B is 1, newly constructed sign covering contains pairs pointed to by dashed arrows. L_{sel} contains the boxes a, d_2, c_1. Their oriented faces are denoted by capital letters.

Example 2.7. Consider the function $f(x) = (x^2 + y^2 - 1, \frac{3}{2}x - y)$ on the box $B = [-1, 1] \times [-1, 1]$ with orientation 1, discussed in Example 2.4. A sufficient sign covering L of an oriented boundary of B wrt. f is in Figure 2.3.

The workflow of a call to $\text{DEG}(L)$, up to the point of constructing a sign covering L' of dimension 0, is in Figure 2.7.

For the sign $s = +$ and index $l = 1$, boxes c_1, d_1 are selected into L_{sel}. For each of these boxes, the respective set *bounds* of oriented faces is created and examined. The total of four boxes are encountered, namely C^-, C^+, D^-, D^+,

all of which are subsets of non-selected boxes, a, c_2, d_2 and b respectively. From these, a new sign covering L' is constructed, containing four pairs in total, $((C^-, -1), (-))$, $((C^+, 1), (+))$, $((D^-, 1), (-))$ and $((D^+, -1), (+))$.

The sign covering L' becomes the input to the bottom level recursive call, that returns

$$\textsc{deg}(L') = \frac{-1 \cdot (-1) + 1 \cdot 1 + 1 \cdot (-1) + (-1) \cdot 1}{2} = 0,$$

and altogether

$$\deg(f, B, 0) = \textsc{deg}(L) = (-1)^{1+1} \cdot \textsc{deg}(L') = 1 \cdot 0 = 0.$$

Notice, that a different selection of $s = -$ and $l = 1$ causes $L_{sel} = \emptyset$, which results in $L' = \emptyset$, and the call to $\textsc{deg}(L')$ simply returns 0, without the need of further computation. In general, the way of selecting the sign and index matters and we discuss it more in Subsection 2.3.4.

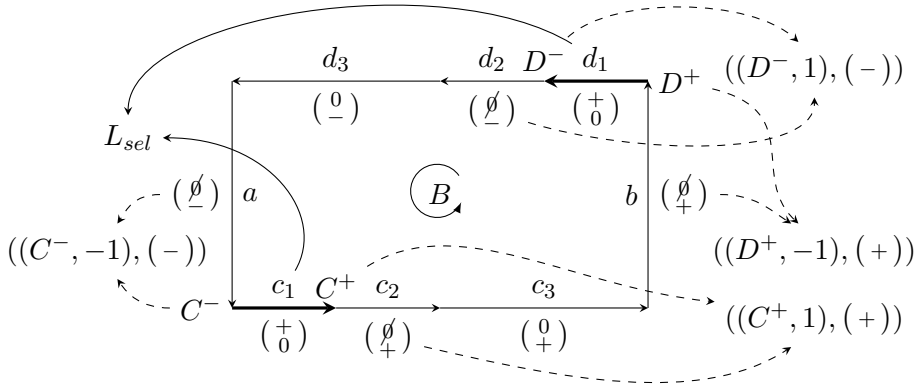


Figure 2.7: One step of combinatorial phase for the input from Example 2.7. Orientation of B is 1 and newly constructed sign covering contains pairs pointed to by dashed arrows

2.3.3 Sketching the theoretical background

The following paragraphs are meant to sketch the general ideas involved in Algorithm 3. Again, for all the in detail explanation and proofs, refer to [2].

Let s, l denote the sign and index, respectively, selected in the second step of Algorithm 3. As explained in depth in [2, Theorem 2.9], if L is a sufficient sign covering of an oriented boundary of B wrt. the input function f, $0 \notin f(|\partial B|)$, then the boxes in L_{sel} can be partitioned into oriented cubical sets $D_1, \ldots, D_p$ and corresponding oriented boundaries $\partial D_1, \ldots, \partial D_p$, such that for every $i \in \{1, \ldots, p\}$, each box in ∂D_i is a subset of some box in L_{non},

and $0 \notin f_{\neg l}(|\partial D_i|)$, where $f_{\neg l}$ is a function formed from f by excluding its l-th component. In addition, the degree can be expressed as

$$\deg(f, B, 0) = s(-1)^{l+1} \sum_{i \in \{1,\ldots,p\}} \deg(f_{\neg l}, D_i, 0). \tag{2.5}$$

Now let $c \in L_{non}$ denote a box, that some fixed ∂D_i is a subset of, and let v denote the associated sign vector of c in the sign covering L. Because the box c was not selected into L_{sel}, we necessarily have $v_l \neq s$. Further, the non-empty intersection of c with ∂D_i implies, that v_l cannot be the opposite sign of s, because L is wrt. f. Therefore $v_l = 0$, However, L is also sufficient, and therefore v must have a non-zero sign at least at one position other than l.

And so a sufficient sign covering of ∂D_i wrt. $f_{\neg l}$ exists and the right hand side of 2.5 can be computed recursively performing the same steps. This is one of the core factors of the algorithm.

However, notice from the described steps of computing $\textsc{deg}(L)$ in Algorithm 3, that it actually does not construct the sign covering L'_i for each ∂D_i separately and does not compute the right hand side of 2.5 by performing p recursive calls at the same level. Instead, it performs the call only once at each level with the sign covering L' (Step 4 in Alg. 2.5). The way L' is constructed guarantees it to contain every box in each ∂D_i. Apart from these, L' may possibly contain some other boxes, namely some pairs of identical boxes with opposite orientation. Nevertheless, as [2, Chapter 3] explains, $\textsc{deg}(L') = \sum_i \textsc{deg}(L'_i)$, and this approach yields the correct result.

By constructing and using L' for the recursive computation of the degree, the actual searching for the decomposition $D_1, \ldots, D_p$ and $\partial D_1, \ldots, \partial D_p$ is elegantly bypassed and the number of recursive calls is reduced to one per level.

Finally, the return value (2.2) for a sign covering with 0-boxes corresponds to the fact, that the boxes in such sign covering are oriented faces of oriented edges, and the topological degree of an oriented edge $\overrightarrow{ab}$ (interval $[a, b]$ with positive orientation) is given as

$$\deg(f, \overrightarrow{ab}, 0) = \frac{1}{2}(\operatorname{sgn}(b) - \operatorname{sgn}(a)). \tag{2.6}$$

2.3.4 Sign-index selection strategy

The way the sign s and index l are selected in Algorithm 3 may impact the running time of the computation. In [2], the authors state that according to their numerical experiments, the computation tends to take more time as the number of boxes selected into L_{sel} increases, and so it makes sense to select s and l in each recursion step, so that the number of selected boxes is minimal.

We call any systematic way of selecting s and l a *sign-index selection strategy*. The advantage of the one mentioned in the previous paragraph,

which we decided to call the *least frequent sign-index selection strategy* is, that when no sign vector in the examined sign covering L matches a certain combination of s and l, then this combination is selected, because it yields $L_{sel} = \emptyset$. This then causes the constructed sign covering L' to also become $\emptyset$, and so the following call DEG(L') immediately returns 0 and the computation ends. We have actually seen this already in Example 2.7.

2.4 Parametrized functions

In Chapter 3, when developing our solving algorithm, we work with functions that are parametrized with some parameter box $P \subseteq \mathbb{R}^m$. By ourselves, we made a few considerations regarding parametrized functions and the topological degree computation.

As already stated in Subsection 1.3.4, a function f_P defined on $X \subseteq \mathbb{R}^n$ and parametrized with $P \subseteq \mathbb{R}^m$ can be viewed as a set of functions $\{f_p \,|\, p \in P\}$, where f_p is a function obtained from f by replacing its parameters with one concrete value $p \in P$. Therefore, we can say, that a given sign covering sc is *with respect to* the function f_P, if it is with respect to every function f_p, $p \in P$.

It may then make sense to ask, whether for a box or an oriented cubical set B, the degree $\deg(f_P, B, 0)$ (Section 1.5) exists and how to compute it. Well, if B together with each f_p, $p \in P$ satisfies the input properties for computing $\deg(f_p, B, 0)$, then we can pass this one sign covering sc to the combinatorial phase. That is, we can make the call DEG(sc), whose contract ensures, that the single computed value is equal to $\deg(f_p, B, 0)$ for any $p \in P$, because sc is a sufficient sign covering wrt. every f_p, $p \in P$. In other words, the $\deg(f_P, X, 0)$ exists and DEG(sc) computes its value.

Notice, that the combinatorial phase does not have to know, that we actually work with parametrized functions. It is solely up to the numerical phase to provide the sign covering with desired properties. And so to this end, we simply enhance the input contract of our function SIGNCOVERING (Subsection 2.2.4), to accept interval inclusion functions F_P of parametrized functions f_P (see Subsection 1.3.4).

Example 2.8. Consider the function $f(x) = (x + p, y)$ on $B = [1, 2]^2$ (with orientation 1), parametrized by $P = [-1, 1]$ (x, y denote the variables, p denotes the parameter).

We are interested in finding $\deg(f_P, B, 0)$, if it exists. We first need to find a sufficient sign covering of some oriented boundary of B wrt. f_P. To this end, we consider an interval inclusion function F_P of f_P in the sense of Subsection 1.3.4 and the set S of all oriented faces of B. We then simply perform the steps of Algorithm 1, passing F_P and S as inputs. This mainly

involves evaluating all boxes in S under F_P, that is

$$\begin{aligned}
F_P(B_1^-) &= F_P([1] \times [1,2]) = (1 + [-1,1], [1,2]) = ([0,2], [1,2]),\\
F_P(B_1^+) &= F_P([2] \times [1,2]) = (2 + [-1,1], [1,2]) = ([1,3], [1,2]),\\
F_P(B_2^-) &= F_P([1,2] \times [1]) = ([1,2] + [-1,1], 1) = ([0,3], 1),\\
F_P(B_2^+) &= F_P([1,2] \times [2]) = ([1,2] + [-1,1], 2) = ([0,3], 2).
\end{aligned}$$

Now, based on these evaluations, boxes in S get associated a sign vector by the rule (2.1) (written in pseudocode as the function signVector in Algorithm 1). This results in a sign covering sc of S wrt. F_P (and therefore also wrt. f_P). The pairs in sc are as follows (boxes are written with their orientations).

box	sign vector	box	sign vector
$(B_1^-, -1)$	$\binom{0}{+}$	$(B_1^+, 1)$	$\binom{+}{+}$
$(B_2^-, 1)$	$\binom{0}{+}$	$(B_2^+, -1)$	$\binom{0}{+}$

We see, that sc is sufficient, so the numerical phase ends and we pass sc to the combinatorial phase. That is, we compute $\text{DEG}(sc)$ (Algorithm 3), whose contract ensures, that the computed value is actually $\deg(f_P, B, 0)$. To compute $\text{DEG}(sc)$, we simply proceed as in the examples from Subsection 2.3.2. We can do this in a smart way and perform the least frequent sign-index selection strategy to select the sign s and index l. This results in setting s to $-$ and l to either 1 or 2, and so L_{sel} and subsequently the new sign covering L' constructed from it all become $\emptyset$. The recursive call $\text{DEG}(L')$ then simply returns 0. So we get

$$\deg(f_P, B, 0) = \text{DEG}(sc) = -(-1)^{l+1} \cdot \text{DEG}(\emptyset) = 0. \qquad \square$$

2.5 Reasons for providing a custom implementation

In this section, we summarize the reasons that led us to provide a custom implementation of the algorithm [2], some of which were already mentioned in the previous text. The main reasons are the following.

- We wanted an implementation, that would use the same underlying interval arithmetic library and all the structures we implemented, as in our implementation of our main solving algorithm from Chapter 3. We wanted to be able to call the topological degree computation directly from our code.
- We needed to adjust the input contracts to better suit our particular needs. See Subsection 2.2.3.

- We wanted to implement a possibility to execute a part of the combinatorial phase in parallel using threads.
- We thought, that it would be useful to allow more parametrization via a user interface. For instance, we allow to set the bisection ratio during the sign covering computation in the numerical phase or even the way of performing function evaluations (we offer three different interpreters, see Section 5.5).
- We wanted an implementation, that would consist of smaller and more maintainable pieces for easier testing, compared to the reference one [12].

The reference implementation [12] does not meet these requirement to our satisfaction. We used it mainly to test the correctness of our implementation and we compared their performances, see Section 6.3.

Chapter 3

Developing the solving algorithm

Here we describe all key steps and insights involved in the effort to develop our solving algorithm. We start with general ideas and follow by formulating the top level workflow. Then we present and analyze different specializations of its core components, that encapsulate the logic impacting the quality of the algorithm. These concepts are then captured into pseudocodes, which we subsequently use as guidelines for our implementation.

3.1 Problem formulation

Let us first remind ourselves of the problem we are solving. We are given an equation system $f_P(X) = 0$ of $n \geq 1$ equations and n real-valued variables from an input box $X \in I(\mathbb{R}^n)$, parametrized by $m \geq 1$ real-valued parameters from an input box $P \in I(\mathbb{R}^m)$. We are asked to characterize subsets of points $p \in P$, based on whether an $x \in X$ exists, such that $f_p(x) = 0$ holds. Our solution should be based on floating point arithmetic and be robust with respect to rounding errors.

3.1.1 Notation

Throughout this chapter, we implicitly assign the meaning of letters X, P, f_P to the one from the paragraphs above to prevent the constant repetition of restating it. In addition, we will use X', P' to express arbitrary sub-boxes of X and P, respectively. Further, we denote $F_{P'}$ an inclusion monotonic convergent interval extension function of $f_{P'}$ in the sense of Subsection 1.3.4 and refer to it simply as the inclusion function of $f_{P'}$.

We also implicitly suppose, that the box X and all its considered sub-boxes have the same implicitly given orientation. Its value is not really important. It

is just utilized by the algorithm for computing topological degree introduced in Chapter 2.

3.2 Basic approach and decisions

3.2.1 Choosing the way of characterization

Because both input sets X and P are entered as boxes, it makes sense to present our output using boxes as well. Boxes (Sections 1.2 and 1.4) are easy to implement and to work with, they posses little memory footprint and are suitable for recursive bisecting into sub-boxes, which allows to approximate areas of non-rectangular shapes with sufficient amount of small-enough boxes, see Figure 3.1.

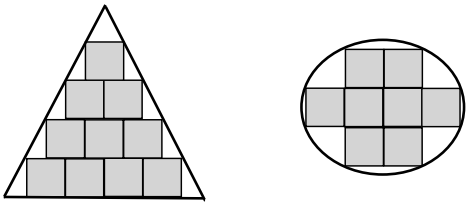


Figure 3.1: Approximation of areas of non-rectangular shapes by boxes in 2D space, where the boxes are simply rectangles.

We will therefore want the output of our algorithm in the form of two lists of boxes N and Y, that meet the following properties,

$$\forall P' \in N.\, P' \subseteq P \wedge \forall p \in P'.\, \forall x \in X.\, f_p(x) \neq 0, \tag{3.1}$$

$$\forall P' \in Y.\, P' \subseteq P \wedge \forall p \in P'.\, \exists x \in X.\, f_p(x) = 0. \tag{3.2}$$

In other words, we are grouping points $p \in P$ into boxes of two mutually exclusive types; one that contains points p, for which $f_p(x) = 0$ for some $x \in X$, and one, for which $f_p(x) \neq 0$ for every $x \in X$. Note, that property (3.1) is equivalent to

$$\forall P' \in N.\, P' \subseteq P \wedge 0 \notin f_{P'}(X). \tag{3.3}$$

Whenever we have enough information to put a box $P' \subseteq P$ into either Y or N, we say that P' is *decided*.

We want the lists Y, N to provide as much information as possible, meaning the more points from P are grouped into a box belonging to either Y or N, the better. We consider this the major criterion of a solution quality and restate it in a more formal way during our testing, see Subsection 6.5.3. For two solutions of similar quality, we could consider a minor criterion in the form of the sizes of the lists Y and N and prefer the one with lesser memory footprint—that is the one containing less but bigger boxes (in terms of volume).

3.2.2 Computing with boxes

We would like to use the convenient properties of boxes stated in 3.2.1 not only for presenting outputs, but also during all intermediate computations. Therefore we need some system of computation with boxes. Namely, we must have a way of obtaining box approximations of images $f_{P'}(X')$. To this end, we will use interval arithmetic (see Section 1.3), and perform all computations with an interval inclusion function $F_{P'}$ instead. Apart from providing a convenient way of approximating results $f_{P'}(X')$ with boxes, computing with $F_{P'}$ also bounds possible round-off errors, allowing the process of deciding $P' \in Y$ or $P' \in N$ to be sound wrt. rounding.

The biggest disadvantage of this approach is probably the over-estimation tendency of $F_{P'}$ (see Section 1.3.3). Approaches of how to counter-weight it are discussed throughout the rest of the chapter.

It is not our goal to provide out own implementation of interval arithmetic. Instead, we perform a survey on existing implementations and choose the one most suitable for our purposes, see Section 4.2.

3.2.3 The idea of showing that a box belongs to the N list

To decide whether $P' \in N$, it just suffices to confirm that $0 \notin f_{P'}(X)$ (property (3.3)). Working with $F_{P'}$ instead of $f_{P'}$ comes in handy here, because its inclusion property ensures, that $0 \notin F_{P'}(X)$ implies $0 \notin f_{P'}(X)$. In addition, testing $0 \notin F_{P'}(X)$ is cheap, as it only involves at most $2n$ comparisons between 0 and the endpoints of n components of $F_{P'}(X)$.

In general, however, the way $f_{P'}$ is prescribed together with the diameter of X may cause the overestimation of $F_{P'}$ to become big enough, so that $0 \in F_{P'}(X)$ will hold, while actually $0 \notin f_{P'}(X)$. In such cases, we may reduce the overestimation by considering a box paving $\bar{T}$ of X containing small enough sub-boxes of X, and evaluate $F_{P'}(X')$ for every $X' \in \bar{T}$. If none of the results $F_{P'}(X')$ contains zero, then also no $f_{P'}(X')$ contains zero, and because $\bar{T}$ is a box paving of X, it follows that $0 \notin f_{P'}(X)$.

Example 3.1. Consider the function $f_P(X) = x_1^2 - x_1 + p_1$ on $X = \left[\frac{5}{4}, 2\right]$, parametrized by $P = [0, 1]$. We want to show, that $P \in N$. To this end, we consider the natural inclusion function F_P of f_P (see Subsections 1.3.2 and 1.3.4) and compute

$$F_P(X) = \left[\frac{25}{16}, 4\right] - \left[\frac{5}{4}, 2\right] + [0, 1] = \left[-\frac{7}{16}, \frac{15}{4}\right]. \tag{3.4}$$

We got $0 \in F_P(X)$, which means no luck, so we pave X with $X_1 = \left[\frac{5}{4}, \frac{6}{4}\right]$

and $X_2 = \left[\frac{6}{4}, 2\right]$ and perform two separate evaluations.

$$F_P(X_1) = \left[\frac{25}{16}, \frac{36}{16}\right] - \left[\frac{5}{4}, \frac{6}{4}\right] + [0,1] = \left[\frac{1}{16}, 2\right], \tag{3.5}$$

$$F_P(X_2) = \left[\frac{36}{16}, 4\right] - \left[\frac{6}{4}, 2\right] + [0,1] = \left[\frac{1}{4}, \frac{7}{2}\right]. \tag{3.6}$$

This time, we have $0 \notin F_P(X_1)$ and $0 \notin F_P(X_2)$. Because $X = X_1 \cup X_2$ and F_P is an inclusion function of f_P, we have that $0 \notin f_P(X)$, and thus $P \in N$. □

3.2.4 The idea of showing that a box belongs to the Y list

To show that $P' \in Y$, we make use of the notion of topological degree and its property of solvability from Section 1.5.

For a given sub-box $P' \subseteq P$ we always try to compute a sufficient sign covering of an oriented boundary ∂X of X wrt. the given inclusion function $F_{P'}$ of the parametrized function $f_{P'}$ defined on X via the function signCovering (Section 2.2). On success, we pass such sign covering to the function deg, that computes $\deg(f_{P'}, X, 0)$ (Sections 2.3 and 2.4).

If $\deg(f_{P'}, X, 0) \neq 0$, then the solvability property ensures, that for every parameter value $p \in P'$, the function f_p, has a root in X, and therefore $P' \in Y$.

Instead of the domain X, we can again consider some box paving $\bar{T}$ of X. To show that $P' \in Y$, it is then sufficient if $\deg(f_{P'}, X', 0) \neq 0$ for at least one $X' \in \bar{T}$. However, while the process of deciding $P' \in N$ benefits from splitting X into smaller sub-boxes, it may not always be beneficial for showing that $P' \in Y$. We describe it in more detail in Section 3.3.

3.3 Manipulating the input domain box to infer the existence of a root

We have mentioned in Subsection 3.2.3, that splitting the input variable domain box X into some box paving $\bar{T}$ helps to decide, if a sub-box P' of the parameter domain P belongs to the list N satisfying (3.1).

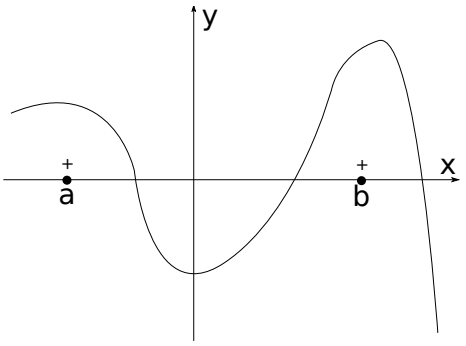
On the other hand, when deciding if P' belongs to the list Y (having the property (3.2)) using the solvability property of the topological degree as described in Subsection 3.2.4, splitting the domain X can both help and hurt. Let us demonstrate it on a simple example.

In Figure 3.2a, we show some arbitrary function $g\colon [a,b] \to \mathbb{R}$. Although $0 \in g([a,b])$, we get $\deg(g, [a,b], 0) = 0$, because $g(a) > 0$, $g(b) > 0$, and we cannot deduce the root existence from the solvability property.

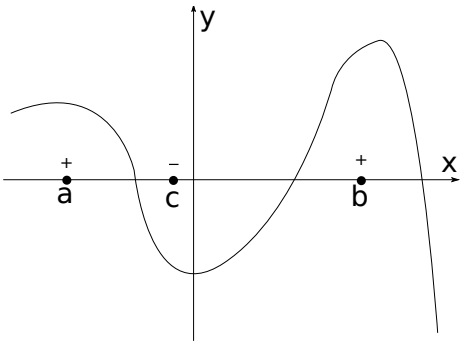
But a splitting of $[a,b]$ into $[a,c]$ and $[c,b]$, such that $g(c) < 0$, as depicted in Figure 3.2b, allows us to infer the existence of a root on $[a,b]$, because the degree on either of the subintervals is non-zero. The performed splitting was helpful here.

In Figure 3.2c, however, a different scenario is shown. Because $g(a)$, $g(b)$ have different signs different from each other, the degree $\deg(g, [a, b], 0)$ is non-zero and the solvability property can be immediately applied to show the root existence on $[a, b]$. Notice now, what would happen, if we would instead split this interval into $[a, c]$, $[c, b]$, where $g(c) = 0$ like the figure shows. Because c is zero, neither $\deg(g, [a, c], 0)$, nor $\deg(g, [b, c], 0)$ is defined and so this time, the splitting did not help us and sticking with the original interval $[a, b]$ was the key.

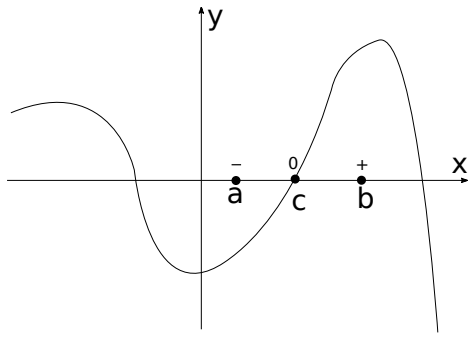
Within our problem domain, we in general work with parametrized functions. Figure 3.2d pictures a situation, where g is parametrized by some parameter box P. When deducing the existence of a root on $[a, b]$ for every parameter $p \in P$, the situation in Figure 3.2c comes to pass, whenever $g_p(c) = 0$ for at least one p.



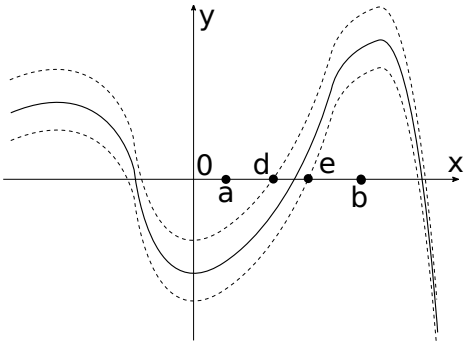
(a) The degree of $[a, b]$ is 0, because $g(a) > 0$, $g(b) > 0$.



(b) Splitting of $[a, b]$ with c. The degree of both $[a, c]$ and $[c, b]$ is non-zero.



(c) Different triplet of points a, b, c. Degree of $[a, c]$ and $[c, b]$ is undefined, because $g(c) = 0$.



(d) Function g is parametrized by some box P. Any $c \in [d, e]$ results in the scenario in 3.2c for some $p \in P$.

Figure 3.2: Manipulating the domain box to infer the existence of a root informally demonstrated on an arbitrary $\mathbb{R}$-valued function. Follow the text of Section 3.3.

As we now see, the difficulty lies in the fact that for an arbitrary sub-

box X' of X, the image of its boundary under the given input function $f_{P'}$ may contain zero, causing the particular topological degree not to be defined. Handling the domain box X therefore needs to be done more carefully, which we focus on next.

3.4 The frame data structure

When deciding sub-boxes of P, we will use the methods mentioned in Subsections 3.2.3 and 3.2.4. To this end, especially to deal with the issue described in Section 3.3, we introduce a generic data structure called *frame*. Frames are meant to encapsulate the logic of manipulating the input box X. Here, we describe their general properties, and provide concrete specializations later.

A frame instance is always created with respect to (using notation from Section 3.1)

- The input domain n-box X.
- A parameter sub-box P' of the input parameter m-box P.
- An implementation $F_{P'}$ of an inclusion function of the input function $f_{P'}$, defined on X.

Every frame stores some collection C of sub-boxes of X, also called the *items collection* of the given frame. There are no general requirements on the underlying structure used to store, access and modify C. The only mandatory things are, that its elements are allowed to be iterated over in a loop, and that it always satisfies the following property

$$0 \notin f_{P'}(X - |C|), \tag{3.7}$$

where $|C|$ denotes the union of all boxes in C. See also Figure 3.3.

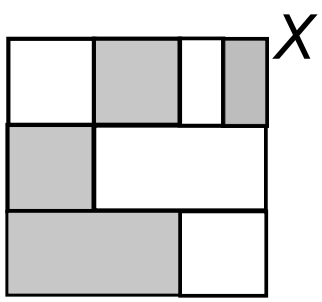
The motivation behind this is the typical way we want the frame's items collection to be processed, which consists of manipulating the set C, so that $|C|$ becomes progressively smaller by removing its subsets that are guaranteed not to contain a root under $f_{P'}$, which is the requirement expressed by (3.7). Notice, that (3.7) implies

$$C = \emptyset \implies P' \in N, \tag{3.8}$$

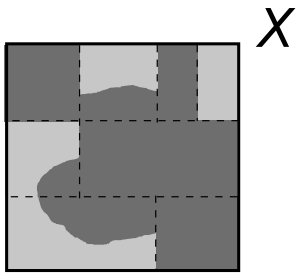
and this idea forms the basics of the process of determining that $P' \in N$.

3.4.1 The capacity of the frame's items collection

We assume that every frame's items collection has a certain fixed capacity of how many items it can store. In this chapter, we do not care about its exact value, and we suppose, that every frame ever created, within a given



(a) Actual sub-boxes (light gray) of X in frame's item collection C.



(b) Largest subset S of X (dark gray), s.t. $0 \notin f_{P'}(S)$. S must be a superset of $X - |C|$ (white areas in Fig. 3.3a).

Figure 3.3: Sample depiction of the core property (3.7) for frames.

algorithm run, has the same capacity. We just bear in mind, that the way of working with the frame structure may depend on whether it is able to store more items or not. We discuss this in more detail later. We deal with the concept of capacity mainly because we want to study the effect of limited resources on the quality of the results of our algorithms.

3.4.2 Working with frames

To work with frames, we use several functions, that can be divided into two categories. The first category contains functions through which we access some attributes or properties of frames. The second one contains functions that create new frames of certain properties.

The general contract of functions in the first category is in Table 3.1. Apart from noting, that the function frHasRoom is our interface to the capacity of the items collection wrt. Subsection 3.4.1, these require no further explanation, behave always the same and their code is only affected by the structural properties of the underlying items collection. We therefore do not list their specializations for concrete frame structure designs and just implicitly assume, that they are provided.

Table 3.1: Contract of functions retrieving properties of frames.

Function frPBox
Input: Frame *frame*.
Output: The underlying parameter box P', $P' \subseteq P$, this frame is associated with.

Function frFunc
Input: Frame *frame*.
Output: The underlying interval inclusion function $F_{P'}$ of $f_{P'}$ this

frame is associated with.

Function frItems
Input: Frame *frame*.
Output: The underlying items collection C of sub-boxes of X, that can be iterated over.

Function frEmpty
Input: Frame *frame*.
Output: Boolean b = true, if the underlying items collection is empty, otherwise b = false.

Function frHasRoom
Input: Frame *frame*; Integer i.
Output: Boolean b = true, if the underlying items collection of *frame* has enough capacity to store i more items (boxes), otherwise b = false.

The second category comprises of five functions named frInit, frPrune, frHasRoot, frRefine and frBisect. In Figure 3.4, we informally sketch the intended interaction between these functions when processing frames. Table 3.2 then shows the general contract of the functions in the second category.

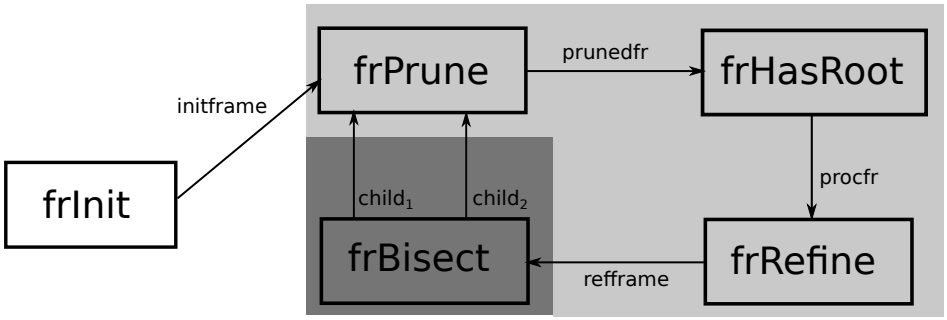


Figure 3.4: Informal sketch of the intended interaction between the methods manipulating frames. In the light-gray part, the items collections are processed and modified, while in the dark gray part, associated parameter sub-boxes are processed and bisected.

The idea is, that an algorithm which uses frames to solve a problem from our problem domain (Section 3.1) creates one initial frame using the function frInit at the very beginning and every other frame is then created by processing some other frame via the rest of the functions.

The function frBisect is used to create child frames of the same properties as the given parent frame, only associated with smaller parameter boxes. Working with the child frames instead of the original frame allows the inter-

val inclusion function to produce less overestimated results in general. This function is meant to work the same way for every specialized design of frames introduced later. We decided to always perform the bisection along the longest component of the particular box and with the given ratio passed as parameter.

Both functions frInit and frBisect require no further explanation and we simply assume, that they are provided and work the same for each specialized frame design.

This leaves us with three functions, that, along with the structural properties of items collection, encapsulate the behavior dictating the quality of result our algorithm provides—that is the portion of P it will be able to decide to belong to either Y or N. We decided to split the encapsulation into these three functions mainly for better maintainability. Their bodies are later covered for each frame design separately.

- Function frPrune. This is used for deciding $P' \in N$ in a way we sketched when we were discussing property (3.7), which we later elaborate on.
- Function frHasRoot. This function is meant to make use of the signCovering and deg functions (Sections 2.2 and 2.3) to look for a box X' in the items collection, such that $\deg(f_{P'}, X', 0)$ is non-zero, which would imply (using the solvability property of the topological degree), that $P' \in Y$. We also intentionally introduce the possibility to return a modified frame (*procfr*), as we think this may be useful in some specializations to express particular ideas.
- Function frRefine. This one is meant to be used in concrete designs to create frames with modified items collections, typically by bisecting some of its boxes, in order to provide more precise information for subsequent calls to frPrune and frHasRoot. This is also the function, whose behavior may depend on the capacity of the underlying items collection.

Table 3.2: Contract of functions manipulating frames.

Function frInit
Input: n-box X; m-box P; Implementation F_P of incl. function of $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ parametrized by P.
Output: New frame structure *initframe* associated with X, P and F_P with the items collection $C = \{X\}$.

Function frBisect
Input: Frame *frame*; Bisection ratio $0 < r < 1$.
Output: Pair of frames $(child_1, child_2)$ associated with $X, P'_1, F_{P'_1}$ and $X, P'_2, F_{P'_2}$, respectively, where P'_1 and P'_2 are results of bisecting P' along its longest component with the ratio r. Both $child_1$ and $child_2$

have the exact same items collection and all other properties as *frame*.

Function frPrune
Input: Frame *frame*.
Output: Frame *prunedfr* associated with the same $P', F_{P'}$ as *frame*, having no box X' in it items collection, for which $0 \notin F_{P'}(X')$.

Function frHasRoot
Input: Frame *frame* with the items collection C; Refinement threshold $ref \geq 0$ to pass to the signCovering function.
Output: Frame *procfr* associated with the same $X, P', F_{P'}$ as *frame* and whose items collection C' satisfies $|C| = |C'|$;
Boolean b, s.t. b = true implies that there exists some $X' \in C$, s.t. $\deg(f_{P'}, X', 0)$ is defined and non-zero.
Notes: This function is meant to use the signCovering and deg functions (Sections 2.2 and 2.3).

Function frRefine
Input: Frame *frame* with the items collection C.
Output: Frame *reframe* associated with the same $X, P', F_{P'}$ as *frame* and whose items collection C' satisfies $|C| = |C'|$.

3.4.3 Proving the correctness of the functions manipulating frames

When considering a concrete design of frames in the subsequent sections, we will need to show, that the following properties are fulfilled.

1. No manipulation with frames violates (3.7).
2. Functions from Tables 3.1 and 3.2 are designed to keep their respective contracts.

As for the first point, we intentionally made all the functions manipulating frames never to modify their inputs, but to return modified copies instead (if applicable). This means, that this verification is reduced only to the those spots, where new frames are created. Based on Table 3.2, there are five ways new frames can be created—as the results of calls to frInit, frBisect, frPrune, frHasRoot and frRefine.

For the result of frInit, property (3.7) holds trivially, because its items collection is $C = \{X\}$, and so $X - |C| = \emptyset$. For the resulting frames of frBisect, the property (3.7) holds, whenever it holds for its input frame, as no changes to the underlying items collection is made. These two functions thus do not need further examination. The other three functions are then inspected further in the text, for each concrete design individually.

With regard to the other point, we make sure to check the three core functions frPrune, frHasRoot and frRefine, as these are the ones whose bodies are specialized in each concrete frame design.

3.4.4 A remark on the finite precision of floating point representation

In practice implementations based on floating point arithmetic (which also includes our own implementation), a bisection of a box may not always succeed, because the box can be so small, that the limited precision of floating point representation could no longer distinguish between the parent box and the results of its bisection.

In this chapter, we do not consider such scenarios and assume implicitly, that every bisection succeeds, and postpone addressing this issue until our implementation, where the way of dealing with such scenario is to take the exact steps, as if the items collection were having insufficient capacity to store the bisection result (the two new child boxes).

3.5 Workflow of the parameter space decomposition using frames

Our approach is to use the previous sections and introduce a top-level function, whose general workflow can stay the same regardless of any concrete specialization of the frame data structure, so that the inner variable parts of the logic can be moved under these specializations as intended. Implementation-wise, this allows for better code usability and maintainability, which we consider to be the main benefit of such approach

At the core of our top-level function, we employ a branch and bound like technique on P using DFS, which realizes a systematic parameter space decomposition. The stack is manipulated in pre-order manner. Elements managed by the stack are instances of the frame data structure, introduced in Section 3.4. The manipulation of frames is designed wrt. Subsection 3.4.2, see especially Figure 3.4.

The initial frame is created by a call to frInit, passing in the whole box P itself, and the function frBisect is used afterwards to spawn child frames by bisecting associated sub-boxes of P. Depth of DFS is dictated by specifying the minimal possible width of sub-boxes of P.

When a frame *frame* is popped from the stack, its associated sub-box P' of P is examined using calls to frPrune and frHasRoot, in a hope to gain enough information to be sure, that either $P' \in N$, or $P' \in Y$. If this attempt is successful and we therefore know, that P' belongs either to the list N or Y, then any $P'' \subseteq P'$ must also belong to the same list as P' (see

Subsection 3.2.1), so we no longer need to bisect the frame via frBisect, and the DFS branch rooted at *frame* can be pruned (removed).

Algorithm 4 summarizes this top-level workflow in a pseudo-code.

Algorithm 4 The top-level branch and bound like parameter space decomposition.

Input: Implementation F_P of inclusion function for $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ parametrized by $P \subseteq \mathbb{R}^m$; Bisection ratio r; Box width threshold $d \geq 0$; Face refinement modifier $E > 0$.

Output: Lists of boxes N, Y satisfying 3.1 and 3.2, respectively.

```
1: function SOLVE(F_P, X, P, d, E)
2:     Y ← empty list, N ← empty list, st ← empty stack
3:     initframe ← frInit(X, P, F_P)
4:     push initframe onto st        ▷ root stack element containing initframe
5:     while st is not empty do
6:         frame ← pop frame from st
7:         P' ← frPBox(frame)                    ▷ extract param sub-box
8:         prunedfr ← frPrune(frame)
9:         if frEmpty(prunedfr) = true then
10:            append P' to N
11:        else
12:            ref ← getFaceRefThreshold(X, P, P', E)
13:            (procfr, b) ← frHasRoot(prunedfr, ref)
14:            if b = true then
15:                append P' to Y
16:            else if ω(P') > d then
17:                ▷ Underlying box P' is not decided, but still wide enough
18:                refframe ← frRefine(procfr)
19:                child1, child2 ← frBisect(refframe, r)
20:                push child1, child2 onto st
21:            end if
22:        end if
23:    end while
24:    return N,Y
25: end function

26: function getFaceRefThreshold(X, P, P', E)
27:    return (ω(P')/ω(P)) · (ω(X)/E)
28: end function
```

3.5.1 Remark on the face refinement threshold

Let us closer discuss the meaning of the parameter E in Algorithm 4. As we know from Subsection 3.4.2, frHasRoot is supposed to make use of the function signCovering to compute sign coverings (Algorithm 1). The function

SIGNCOVERING accepts the parameter *ref*, that dictates the extent to which the constructed sign covering is refined in order to create one that is sufficient.

During the execution of Algorithm 4, the widths of the processed sub-boxes of P as well as the sub-boxes of X in items collection of frames are meant to get smaller over time. Because of that, we wanted the value of *ref* to be computed adaptively to somehow reflect the ratios of the widths of currently processed sub-boxes and the widths of the input boxes. We also wanted it to be adjustable from the outside and its computation to be easy.

Our solution is to compute it as

$$ref = \frac{\omega(P')}{\omega(P)} \cdot \frac{\omega(X)}{E}, \tag{3.9}$$

which basically allows *ref* to become smaller, hand in hand with widths of parameter sub-boxes P' getting smaller. This dependence is then customizable by the passed modifier E. In Algorithm 4. we encapsulated (3.9) into the function GETFACEREFTHRESHOLD.

3.5.2 A discussion of correctness of the core algorithm

We need to be assured that Algorithm 4 is sound wrt. putting parameter sub-boxes into the lists N and Y, meaning that it never puts a box P' into a list, which it actually does not belong to. We assume, that all functions manipulating frames are correct wrt. Subsection 3.4.3.

Consider the contracts of the frame functions from Table 3.2. First of all, the output frames *frame*, *prunedfr*, *procfr*, *refframe* all have the same underlying parameter box P', because the only function, that creates frames with a different parameter box is FRBISECT. Therefore it is fine to extract P' from *frame* on line 7, and then use it on lines 10, 12, 15 and 16. In other words, if we would use a call to FRPBOX right before each of those lines, with the particular frames listed above as inputs, it would always yield the same box P'.

The rest is then simple. If FREMPTY(*prunedfr*) is true on line 9, then we have $P' \in N$ thanks to property (3.8). Correctness of line 15 then follows immediately from the contract of FRHASROOT and the solvability property of the topological degree.

3.6 Static frames

Static frames are our simplest specializations of the generic frame data structure. Its items collection can be based on any structure that allows adding new elements one by one and iterating over its elements in a loop. We usually think of it as a list (and this is the way we use in our actual implementation), but it is not necessary. No particular order of boxes in items collection is required.

The main characteristic of a static frame is, that boxes in its items collection never get bisected, which can be expressed by making its frRefine function to simply return its input frame with no changes. This means, that if we use the Algorithm 4 with static frames, we only ever work with frames, whose items collection is a one element list containing the input variable domain box X.

The specialization of the function frPrune for static frames can be think of as a base for more complicated designs introduced later. Here, it simply scans the items collection of the input frame and returns a new frame that has no such box in its items collection, whose image under $F_{P'}$ contains zero. The pseudocode is in Algorithm 5.

Algorithm 5 Basic version of frPrune. (Contract in Table 3.2)

```
1: function frPrune(frame)
2:     F_P' ← frFunc(frame)                    ▷ get the associated interval func.
3:     prunedfr ← a copy of frame with an empty items collection
4:     for each X' in frItems(frame) do
5:         if 0 ∈ F_P'(X') then add X' to items collection of prunedfr
6:     end for
7:     return prunedfr      ▷ prunedfr contains no boxes X', s.t 0 ∉ F_P'(X')
8: end function
```

The specialization of the function frHasRoot again serves as a base for more complex designs. The pseudocode is in Algorithm 6 and it captures the main idea of how to soundly decide, if the parameter sub-box P' associated with the given input frame belongs to the list Y. The input frame's items collection is iterated over in order to find a box X', such that $\deg(f_{P'}, X', 0)$ is defined and non-zero. To this end, the algorithm for computing the topological degree is used and it is executed using our custom adjusted workflow we introduced earlier (see Figure 2.4). (Functions signCovering, isSufficient and deg are parts of this workflow and are described in Section 2.2 and 2.3.)

You may notice, that the resulting frame *procfr* is identical to the input frame *frame*. In this version of frHasRoot, we simply do not make use of the possibility to return a modified frame, in compliance with the respective contract from Table 3.2.

3.6.1 Discussing the correctness of the frame manipulating functions

We need to make sure, that the introduced specializations of the frame manipulating functions are correct in terms of fulfilling both points from Subsection 3.4.3.

Algorithm 6 Basic version of frHasRoot. (Contract in Table 3.2)

```
1: function frHasRoot(frame, ref)
2:     F_P' ← frFunc(frame)                    ▷ get the associated interval func.
3:     procfr ← an exact copy of frame
4:     for each X' in frItems(frame) do
5:         bnd ← any oriented boundary of X'
6:         sc ← signCovering(bnd, F_P', ref)
7:         if isSufficient(sc) then
8:             if deg(sc) ≠ 0 then
9:                 return (procfr, true)          ▷ no need to examine the rest
10:            end if
11:        end if
12:    end for
13:    return (procfr, false)
14: end function
```

Function frRefine This one requires no discussion, as all it does is that it simply returns its unchanged input.

Function frPrune Consider Algorithm 5. The output contract of the function is met, because items collection of *prunedfr* contains no box X', such that $0 \notin F_{P'}(X')$.

As for the property (3.7), the core argument is, that Algorithm 5 copies every sub-box X', such that $0 \in F_{P'}(X')$ from input frame *frame* to its resulting frame *prunedfr*.

For a more detailed explanation, take into account Figure 3.5. Let C_1, C_2 denote items collections of *frame* and *prunedfr*, respectively, and assume *frame* satisfies the property (3.7), that is $0 \notin f_{P'}(X - |C_1|)$. Let S be an arbitrary subset of X, such that $S \subseteq X - |C_2|$. If also $S \subseteq X - |C_1|$, then $0 \notin f_{P'}(S)$, thanks to the assumption, that (3.7) holds for *frame*.

If on the other hand $S \nsubseteq X - |C_1|$, then let $S' = S \cap |C_1|$. Then for any box $X' \in C_1$, such that $X' \cap S' \neq \emptyset$ (like X'_1, X'_2 in Fig. 3.5), we necessarily have $X' \notin C_2$. But the only way for $X' \notin C_2$ to hold is, that $0 \notin F_{P'}(X')$, because of how the algorithm works. Inclusion property of $F_{P'}$ then implies $0 \notin f_{P'}(X')$. Because X' was arbitrary, we get $0 \notin f_{P'}(S')$, and therefore also $0 \notin f_{P'}(S)$, as $(S - S') \cap |C_1| = \emptyset$. Because S was also arbitrary, we have altogether, that (3.7) holds for *prunedfr* as well.

Function frHasRoot Consider Algorithm 6. If the input frame *frame* satisfies (3.7), then so does resulting frame *procfr*, because it is its exact copy.

As for the contract of the function, the only way of returning $b = \text{true}$ is by executing line 9. But if that happens, then it is assured that a sub-box X'

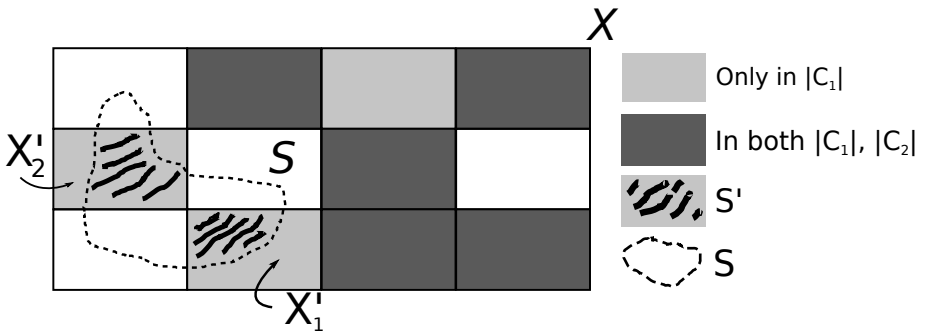


Figure 3.5: Visual support for discussing frPrune corectness in Subsection 3.6.1.

of the domain box X exists, such that $\deg(f_{P'}, X', 0)$ is non-zero thanks to lines 5 to 8, which form our workflow of executing the algorithm for computing the topological degree, discussed in Chapter 2. (See especially Figure 2.4.)

3.7 Bisect-only frames

A bisect-only frame design is a specialization almost identical to the static frame specialization. The only difference lies in the way the function frRefine is designed.

A bisect-only frame has its frRefine function designed, so that it simply bisects each box X' in the items collection of its input, and passes the bisection result X'_1, X'_2 to the items collection of the resulting frame, omitting the parent box X'. This is where its name originates from. It is a sort of opposite behavior a static frame has. There is no explicit rule dictating how exactly the bisection should be done. In our implementation, the default behavior is to bisect X' in its longest component with the ratio 0.5. This is also true for all subsequent specializations of this function presented later.

The intended goal of this approach is to reduce the overestimation tendency of interval evaluations by working with generally smaller boxes in the items collections of frames.

Function frRefine for bisect-only frames is in Algorithm 7. It also shows how to deal with a possibly finite capacity of the items collection (see also Subsection 3.4.1). It starts with bisecting every box in the items collection of the input frame, passing the resulting sub-boxes (which we also call the children) to the resulting frame. But if it is found, that the result could not hold all the children, the iteration simply falls back to a simple copying of non-bisected boxes. For keeping track of the remaining room, we use the function frHasRoom along with a counter variable c is that stores the number of additional room needed by the resulting frame compared to the input frame. The function ensures, that for every box X' in the items collection of the input

frame *frame*, the result *refframe* gets either both children X'_1, X'_2, or the parent X' (and no children).

Algorithm 7 frRefine for the bisect-only frames. (Contract in Table. 3.2)

```
1: function frRefine(frame)
2:     refframe ← a copy of frame with an empty items collection
3:     c ← 0            ▷ The total additional room ideally needed in refframe
4:     for each X' in frItems(frame) do
5:         c ← c + 1    ▷ replacing parent with 2 children needs 1 more room
6:         if frHasRoom(frame, c) then
7:             X'_1, X'_2 ← result of bisecting X'
8:             add X'_1, X'_2 to refframe
9:         else    ▷ no more room for children, just store the parent instead.
10:            add X' to refframe
11:        end if
12:    end for
13:    return refframe
14: end function
```

In a practical implementation, we expect a computation of Algorithm 4 based on bisect-only frames to give a better quality of the resulting list N than a computation based on static-frames, because working with smaller boxes in items collections should reduce the overestimation tendency of interval evaluations. On the other hand, the resulting list Y provided by the bisect-only approach might be of a poor quality for certain inputs due to the complete inability to undo a bisection and the fact, that a reckless bisection may not always be beneficial, as be pointed out in Section 3.3. We provide results of related practical experiments in Section 6.5.

3.7.1 Discussing the correctness of the frame manipulating functions

As in the previous section, we should make sure, that both points from Subsection 3.4.3 are fulfilled for the specializations of the frame manipulating functions.

Functions frPrune, frHasRoot These functions are identical to the ones of the static frame specialization, therefore see Subsection 3.6.1.

Function frRefine Consider Algorithm 7 and let the input frame *frame* satisfy (3.7). Denote C_1, C_2 the items collections of *frame* and *refframe*, respectively. The algorithm ensures, that for every box $X' \in C_1$, C_2 contains either both children X'_1, X'_2, or the parent X' itself. Because $X'_1 \cup X'_2 = X'$ (the definition of bisection), we have $|C_1| = |C_2|$, and so *refframe* satisfies (3.7).

This also means, that frRefine satisfies its output contract, as *reffframe* is a copy of *frame* in all other aspects.

3.8 Bisect-and-keep frames

Bisect-and-keep frames represent a slightly more complex specialization that is aimed to deal with the type of situations discussed in Section 3.3. This design builds upon the bisect-only frame design with the main difference, that the frRefine function adds parent boxes X' from the items collection of the input frame into the resulting frame along with their children.

To prevent bisecting the same box X' over and over in subsequent calls to frRefine, we assign a boolean flag *isLeaf* to every box in the items collection and bisect only those boxes, that have this flag set to true. After a bisection is performed, the children created from the parent X' will have the flag set to true, while the parent's flag will become false. We call a box with the flag set to true a *leaf*. This concept is demonstrated by Example 3.2.

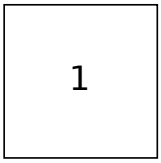
Example 3.2. Figure 3.6 show a sample life-cycle of a bisect-and-keep frame along a particular DFS branch in Algorithm 4, starting from the root. At the beginning, an initial frame is created by calling frInit(X, P, F_P), Figure 3.6a.

Assume that no boxes from the items collection are removed during the following call to frPrune and that the call to frHasRoot returns false. The next call to frRefine causes box 1 to be split into boxes 2 and 3. Box 1 is then no longer a leaf, Figure 3.6b.

Under the same assumptions, the next iteration leads to the situation in Figure 3.6c. Finally assume that the following call to frPrune removes all boxes but 1 and 3. This scenario is in Figure 3.6d. Notice, that the frame has no more leaves now, so any further refining will just return the same frame, with the items collection unchanged. □

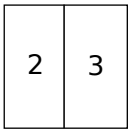
Consider now a particular frame *frame* with the underlying parameter box P'. Besides only ever splitting the leaves of *frame*, we also do not want to keep around those non-leaf boxes X' in its items collection, for which it was found during a call to frHasRoot, that $\deg(f_{P'}, X', 0)$ is zero. Why? Because their zero degree prevents them from being used to soundly decide that $P' \in Y$ via the solvability property, so they are useless in this way. Note that a removal of a non-leaf box does not violate the property (3.7), as it must have already been bisected into sub-boxes that were stored in the items collection.

To this end, we introduce a second boolean flag *rem* assigned to all boxes. Its value is always false for newly created boxes, and for a certain box X', it becomes true only when $\deg(f_{P'}, P', 0) = 0$ is found out to hold when scanning the items collection inside the frHasRoot function.



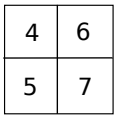
box	1
isLeaf	T

(a) The frame created by frInit.



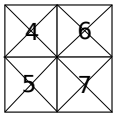
box	1	2	3
isLeaf	F	T	T

(b) Box 1 was bisected into 2 and 3 by frRefine and is no longer a leaf.



box	1	2	3	4	5	6	7
isLeaf	F	F	F	T	T	T	T

(c) A further bisection of leaves 2 and 3 by a subsequent call to frRefine.



box	1	3
isLeaf	F	F

(d) A call to frPrune removed all boxes except from 1 and 3. Because neither is a leaf, no other bisecting is performed in the following frRefine.

Figure 3.6: Sample life cycle of a bisect-and-keep frame along one particular DFS branch in Algorithm 4, starting from the root.

The function frRefine then checks the value of this flag of every box X' in the items collection and aims to remove X', if its *rem* flag is true. However, if X' is a leaf, it is removed only after it has been bisected into children. If the bisection cannot be performed due to an insufficient capacity of the items collection to hold the children, then X' is kept (despite of having *rem* set to true), as it would otherwise may violate the property (3.7). The actions performed on X' in frRefine based on the values of *rem*, *isLeaf* and the capacity of the items collection are listed in Table 3.3.

isLeaf	*rem*	sufficient capacity	bisect X'	remove X'
true	true	yes	yes	yes
false	true	yes	no	yes
true	false	yes	yes	no
false	false	yes	no	no
true	**true**	no	no	**no**
false	true	no	no	yes
true	false	no	no	no
false	false	no	no	no

Table 3.3: Possible actions taken when manipulating a box X' in the bisect-and-keep frame's item collection C based on the flags *isLeaf* and *rem* of X' and the capacity of C to hold the bisection results of X'. Notice that a leaf is never removed if the capacity of C is insufficient.

Structurally, we understand the assignment of *isLeaf* and *rem* flags to the box X' simply as an ordered triplet $(X', \mathit{isLeaf}, \mathit{rem})$, although it is not strictly

required. The items collection, apart from being able to store these triplets, requires no extra structural properties compared to bisect-only frames.

3.8.1 Pseudocode of the frame manipulating functions

Let us now focus on the code of the three main functions of interest, that is frPrune, frHasRoot and frRefine.

The function frPrune is simple. It ignores both *isLeaf* and *rem* flags and behaves exactly like for static and bisect-only frames, that is Algorithm 5.

Funcion frHasRoot also behaves like the one for the static and bisect-only design in Algorithm 6. The only exception is, that it sets the flag *rem* for boxes, for which a sufficient sign covering of their boundary wrt. $F_{P'}$ was computed and the corresponding topological degree was found to be 0. Flag *isLeaf* is ignored and just passed unchanged. This results in Algorithm 8

Algorithm 8 Function frHasRoot for bisect-and-keep frames. (Contract in Table 3.2)

```
1: function frHasRoot(frame, ref)
2:     F_P' ← frFunc(frame)
3:     procfr ← a copy of frame with an empty items collection
4:     for each (X', isLeaf, rem) in frItems(frame) do
5:         bnd ← any oriented boundary of X'
6:         sc ← signCovering(bnd, F_P', ref)
7:         if isSufficient(sc) then
8:             if deg(sc) ≠ 0 then
9:                 return (procfr, true)          ▷ no need to examine the rest
10:            else
11:                ▷ Mark X' for removal.
12:                add (X', isLeaf, true) to the items collection of procfr
13:            end if
14:        else
15:            add (X', isLeaf, rem) to the items collection of procfr
16:        end if
17:    end for
18:    return (procfr, false)
19: end function
```

Finally, the pseudocode for frRefine is in Algorithm 9. It utilizes the *isLeaf* and *rem* flags according to Table 3.3 and ensures, that for every box X' in the items collection of the input frame *frame*, the resulting frame *refframe* contains either X', or both its bisection results X'_1, X'_2, or all three of these boxes.

The counter variable c is used to keep the track of the additional room the resulting frame needs compared to the input frame. It may become negative,

should the resulting frame end up with less boxes in its items collection than the input.

Algorithm 9 Function frRefine for bisect-and-keep frames. (Contract in Table 3.2)

```
1:  function frRefine(frame)
2:      refframe ← a copy of frame with an empty items collection
3:      c ← 0                ▷ The total additional room ideally needed in refframe
4:      for each (t ← (X′, isLeaf, rem)) in frItems(frame) do
5:          if isLeaf = true then
6:              if rem and frHasRoom(frame, c + 1) then
7:                  c ← c + 1          ▷ t is not kept and one children replaces it
8:                  X′1, X′2 ← the result of bisecting X′
9:                  add X′1, X′2 to refframe
10:             else if not rem and frHasRoom(frame, c + 2) then
11:                 c ← c + 2                    ▷ t is kept along with the children
12:                 add t to refframe
13:                 X′1, X′2 ← the result of bisecting X′
14:                 add X′1, X′2 to refframe
15:             else                          ▷ no more room to store the children
16:                 add t to refframe
17:             end if
18:         else
19:             if rem = true then
20:                 c ← c − 1                                  ▷ one room freed
21:             else
22:                 add t to refframe
23:             end if
24:         end if
25:     end for
26:     return refframe
27: end function
```

3.8.2 Limitations of the bisect-and-keep approach

Let us return back to the Example 3.2. It indicates the most obvious limitation of bisect-and-keep design approach.

First, if a box X' from items collection is about to be removed in frPrune, there is no way to also remove all of its sub-boxes X'' without the need of (pointlessly) evaluating $F_{P'}(X'')$ (P' is the underlying parameter box associated with the frame). This is because there is no underlying parent-child relationship structure and no order, that would guarantee to evaluate the parent before its children. This is the situation of the parent box 2 and its

children $4, 5$ in Figure 3.6. We only know, which boxes have not yet been bisected (leaves), but that is it.

Secondly, if the items collection contains some box X' and some other boxes $X'_1, \ldots, X'_p$, that form a box paving $\bar{T}$ of X', then as soon as all the boxes from $\bar{T}$ are removed, the X' itself can be removed as well. But as in the previous situation, there is no underlying structure that would allow us to do so without explicitly evaluating $F_{P'}(X')$. But what is worse, this time, X' may not be removed at all, because the over-approximation of $F_{P'}$ may cause $0 \in F_{P'}(X')$ to actually hold. This is the situation of boxes $3, 6, 7$ in Figure 3.6—although 6 and 7 got removed in Figure 3.6d, box 3 did not.

3.8.3 Discussing the correctness of the frame manipulating functions

Function frRefine As the function frRefine works the same for bisect-and-keep frames as for static and bisect-only frames, arguments presented in Subsection 3.7.1 apply with regard to its correctness.

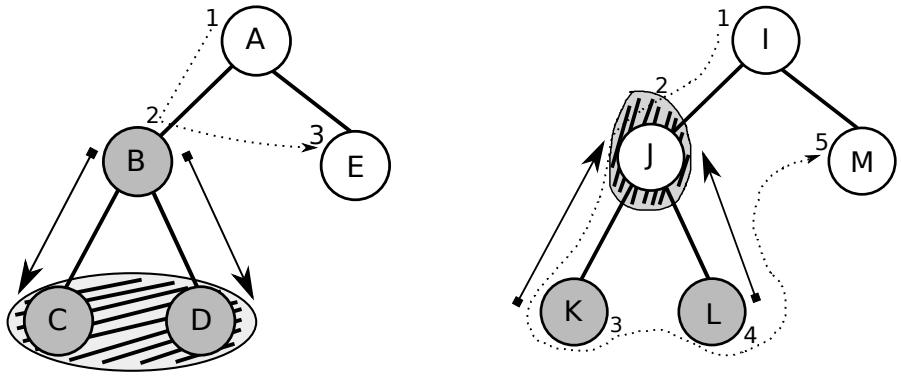
Function frHasRoot Arguments from Subsection 3.7.1 are also valid for frHasRoot function, because its specialization for bisect-and-keep frames possibly only modifies flags of the boxes in the items collections and otherwise keep them unchanged, which does not affect the output contract of this function, nor the validity of property (3.7).

Function frRefine Correctness of this function is also supported by the content of Subsection 3.7.1. The key point is, similar to the case of bisect-only frames in Section 3.7, that the resulting frame *refframe* always contains either the box X' from items collection of the input frame, or both its bisection results X'_1, X'_2, $X'_1 \cup X'_2 = X'$, or all three of these boxes.

By introducing the counter variable c in Algorithm 9 to monitor the additional room the resulting frame will need, we ensure in each iteration of the for loop that there is always a room to copy the parent box X' into the items collection of the result, if needed.

3.9 Frames based on trees

This approach structurally enhances the bisect-and-keep design by capturing the whole parent-child relationship of boxes in the items collection, giving us more ways to manipulate it. We therefore see the items collection as a tree, whose nodes are identified with the sub-boxes of the input domain box X. The tree is initially rooted at the input domain box X itself. Child nodes are then results of bisecting sub-boxes of X in the frRefine function. In this



(a) The forward pruning at B removes the nodes C, D.

(b) The backward pruning of node J allows to remove it, although $0 \in F_{P'}(J)$.

Figure 3.7: Forward and backward prunings in tree frames. Node X' is colored, iff $0 \notin F_{P'}(X')$. The dotted arrow shows in which order nodes are visited during a pre-order traversal and evaluated with $F_{P'}$.

section, we use the term *node* as a synonym to an arbitrary box from the items collection.

Using a tree-like structure, we address the limitations from Subsection 3.8.2. When a node X' is about to be removed in the function frPrune, we can immediately remove all sub-boxes of X', using only the structural information without the need of evaluating them with $F_{P'}$—so called *forward pruning* at X'. And the other way, once all sub-boxes of X' have been removed, we may safely remove X' itself as well—again, without the need of evaluating it with $F_{P'}$. This is called the *backward pruning* of X'. The typical way we iterate through our tree based items collection, in order to carry out both of these operations, is by performing a pre-order traversal starting from the root. See Figure 3.7 for an example.

The previously discussed designs did not require support for deleting items from their items collections, because their modification could always be expressed in terms of creating a new resulting frame, and then iterating through the original one, adding only suitable items to the result. Here, we want the items collection to support an actual item removal, mainly to comfortably express and then further implement the backward pruning.

The *isLeaf* flag from the bisect-and-keep design is obsolete here, as it can be figured out from the tree structure itself. The other flag, *rem*, is used in the same way.

3.9.1 Note on terminology

We base our tree related terminology on [13, p. 279–280], namely the notions of a *child*, *leaf*, *descendant* and *ancestor*, but we believe these are quite in-

tuitive and generally known. We therefore present them only via examples using Figure 3.7. Nodes B, E are children of A. Node A has four descendants B, C, D, E, nodes C and D are not its children. Nodes B and A are ancestors of nodes C, D. Nodes C, D, E are leaves and do not have any descendants.

Since we identify the tree nodes with sub-boxes of domain box X they are associated with, we may say, that Z is a child of X', if Z is one of the two results of bisecting X'. We will also denote $\text{DESC}(X')$ the set of all descendants of node X', so for instance $\text{DESC}(B) = \{C, D\}$.

3.9.2 Pseudocode of the frame manipulating functions

The functions frHasRoot and frRefine are practically the same as for the bisect-and-keep frames, with the only difference that a tree instead of a simple list is traversed, so implementations must ensure correctly rebuilding the parent-child relationship in the resulting frames. Also no boolean flag is required to tell, whether a node is a leaf. We may therefore refer to Subsection 3.8.1. (The other boolean flag *rem* is used in the same way.)

What is different is the frPrune function, which is, as already stated, designed to address the content of Subsection 3.8.2. The main idea is as follows. The function traverses the tree from the root in a pre-order manner and if it finds out, that $0 \notin F_{P'}(X')$, it performs the forward pruning at X', removing X' as well as $\text{DESC}(X')$.

If the forward pruning could not be performed, X' is still observed further. If X' was originally not a leaf, but has become one after its descendants had been processed, then the backward pruning of X' can be done.

The pseudocode is in Algorithm 10. A significant work is done by the procedure process, which is recursive and modifies its *prunedfr* argument in place.

3.9.3 Discussing the correctness of the frame manipulating functions

Out of our three functions of interest, frPrune, frHasRoot and frRefine, only frPrune differs from its version designed for bisect-and-keep frames, so we focus our attention on this one.

Consider Algorithm 10 and first, let us focus on the output contract of frPrune from Table 3.2. We need to show, that the resulting frame *prunedfr* contains no node X', such that $0 \notin F_{P'}(X')$. Scanning through the algorithm, we see, that line 10 ensures, that every visited node X' (i.e. a node X' passed as input *node* to process) is removed from *prunedfr*, if $0 \notin F_{P'}(X')$.

What about a node Z', that does not get visited (i.e. Z' is not passed as input to process procedure)? Well, for this to happen, Z' must be removed on line 10 as a member of $\text{DESC}(Z)$ for a particular ancestor node Z, because

Algorithm 10 frPrune for tree frames. (Contract in Table 3.2)

```
 1: function frPrune(frame)
 2:     prunedfr ← an exact copy of frame incl. the items collection.
 3:     root ← the root of the items col. of prunedfr
 4:     F_P' ← frFunc(frame)
 5:     process(prunedfr, F_P', root)
 6:     return prunedfr
 7: end function

 8: procedure process(prunedfr, F_P', node)
 9:     if 0 ∉ F_P'(node) then
10:         remove desc(node) ∪ {node} from prunedfr      ▷ forward pruning
11:     else if node is not a leaf then
12:         for each child child of node do
13:             process(prunedfr, F_P', child)
14:         end for
15:         if node is leaf then
16:             remove {node} from prunedfr               ▷ backward pruning
17:         end if
18:     end if
19:     ▷ else keep the leaf node
20: end procedure
```

lines 12 to 14 otherwise ensure visiting every child of a node, that did not get removed. But then $0 \notin F_{P'}(Z)$, which implies $0 \notin F_{P'}(Z')$, as we suppose, that function F is inclusion monotonic (Subsection 3.1.1). Therefore the output contract of frPrune is satisfied.

As for the property (3.7), we want to show, that no node Z' in the input frame *frame*, such that $0 \in F_{P'}(Z')$, gets removed from the result *prunedfr*. Then the same arguments as in 3.7.1 can be used to justify the correctness of frPrune wrt. (3.7). But this just requires the same argumentation as above. The only way for a node Z' to be removed is by executing line 10 for either Z' itself, or some of its ancestor Z. This necessarily means, as already stated, that $0 \notin F_{P'}(Z')$.

3.10 Frames based on grids

Frames based on grids, or *grid frames* for short, offer a more advanced approach. The core feature we provided in this specialization is the ability to temporarily join the boxes in the items collection into oriented cubical sets, which gives more possibilities to soundly decide, that a certain parameter subbox P' belongs to the list Y. The idea of joining is inspired by [14, p. 10–11].

Let us start with an informal introduction. A grid frame's items collection is indeed a grid of sub-boxes of the input domain box X, that can be depicted as in Figure 3.8a. We use the term *grid* as a synonym for the items collection of a grid frame. Unlike a tree farme (Section 3.9), a grid frame does not directly introduce the parent-child relationship structure among boxes in is items collection. In fact, boxes in its items collections that are bisected are always replaced by the results of the bisection, so the parents are not stored at all—like for the bisect-only frames (Section 3.7). Instead, the grid captures a relationship of neighbourhood between its boxes. This is intuitively shown in Figure 3.8a with dashed arrows.

The neighbourhood relationship can be exposed to temporarily join adjacent boxes into oriented cubical sets. This is useful when dealing with a box X' from the grid, for which we are unable to find a sufficient sign covering of its oriented boundary wrt. $F_{P'}$ (either because it does not exist, or $F_{P'}$ overapproximates too much).

In such case, we can join X' with some of its neighbours (ideally along the problematic sub-faces, that could not be sufficiently covered), forming an oriented cubical set Ω. We then try to compute a sufficient sign covering of an oriented boundary of Ω instead. If we are still unlucky, we continue this process, examining and possibly adding neigbours of boxes currently in Ω, that have not been added yet. We stop, as soon as an oriented cubical set with sufficient sign covering of its oriented boundary is obtained, or there are no suitable neighbours to join with. Figure 3.8b visualizes a sample oriented cubical set in the grid and Figure 3.9 depicts one simple step in such joining.

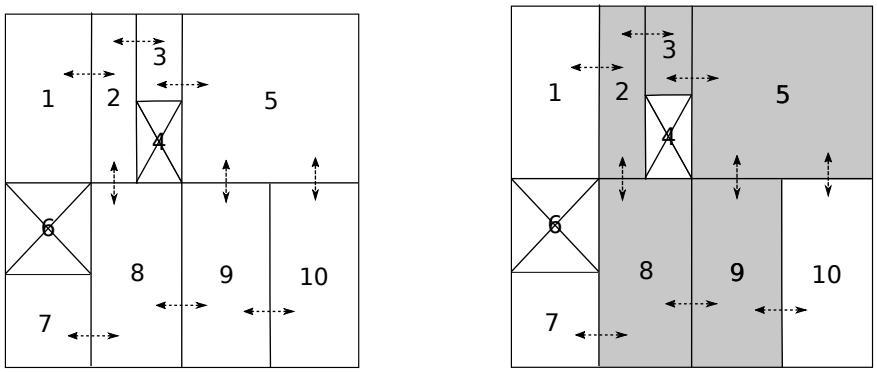
We can summarize that the goal of joining is to temporarily replace each box X' in the items collection with the smallest oriented cubical set (in terms of box count), for which we are able to find a sufficient sign covering of its oriented boundary.

To be able to algorithmically express these ideas, we need to capture the concept of neigbourhood more formally, which we focus on next.

3.10.1 The concept of neighbourhood and edges

Here we focus our attention on the concept of neighbouring boxes and edges in the grid. An edge should be an entity capturing the fact, that two boxes in the grid are neighbours and it should store information useful for potential joining of the neighbours and their boundaries.

The way we decided to design edges is visualized in Figure 3.10. Edges connecting actual boxes are actually pairs of two connections, one originating from the first box and one originating from the other. This way, there is no ownership ambiguity, as each neighbour owns one edge from the pair. As indicated in the figure, the connections are meant to be realized via sub-faces of oriented faces of the neighbours.



(a) Sample grid. (b) Or. cubical set of boxes 2, 3, 5, 8, 9.

Figure 3.8: Sample depiction of the grid of a grid frame. Arrows depict adjacency between boxes. Crossed out boxes are no longer part of the grid. Fig. 3.8b shows an example of a or. cubical set, that can be created by exposing the adjacency of boxes.

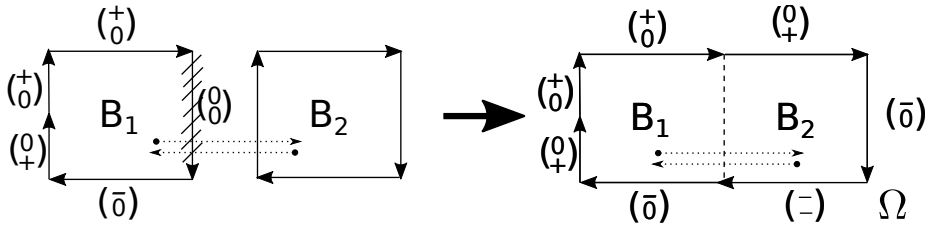


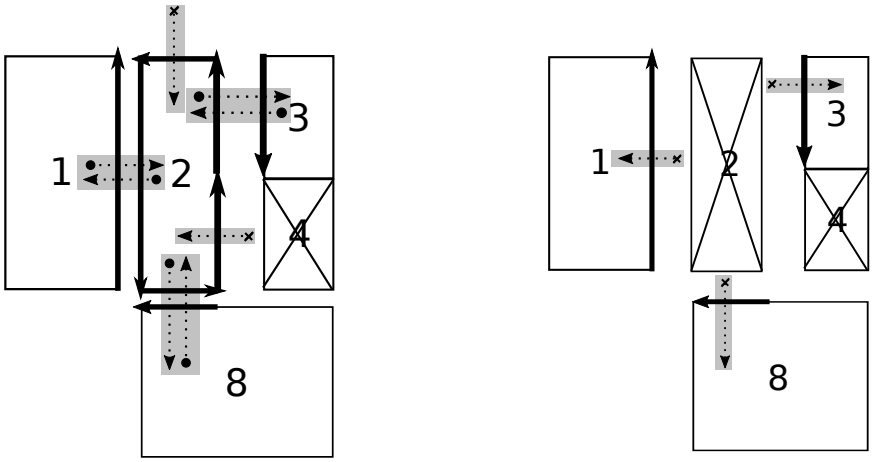
Figure 3.9: Joining of two boxes in the grid into an oriented cubical set. We see that a sign covering of an oriented boundary of B_1 was computed, but it is not sufficient. B_1 is therefore joined along the problematic face with B_2, forming an or. cubical set Ω. A sign covering of $\partial\Omega$ is then computed instead.

The figure also shows our intention to be able to express, that a box in the grid might not always be fully surrounded by other boxes. Initially, this is the case of the boxes lying on the margin of the grid. But if some boxes are removed from the grid, this becomes true for other originally inner boxes as well (compare Fig. 3.10a with 3.10b). To this end we introduce special edges without endpoints, so that they fill those "no-neighbour gaps". The idea behind this becomes clearer later.

Let us now formalize the concept of edges. Let a frame with grid G is given and let the dimension of the boxes in G be n. For a given box $X' \in G$, we call an oriented sub-face of some oriented face of X' simply a *wall of* X'. The dimension of any wall is therefore $n - 1$.

An *edge of* $X' \in G$ is either

- Any pair (b, Z), where $Z \in G$ and b is a maximal (in terms of set



(a) A closeup of box 2 from Fig. 3.8.

(b) A removal of box 2 from the grid.

Figure 3.10: Edges in a grid of a grid frame. Full arrows are oriented subfaces, dotted ones symbolizes edges. Boxes are connected by a pair of edges. Arrows with a crossed endpoint represent an edge without endpoint. When a box is removed, edges need to be updated.

inclusion) wall of X', such the box $c = b$ with the orientation opposite to b is a wall of Z. We then define

$$\textsc{opposite}(b, Z) = (c, X'). \tag{3.10}$$

- Any pair $(b, \emptyset)$, where b is a maximal (in terms of set inclusion) wall of X', s.t. the intersection of b with any box in the grid other than X' has dimension at most $n-2$. We call such edge as an *edge without endpoint*.

We introduce edges without endpoints, so that for any box X' from the grid the following holds,

$$\bigcup_{(b,Z)\in\textsc{edges}(X')} b = \partial|X'|, \tag{3.11}$$

where $\textsc{edges}(X')$ denotes the set of all edges of X'.

When later expressing our considerations in forms of pseudocodes and when writing an actual implementation, we also find it useful to be able to divide the edges of X' into categories, based on (informally) along which component and direction they realize the connection between X' and its neighbours.

This idea is sketched in Figure 3.11 and formalized as follows. Consider $i \in \{1, \ldots, n\}$ and let $[a_1, a_2]$ be the i-th non-degenerated component of X'. We then denote $\textsc{loEdges}(i, X')$ the set of all edges of X' with the wall having their

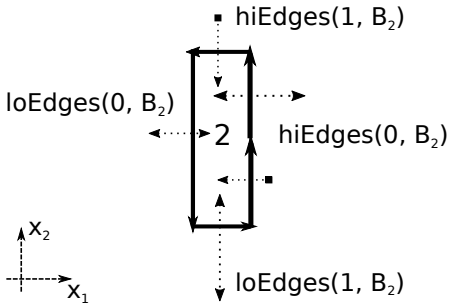


Figure 3.11: Edge categories in the grid of a grid frame, based on along which component and direction they make the connection. Box 2, referred to as B_2 here, from Fig. 3.10 is shown.

i-th component degenerated and equal to a_1, and similarly $\textsc{hiEdges}(i, X')$ the set of all edges of X', with the wall having their i-th component degenerated and equal to a_2. Let us also demonstrate this on a example.

Example 3.3. Consider again Figure 3.10a. Suppose that each box mark i is mapped to an actual box B_i as follows.

mark	box	mark	box
1	$B_1 := [-1, 0] \times [0, 1]$	3	$B_3 := [1, 2] \times [1/2, 1]$
2	$B_2 := [0, 1] \times [0, 1]$	4	$B_4 := [1, 2] \times [0, 1/2]$
8	$B_8 := [0, 2] \times [-1, 0]$		

Then we have (omitting the orientation of boxes for simplicity)

$$\begin{aligned}
\textsc{loEdges}(1, 2) &= \{([0] \times [0, 1], B_1)\},\\
\textsc{hiEdges}(1, 2) &= \{([1] \times [0, 1/2], \emptyset), ([1] \times [1/2, 1], B_3)\},\\
\textsc{loEdges}(2, 2) &= \{([0, 1] \times [0], B_8)\},\\
\textsc{hiEdges}(2, 2) &= \{([0, 1] \times [1], \emptyset)\}.
\end{aligned}$$

□

We now focus in more detail on how we specialize the frame manipulating functions in the grid frame design.

3.10.2 Initialization and pruning

For grid frames, the initial call $\textsc{frInit}(X, P, F_P)$ creates a grid with only one box, X itself, having $2n$ edges without endpoints. Each wall in those edges is actually an oriented face of X.

Unlike for tree frames, the function frPrune for grid frames is not based on any particular order, in which boxes are iterated, and is in fact the same as for static and bisect-only frames, i.e. Algorithm 5. Implementations only need to take care of correctly turning any edge (b, X') of an adjacent box

Z into $(b, \emptyset)$, if X' is the box about to be removed, just like it is shown in Figures 3.10a and 3.10b.

In a pseudocode, such edge updating could be for example expressed as in Algorithm 11. Notice, that because each substituting edge (line 8) has the same wall as the substituted one (line 7), meaning that if the neighbours of X' satisfied (3.11) before processing X', they continue to do after the processing of X'.

Algorithm 11 Updating edges in the grid of a grid frame before removing a box.

Input: A box X' from the grid of a given grid frame.

```
 1: procedure DETACH(X')
 2:     for each (b, Z) in EDGES(X') do
 3:         if Z ≠ ∅ then                          ▷ this edge has an endpoint
 4:             (c, Z') ← OPPOSITE(b, Z)
 5:             assert Z' = X'
 6:             ▷ replace the original edge with a one without endpoint
 7:             remove (c, Z') from EDGES(Z)
 8:             add (c, ∅) to EDGES(Z)
 9:         end if
10:     end for
11: end procedure
```

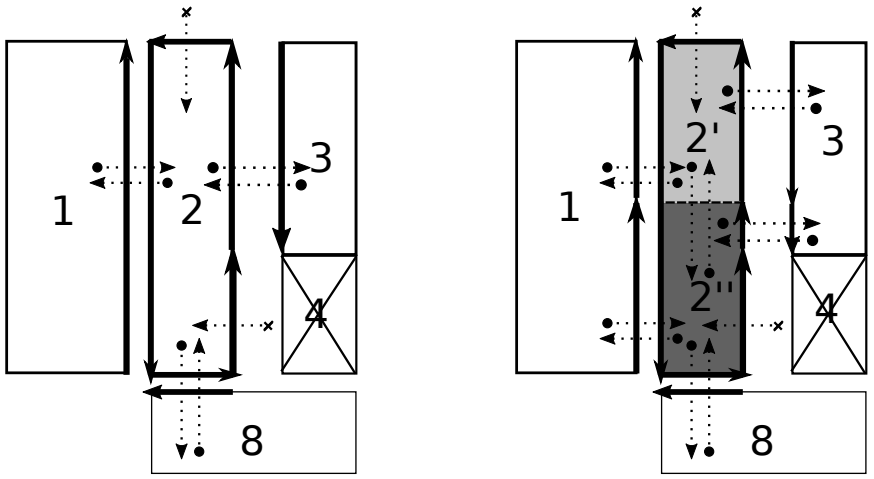
3.10.3 Bisecting boxes in the grid

Bisecting boxes in the grid of grid frames by the frRefine function works the same as for items collection of bisect-only frames, that is Algorithm 7. If the grid has enough capacity, then a box X' is bisected into sub-boxes X'_1, X'_2 and they are stored instead of X'. If the capacity is insufficient for storing those two child boxes, X' is kept instead.

However, the edge manipulation associated with the bisection, that implementations need to deal with, is a little more complex here than during a box removal (Subsection 3.10.2), so we better analyse in more detail, what happens with edges during a bisection of some box in the grid.

Figure 3.12 depicts this visually. The goal is to correctly distribute all edges of the parent box X' between its children X'_1, X'_2 and update edges of adjacent boxes accordingly. Some edges may get replaced by a pair of new ones, associated with smaller walls (in terms of set inclusion). The children also must get connected to each other.

More formally, consider a box X' from the grid. Let $I_1, \ldots, I_m$ be its non-degenerated components and for $j \in \{1, \ldots, m\}$, let $I_j = [a_j, b_j]$. Choose $i \in \{1, \ldots, m\}$ and suppose that the i-th non-degenerated component of X' is



(a) Before the bisection. (b) After bisecting box 2 into $2'$ and $2''$.

Figure 3.12: Bisection of a box in the grid of a grid frame. Same legend as in Fig. 3.10 is used. After a bisection, edges must be correctly redistributed to the children. Some edges may be replaced by a pair of new ones (like between boxes $2, 3$ and $2, 1$). The children must also get connected to each other.

about to be bisected into $[a_i, s]$ and $[s, b_i]$. The edges of X' must be updated considering the following three points.

1. The results of the bisection are boxes
$$\begin{aligned} X'_1 &= I_1 \times \cdots \times I_{i-1} \times [a_i, s] \times I_{i+1} \times \cdots \times I_m, \\ X'_2 &= I_1 \times \cdots \times I_{i-1} \times [s, b_i] \times I_{i+1} \times \cdots \times I_m. \end{aligned} \tag{3.12}$$

2. For the i-th component, we have
$$\begin{aligned} \textsc{loEdges}(i, X'_1) &= \textsc{loEdges}(i, X'), \\ \textsc{hiEdges}(i, X'_2) &= \textsc{hiEdges}(i, X'), \\ \textsc{hiEdges}(i, X'_1) &= \{(d, X'_2)\}, \\ \textsc{loEdges}(i, X'_2) &= \{(e, X'_1)\}, \end{aligned} \tag{3.13}$$
where both d and e are boxes $I_1 \times \cdots \times I_{i-1} \times [s] \times I_{i+1} \times \cdots \times I_m$ with mutually opposite orientation.

3. For every other component $k \neq i$, let $(d, Z) \in \textsc{loEdges}(k, X')$. Let the i-th component of wall d is $[v, w]$. Then
$$\begin{aligned} w \leq s &\implies (d, Z) \in \textsc{loEdges}(k, X'_1), \\ s \leq v &\implies (d, Z) \in \textsc{loEdges}(k, X'_2), \\ v < s < w &\implies (d_1, Z) \in \textsc{loEdges}(k, X'_1) \\ &\quad \wedge (d_2, Z) \in \textsc{loEdges}(k, X'_2), \end{aligned} \tag{3.14}$$

where d_1, d_2 are the results of bisecting d along its i-th component. Wall d_1 has $[v, s]$ as its i-th component, and d_2 has $[s, w]$. The very same also applies for the edges from hiEdges(k, X'). The situation $v < s < w$ is the case of edges between boxes $2, 3$ and $2, 1$ in Figure 3.12.

Edges of adjacent boxes then need to be updated in the same manner. Typically, as one loops through the edges of X' updating them, calls to opposite are meant to be used to mirror the updates in opposite edges.

3.10.4 The existence of a root

Finally, the frHasRoot function is the most innovated function of the grid frame specialization. The idea is to temporarily replace each box X' in the items collection with the smallest oriented cubical set (in terms of box count), for which a sufficient sign covering of its oriented boundary was found. Let us discuss such this process of creating oriented cubical sets.

Suppose a grid frame, whose associated parameter box is P'. The workflow of frHasRoot can be expressed as follows. Boxes from the grid are iterated over and for each particular box X', we consider all of its edges—that is the set edges(X'). We then take each edge (b, Z) from this set and compute a sign covering of the wall b wrt. $F_{P'}$. If we union these sign coverings into one set, we get a sign covering of some oriented boundary of X', because we intentionally designed the concept of edges, so that (3.11) holds.

Now this sign covering might not be sufficient, but by creating it as a union of the coverings of the walls, we can easily localize those walls, that could not be covered sufficiently, and join the corresponding neighbouring boxes to X' creating an oriented cubical set Ω. Then, we can instead attempt to compute a sufficient sign covering of some oriented boundary $\partial\Omega$ of Ω and we do not have to do this from scratch, as we can reuse all the former sign coverings of the walls of X', that were sufficient.

The process then similarly continues by examining the newly joined boxes by trying to find sufficient sign covering of their walls possibly joining other neighbours if such coverings are not found. Figure 3.13 visualizes the process.

When no more boxes can be added this way, the joining ends. To put it more formally, it ends when no box X' in the constructed oriented cubical set has an edge (d, Z), such that Z is not in the set and no sufficient covering of d was found. At that point we end up with some resulting oriented cubical set Ω and we examine the constructed sign covering of its oriented boundary. Two situations can follow.

- The sign covering is sufficient and we can follow by computing the $\deg(f_{P'}, \Omega, 0)$ using the function deg from Section 2.3.
- The sign covering is not sufficient, which happens if we reached the end of the grid during the construction of Ω. In other words, there was an

edge whose wall did not have a sufficient sign covering, but the edge had no endpoint we could join. The degree will not be computed in this case.

If the degree could not be computed or is zero, we restart the whole process by moving to some other not yet visited box. If there is no such box, frHasRoot ends, unable to soundly decide, if $F_{P'}$ has a root in the input domain X.

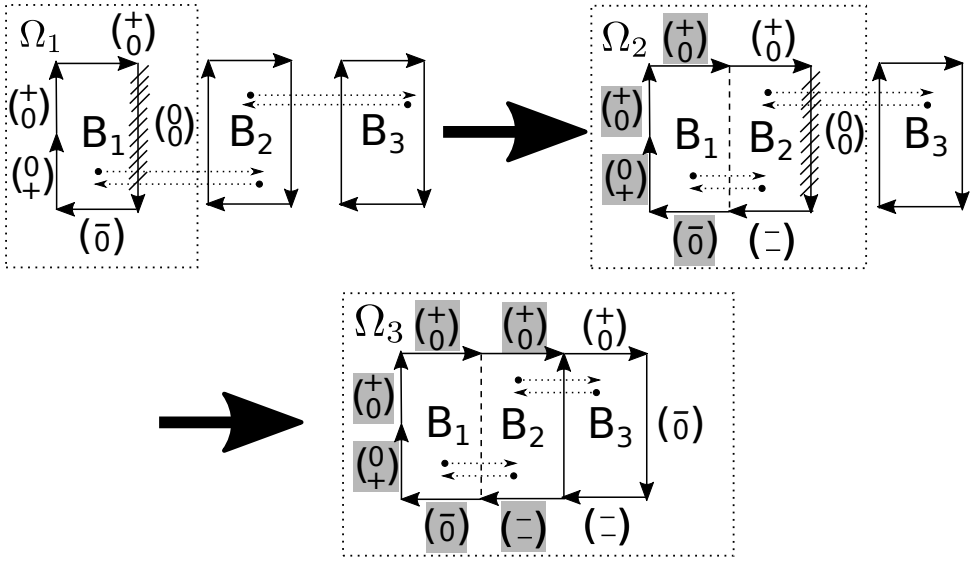


Figure 3.13: Example of joining of boxes in the grid. The initial box here is B_1. Each of its walls (depicted as arrows) is covered separately. The covering of one wall of B_1 is not sufficient, so B_1 is joined to its neighbour B_2, forming an oriented cubical set Ω_2. We then compute a sign cov. of $\partial\Omega_2$ instead, but can reuse the gray-colored coverings. This covering is again not sufficient, so a neighbour B_3 of B_2 is joined forming or. cubical set Ω_3.

We can summarize, that by such process, the items collection gets partitioned into non-overlapping oriented cubical sets and the process stops, as soon as an oriented cubical set Ω for which $\deg(f_{P'}, \Omega, 0)$ was found out to be defined and non-zero is found, or until we exhaust boxes to construct the partitions from.

The pseudocode for function frHasRoot utilizing all the ideas from this subsection is in Algorithm 12. The inner loop represented by the function join takes care of constructing an oriented cubical set from the initial box X' as described above.

Notice especially lines 27 to 31. They dictate under which conditions will another box be added to the constructed oriented cubical set *cubSet*. The following are the two cases in which the currently examined endpoint will not be added to the oriented cubical set.

- $Z = \emptyset$. This is obvious, because in this case the examined edge simply has no endpoint and there is no box that could be added.

- isSufficient(*subfaceCov*). This condition expresses our aim to come up with possibly the smallest oriented cubical set for which sufficient sign covering can be computed. So as soon as some wall can be covered sufficiently, we simply do not extend the oriented cubical set with the corresponding endpoint.

The outer loop written directly in frHasRoot always adds all the boxes from the recent *cubSet* into the set *visited*, so that no other box from *cubSet* is used to initiate the joining by the join function, because that would lead to pointlessly recomputing the same oriented cubical set.

3.10.5 Note on the correctness of the joining process

Here we want to address an important note on the joining process as we introduced in the previous paragraphs and expressed it in Algorithm 12.

Sometimes during the joining, the following can happen. Suppose that X'_1 is currently examined box in the constructed oriented cubical set Ω, that has a neighbour Z which is not yet in Ω. Suppose further that a sufficient sign covering of the wall d in the edge (d, Z) of X'_1 is actually found. This causes Z not to be added to Ω in this step and the covering of d to be added to the continually constructed resulting sign covering *sc*.

However, box X'_1 might cause some other box X'_2 to be added to Ω, and X'_2 may cause box X'_3 to be joined and so on, until come across a box X'_k, which causes Z to be added to Ω.

Now, as a part of the processing of Z, an attempt to find a sufficient sing covering of the opposite edge to (d, Z) will be performed. This of course succeeds, because the walls of (d, Z) and opposite$((d, Z))$ are the same boxes, only with different orientation. This covering is then added to *sc*.

Now we ended up with a sign covering *sc*, that is actually not a covering of any oriented boundary of Ω, because it contains a pair of extra boxes with opposite orientation. This scenario is shown in Figure 3.14.

So if we perform the joining process as we devised it, then each time we obtain some oriented cubical set Ω, the associated sign covering *sc* we continually build might actually not be of some oriented boundary $\partial\Omega$ of Ω, but instead of the set $\Psi = \partial\Omega \cup \Phi$, where Φ only contains some pairs of identical boxes differing only in orientation.

However, it follows from what we indicated in Section 2.3, and from what is thoroughly explained in [2, chapter 3], that it is ensured during the combinatorial phase of the topological degree computation, that deg$(\partial\Omega)$ = deg(Ψ).

We take this approach to the joining process, because it is easier to construct such set Ψ, than actual $\partial\Omega$, as the latter requires checking every sub-

Algorithm 12 frHasRoot for grid frames. (Contract in Table 3.2)

```
1: function frHasRoot(frame, ref)
2:     ▷ this specialization returns simply the unmodified input frame
3:     procfr ← an exact copy of frame
4:     visited ← an empty set of boxes
5:     F_P' ← frFunc(frame)
6:     for each X' ∈ frItems(frame) do
7:         if X' ∈ visited then continue
8:         (sc, cubSet) ← join(X', F_P', visited, ref)
9:         add all boxes from cubSet to visited
10:        if isSufficient(sc) and deg(sc) ≠ 0 then
11:            return (procfr, true)
12:        end if
13:    end for
14:    return (procfr, false)
15: end function
16: function join(X', F_P', visited, ref)
17:    ▷ attempts to find an or. cubical set and its suff. sign. cov. wrt. F_P'
18:    sc ← an empty sign covering, st ← an empty stack of boxes
19:    cubSet ← an empty set of boxes          ▷ the constructed or. cub. set
20:    push X' onto st
21:    while st is not empty do
22:        X_c ← pop the top box from st
23:        if X_c ∈ cubSet then continue
24:        insert X_c into cubSet
25:        for each (d, Z) ∈ edges(X_c) do
26:            subfaceCov ← signCovering({d}, F_P', ref)
27:            if Z = ∅ or isSufficient(subfaceCov) then
28:                append subfaceCov to sc
29:            else
30:                push Z onto st
31:            end if
32:        end for
33:    end while
34:    return (sc, cubSet).
35: end function
```

face, that is about to be added into $\partial\Omega$, whether there is already the same sub-face with opposite orientation (and if yes, remove both).

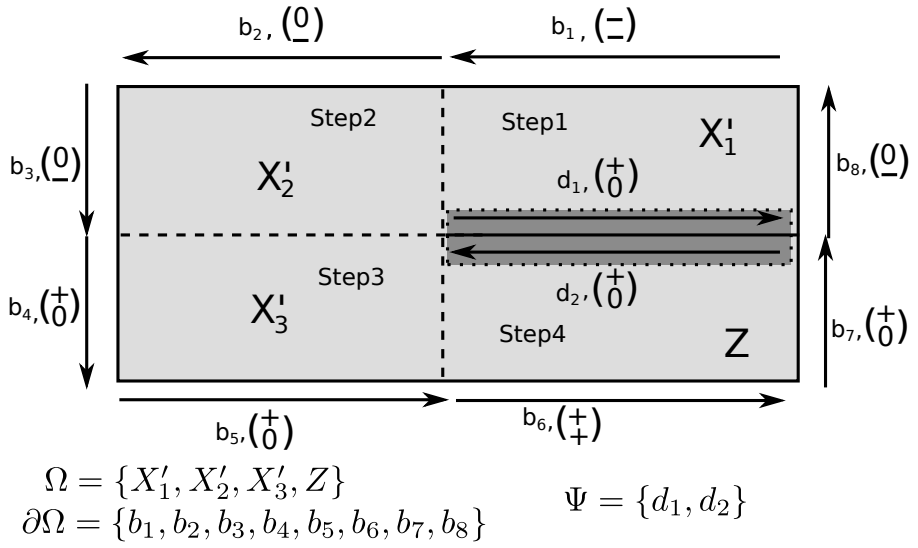


Figure 3.14: Sign covering constructed during the joining in the grid. The oriented cubical set Ω was created. Its construction was initiated with X'_1 (Step 1), which caused X'_2 to be added (Step 2) and that caused X'_3 to be added (Step 3) and this box finally caused Z to be added (Step 4). Because a suff. sign cov. for the wall between X'_1 and Z was actually found, the result of the joining is a sign covering of $\partial\Omega \cup \Psi$ instead of just $\partial\Omega$, where Ψ contains a pair of identical boxes with opposite orientation.

Chapter 4

Choosing tools for implementation

In this chapter, we discuss the main tools we used for implementing our application, that encapsulates the main algorithm from Chapter 3 as well as the algorithm for computing the topological degree [2] from Chapter 2.

4.1 Programming language used for the implementation

We choose to write our implementation in C++. One of the main reasons is, that the interval arithmetic library of our choice (see Section 4.2), Gaol [9], is written in this language. We make use of the C++ standard library and aim our implementation to comply with C++11 standard.

We further made use of Java Swing [15], to implement a GUI for our algorithm and we use Python for writing scripts related to testing and experimental measurements.

4.2 Interval arithmetic

We are looking for a suitable library implementing interval arithmetic, that our implementation can make use of. The requirements are, that it must provide all standard interval operations and functions, it should be open-source, reasonably up-to-date, well documented and tested. After considering several available libraries, briefly mentioned in the following subsections, we choose to use Gaol [9].

4.2.1 Boost interval arithmetic

This IA's implementation is a part of Boost free peer-reviewed C++ source libraries, available from [16], distributed under the Boost Software License [17].

It aims to provide a flexible framework for defining interval computations with both native floating-point and user defined data types and allows to specify custom rounding and checking policies[7] For intervals with native floating-point endpoints (`float` and `double`), inclusion property of algebraic functions is guaranteed on platforms listed in [18]. Rounding of transcendental functions depends on their implementation in the C++ standard library, which we consider limiting and non-robust for our purposes.

The advantage is, that this is a header-only library, and thus requires no installation. It just suffices to include particular header files into one's project and then it is ready to use.

4.2.2 Filib++

This is a library written in C++, that is actually an extension to a former library Filib, developed at the University of Karlsruhe. It is documented in [19]. It provides methods for working with intervals with either `float` or `double` precision and supports sound evaluations of interval inclusion functions for a comprehensive set of elementary functions. It offers multiple rounding modes to be set for computations.

4.2.3 IAMath

IAMath is an interval arithmetic implementation written entirely in Java. It does not rely on any support for a hardware rounding mode switching, and instead performs manual bit manipulations of Java's `double` type, to compute successors and predecessors of a given number, to ensure the inclusion property of interval computations. It is available at [20], and it is for example used in an application IASolver, described in detail in [21].

4.2.4 Gaol

Gaol is an IA library written in C++, available at [22], and documented in [9]. It is available under the GNU Lesser General Public License [9, pp.73–80].

Gaol is designed as a collection of classes and template methods that allow computations with intervals with `double` endpoints. It relies on an underlying mathematical library, either APMathlib or CRlibm, see [9, p. 4], that provide correctly rounded elementary functions, which may otherwise not be available directly by the standard library on the given platform. We consider this as an advantage.

[7] A checking policy is a setting, that defines how interval computations will deal with cases like empty intervals, unbounded intervals, NaNs etc.

Another advantage is, that it provides methods for constructing intervals by parsing them directly from string literals. For example, if we want to construct an interval constant $[0.1, 0.3]$, we can write

```
interval sample("[0.1, 0.3]");
```

instead of using a constructor, that simply takes two `double` values as respective endpoints, i. e.

```
interval sample(0.1, 0.3);
```

This is important, as the latter way of construction may not ensure the intended inclusion property, because the passed endpoints may not be exactly representable in the underlying binary floating-point format and they will be rounded at compile time, presumably using the default round to nearest mode. The parsing is actually not restricted just to simple literals, and can handle more complicated interval expressions, see [9, chap. 10]. It guarantees the inclusion property, as long as all number literals in the string do not exceed 15 digits.

Gaol allows to use the trust rounding mode (see Subsection 1.3.5 or [9, p. 10]), which we make use of in our implementation. It is initiated with the call `gaol::init()` and ended with `gaol::cleanup()`. Note, that for this to work, Gaol must be compiled with the `--preserve-rounding=no` flag, read more in [9, p. 5].

4.3 User interface

We provide a standard command line interface to be used to execute our implementation with different inputs. We expect inputs to be created in separate text files, that will be then used to feed the standard input.

Apart from that, we also make use of Java Swing technology [15] to provide a simple GUI, that collects the user input via graphical components and then sends it directly to the command line interface, processing its output afterwards. From the nature of our algorithm (Chapter 3), it makes sense to present its results (sets of boxes) visually, where applicable. To this end, we again decided to use Java Swing and its graphics capabilities, as well as a specialized software called VIBes, see the following subsection.

4.3.1 VIBes

VIBes, which stands for Visualizer for Intervals and BoxES, is a software allowing interactive visualization of intervals and boxes. It is available at [23] and described in [24]. The software consists of two parts.

- The VIBes server, which is the actual visualization program, that allows users to view or export existing figures and interact with them.

- Application programming interfaces for several languages (C, C++, MATLAB, Python), through which clients send drawing requests to the server. We use API for C++, which is provided via header files, that are simply included to one's project.

The server and client programs communicate through named pipe and all the messages being sent are in a human-readable format, which makes it convenient for debugging. Resulting figures can be easily exported.

4.4 Unit testing

We want to incorporate unit testing to our development process to verify, that our classes and utility functions comply to their public interfaces. To this end, we choose a C++ framework named Catch, which we already have some experience with.

4.4.1 Catch

Catch is an automated test framework for C++ and Objective-C, available from [25], distributed under the Boost Software License 1.0, see [17]. It requires no installation, as it is written completely in a series of header files, that can be directly included in one's projects. Apart from the standard library, it has no other external dependencies.

Catch implements test cases environment, assertions and logging via macros. A test case can be further divided into sections, which are nested into it, and are executed independently one from another. This behavior effectively substitutes classic test fixtures and allows writing simple tests with less code. Basic support for matchers is also provided. For full feature overview, see [26].

Chapter 5

Application design and implementation

This chapter describes relevant design and implementation decisions we made when implementing our final application, that encapsulates the solving algorithm we developed in Chapter 3. Its main and stand-alone part is written in C++ language with the use of the standard library and Gaol interval arithmetic library [9]. We also supply a GUI written using Java Swing [15] which serves as an adapter to the C++ application. (How to use the GUI from a user perspective is described in Appendix A.)

Our design is mostly object-oriented, containing separate classes for each significant entity type. We tried to keep the class hierarchy shallow and broad, rather than deep and thin. The primary intention is to encapsulate common code for reuse and to define public interfaces convenient for accessing and unit testing each entity.

In some cases, where polymorphic behavior was needed, we make use of class inheritance—for example when using the visitor pattern [27, pp.634–637] to process ASTs of parsed function expressions. We also aimed to make every non-leaf class abstract, as suggested in [28, item 35].

In the following, we mostly focus on describing the most significant classes we designed, map their functionality to the concepts from previous chapters and discuss relevant implementation details.

The details about compilation and running the application are summarized in Appendix A.

5.1 Intervals and boxes

Intervals and boxes are the basic entities and every complex functionality of our implementation makes extensive use of them. We represent and manipulate intervals via class `Component` and boxes via class `Box` which we cover

separately. We also mention representation of bisectors as they are closely related to boxes.

5.1.1 Class Component

This class represents a closed interval $[a, b]$ with floating point endpoints and provides an interface for performing sound interval arithmetic computations.

More technically, an instance of `Component` encapsulates an instance of `gaol::interval` [9] and defines binary operators and methods, which serve as adapters to the corresponding interval arithmetics operations provided by Gaol. Apart from that, various utility methods are included (like getting the width of the interval, bisecting it etc.).

We introduced this extra level of indirection, so that only this one class (apart from a few unit tests) needs to know about the concrete underlying interval arithmetic library. This makes it easy to make a change, if we would decide to use a different IA library in the future.

`Component` is designed and implemented as a mutable class, which is a decision we made to achieve a better efficiency of resources usage, as manipulations with its instances are the very base of every significant action or step our solving algorithm performs.

It is important to mention, that the class contains the following static methods.

- `static void Component::init()`.
- `static void Component::cleanup()`.

The first one is meant to be called prior to any manipulation with its instances to execute an initialization code which ensures retaining the inclusion property in subsequent interval computations. This is simply an adapter to the call `gaol::init()` to initialize the trust rounding mode (Subsection 1.3.5). The latter one is meant to execute a cleanup code needed to be able to perform any subsequent floating point computation in the standard round to nearest mode. Similarly to the first method, this an adapter for `gaol::cleanup()` (see [9, p. 10]).

5.1.2 Class Box

This class represents an oriented box in $\mathbb{R}^n$ of a given dimension. For efficiency purposes, this class is designed to be mutable, allowing callers to change its components, which is especially useful during a bisection, because the bisected box can itself become one of the resulting child boxes and only one extra instance needs to be created (to store the other child), instead of two. Important instance methods are

- `void getComp(int i) const`; `void setComp(int i)`;

  Gets and sets, respectively, the i-th component of the box.

- `void getTopologicalDimension() const`;

  Returns the dimension of the box.

- `bool containsZero() const`;

  Returns true, if and only if the point 0 is contained in the box.

- `SignVector computeSignVector() const`;

  Computes a sign vector (as an instance of class `SignVector`) of n elements, whose i-th element is the sign of the i-th component of the box, or 0, if the component contains zero.

- `std::list<Box> computeInducedBoundary() const`;

  Creates and returns a list containing all oriented faces of the box.

- `Box bisectBy(const IBisector &bis)`;

  Attempts to bisect this box using the given `IBisector` instance, which encapsulates both logic and data to select the dimension and ratio of the bisection. The calling box is mutated into one of the child boxes, while the other child becomes the return value.

5.1.3 Class IBisector and derived classes

Class `IBisector` is a base class providing a common interface related to box bisections. We currently offer one specialization, class `Bisector`, which implements a bisection of a given box along its longest component with a given ratio. The bisection ratio defaults to 0.5 and its value is passed as a parameter during the instantiation.

Another important parameter passed during the instantiation is the bisection threshold, which maps to the parameter d in Algorithm 4 and it is therefore used to specify the lowest width of the box that will be bisected.

The `Bisector` instances do not actually perform the bisections themselves. What they do is they examine the passed box and return instances of `BisectionResult` containing information about which component of the given box and at which ratio should be bisected. This information is then used by `Box` instances to actually bisect themselves (via their method `bisectBy`).

Callers access the functionality of bisectors via the following virtual instance method.

- `virtual BisectionResult bisect(const Box &box) const`;

Examines the given box, whether it can be bisected according to the underlying rules of the bisector. In the case of the `Bisector` specialization, this locates the longest component of the box and checks, if it is wider than the underlying threshold. It returns the information about how (and if) the bisection should be performed as an instance of `BisectionResult`.

5.2 Sign vectors and sign coverings

Here we cover the representation of sign vectors and sign covering, which is rather simple. The actual computation of sign coverings via our Algorithm 1 was made part of the `IFunction` interface, see 5.4.2.

5.2.1 Class SignVector

Instances of this class encapsulate sign vectors. Like `Box`, class `SignVector` is mutable. They are internally represented as `std::vector<Sign>`, where `Sign` is an enumeration type containing three values `plus`, `minus`, `zero`.

The sign vector elements are accessible by the subscript operator `[]`, which acts as an adapter to the subscript operator of the underlying vector.

Very often, we make use of the instance method for determining, if the sign vector contains at least one non-zero entry.

- `bool isSufficient() const;`

  Returns true, if the instance contains at least one non-zero entry, and false otherwise.

The combinatorial phase of the topological degree computation also makes use of the ability to remove an entry from the vector at the given position, reducing its size.

- `void removeAt(int index);`

  Removes the sign at the given position from this instance.

5.2.2 Structures used to express sign coverings

To express sign coverings, our code introduces and uses two typedefs.

- `typedef std::pair<Box, SignVector> BoxSignVectorPair;`

  Pair of a box and a sign vector used to represent a pair in a sign covering.

- `typedef std::list<BoxSignVectorPair> SignCoveringList;`

  List of `BoxSignVectorPair` instances used to represent sign coverings.

To determine if a given sign covering represented by a `SignCoveringList` instance is sufficient, the following function (which is defined outside of any class, in the namespace called `boxes`) is used.

- `bool isSignCoveringListDetermined(const SignCoveringList &sl) const;`

  Returns true, if the given sign covering is determined and false otherwise. This is an implementation of Algorithm 2.

5.3 Combinatorial phase of the topological degree computation

This section covers classes that implement the function DEG from Section 2.3 and provides a public interface to perform the combinatorial phase of the topological degree computation.

5.3.1 Class ISignSelectionStrategy and derived classes

This class represents the strategy for selecting the index and the sign during the combinatorial phase of the topological degree computation. It has two specializations.

- `class DefaultSelectionStrategy`; Implements a default strategy of simply selecting the first index and the sign +.
- `class LeastFrequentSelectionStrategy`; Implements the least frequent selection strategy (Subsection 2.3.4).

We almost exclusively use the least frequent strategy selection and we introduced the default one mostly for debugging purposes.

5.3.2 Class IDegree and derived classes

These classes encapsulate our implementation of the combinatorial phase of the algorithm for computing the topological degree, that is the recursive method DEG (Section 2.3). Class `IDegree` is a base class with common interface. It has two specializations. Class `Degree` contains an actual implementation and class `ParallelDegree` is its enhancement with the ability to process the sets of selected faces L_{sel} in parallel using the given number of worker threads (see further).

Instances of these classes are immutable and once its they are constructed, they can be used repeatedly for computations via the instance method `deg`, which forms their public interface.

- `int deg(const SignCoveringList &cov, const ISignSelectionStrategy &st) const;`

  Executes the combinatorial phase of the degree computation using the given sign and index selection strategy. This is an implementation of Algorithm 3 and uses the same contract as presented with the pseudo-code.

The parallelization ability offered by `ParallelDegree` is straightforward. Before the set L_{sel} is processed (Step 3 in Algorithm 3), it is evenly partitioned into k subsets, where k is the number of worker threads. Every thread then works separately and writes its results into its own reserved list. As soon as every worker thread finished, these lists are appended together by the main thread using `std::splice` method from the standard library.

To operate with threads, we use the POSIX thread library pthreads [29].

5.4 Representation of interval inclusion functions

Our implementation allows to parse and store definitions of mathematical functions formed by arithmetic expressions containing constants, variables, standard binary operators and elementary functions. The stored functions can be repeatedly evaluated (interpreted) with different inputs and they are treated as interval inclusion functions, meaning that the evaluation is sound wrt. rounding errors.

Here we cover the classes responsible for creating and storing the internal representation. In Section 5.5, we talk more about the way we implement their interpretation.

5.4.1 Class Input, Scanner and Parser

These three classes are used to scan the input function definitions and parse it into an intermediate representation in the form of an AST, which can then be further adjusted by the particular interpreter chosen to perform the evaluations. (Section 5.5).

The caller does not need to know the details of these classes, because their functionality can be used transparently from the `IFunction` class covered in Subsection 5.4.2. Let us however discuss the form the function definition must have.

The function definition starts with the enumeration of its formal arguments (mathematical variables) in square brackets. Their total number determines the input dimension of the function.

This is then followed by a comma separated list, again enclosed in square brackets, of arithmetics expressions which may involve the formal arguments

specified, constants, binary operators and elementary functions and their compositions. It can even contain interval expressions. The size of this list determines the output dimension of the function.

For example, the result of parsing the string

- `[x, y, z][x+1, sin(z) + [y, y+1]`

would be the internal representation of the interval function F defined as

$$F(x, y, z) = (x + 1, \sin(z) + [y, y + 1])$$

Using Extended Backus–Naur form [30], we can express the underlying grammar used for parsing as follows (leaving out the productions for nonterminals like numbers, identifiers etc. to keep it short).

```
S ::= ARGS , RET_STATEMENT
ARGS ::= '[' , [ IDENT , [ SEPARATOR ] ] , ']'
RET_STATEMENT ::= '[]' |
     ( '[' [EXPRESSION , SEPARATOR] , EXPRESSION ']' )
EXPRESSION ::= E1 , { (+ | -) , E1}
E1 ::= E2 , { (* | /) , E2 }
E2 ::= E3 , { ^ , E3 }
E3 ::= ( (+ | -) , E3 ) | E4
E4 ::= NUMBER | INTERVAL | IDENT | UNARYFUNCCALL |
     NULLARYFUNC | ( '(' , EXPRESSION ')' )
INTERVAL ::= '[' , EXPRESSION , SEPARATOR , EXPRESSION , ']'
UNARYFUNCALL ::= UNARYFUNC , '(' , EXPRESSION , ')'
SEPARATOR ::= ';' | ','
```

The nonterminal `UNARYFUNC` produces strings matching particular elementary functions, like `sin` or `arctg`. Let us note, that to express a square root, one writes `sqrt` and to express an n-th root, one writes `sqrtn`, where the `n` is the actual number, like `sqrt4`.

The nonterminal `NULLARYFUNC` produces strings matching mathematical constants, `pi` for π and `e` for e.

5.4.2 Class IFunction and derived classes

These classes provide the algorithmic representations of interval inclusion functions obtained by parsing and processing given function expressions. They allow to repeatedly evaluate the stored functions with different given inputs. `IFunction` is the base class and its specializations differ in the underlying interpreter they use to perform evaluations.

The derived classes are designed to be used polymorphically via references and pointers to the base class. We make use of the typedef `UPtrIFunction` which is a unique pointer (`std::unique_ptr`) to `IFunction`. Instances of these classed are mean to be created via the factory methods from `IFunction` class.

- `UPtrIFunction parseRecursive(std::string s) const;`

  Parses the given function expression and returns a unique pointer to an instance of `IFunction`, allowing it to be repeatedly interpreted with the recursive interpreter (Subsection 5.5.1).

- `UPtrIFunction parseLinear(std::string s) const;`

  Parses the given function expression and returns a unique pointer to an instance of `IFunction`, allowing it to be repeatedly interpreted with the linear interpreter (Subsection 5.5.2).

- `UPtrIFunction parseLinearCached(std::string s) const;`

  Parses the given function expression and returns a unique pointer to an instance of `IFunction`, allowing it to be repeatedly interpreted with the linear cached interpreter (Subsection 5.5.3).

Overloads accepting `std::istream` are also supplied.

The actual scanning and parsing processes are performed via our by hand written scanner and parser, that are encapsulated in `Scanner` and `Parser` classes, respectively. See Subsection 5.4.1.

The actual evaluation of the stored function expression is done via the function `eval`. This function accepts an instance of `Box` that must have the same number of components as the input dimension of stored function and evaluates it using the underlying interpreter, returning the result as another instance of `Box`.

When modeling (mathematical) parametrized functions, we expect the caller to treat the first m formal arguments of the stored algorithmic implementation as variables and the remaining k of them as parameters. If the caller then maintains the m-box representing the variable domain and the k-box representing the parameter domain separately, she can make use of another overload of `eval` that accepts two instances of `Box` instead of one. It is a convenient method which behaves as if the passed boxes $B_1 = I_1 \times \cdots \times I_m$ and $B_2 = J_1 \times \cdots \times J_k$ were actually forming a single box

$$B = I_1 \times \cdots \times I_m \times J_1 \times \cdots \times J_k.$$

Other than that, `IFunction` also contains virtual methods implementing the actual sign covering computations (Algorithm 1).

- `SignCoveringList computeSignCovering(Box box, ...);`

  Implements a call to signCovering (Algorithm 1), passing in just the one given box `box`.

- `SignCoveringList computeSignCovering(`
  `std::list<Box> boxList,...);`

Implements a call to SIGNCOVERING (Algorithm 1), passing in an arbitrary list of boxes.

- `SignCoveringList computeBoundarySignCovering(Box box,...);`

  Starts by computing a list of oriented faces of `box`, and then passes it to the particular overload of `computeSignCovering`.

These methods make use of `eval` and the actual bisections performed during the computation are dictated via `IBisector` instance (omitted for clarity in the previous list).

5.5 Interpretation of interval inclusion functions

To perform repeated evaluating of the stored algorithmic representations of interval inclusion functions with different inputs, we decided to write three custom interpreters. Each of them is encapsulated in a particular `IFunction` specialization (Subsection 5.4.2). The interpreter dictates both the way the parsed function expression is stored, as well as the actual way of interpretation.

The parsed function expression is always passed to the interpreter in a form of a standard AST, like the one in Figure 5.1. Each node of the AST is an instance of some specialized class derived from base class `Node`, representing a particular element in the function expression like constant, variable, binary operator etc.

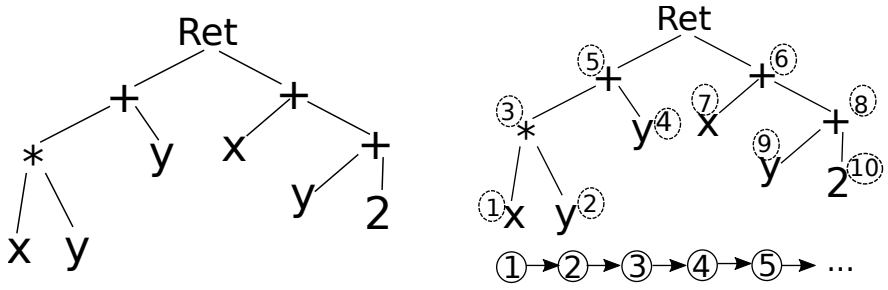


Figure 5.1: AST for the function $f = (xy + y, x + y + 2)$ and its linearization by post-order traversal.

It this then up to the particular interpreter to preprocess this AST so it will suit its needs. We implemented three different interpreters, whose efficiency in terms of execution speed we compared in Section 6.4.

To construct an instance of `IFunction` associated with a certain type of the interpreter, the factory methods described in 5.4.2 can be used.

5.5.1 Recursive interpreter

This interpreter simply evaluates the function's AST by recursively traversing it in post-order. Double dispatching via the visitor pattern is used to this end (see [27, pp.634–637]).

5.5.2 Linear interpreter

During its construction, this interpreter takes the input AST, performs a one-pass post-order traversal on it, creating a linear sequence of its nodes (Figure 5.1), which it then internally stores as a list. Evaluation of the function with certain inputs is then done by simply iterating this list. The information about child-parent relationship from the tree is still used, but no more recursive double dispatching is required. Any intermediate values are stored directly in the corresponding nodes.

5.5.3 Linear cached interpreter

This implementation enhances the linear interpreter, by exposing the nature of our problem domain. During the solving algorithm computation, it is expected, that two consecutive function evaluations will be often done with inputs that will not be completely different and unrelated, but will instead match in some components. This is in general happening in two scenarios.

- When performing the parameter space decomposition in Algorithm 4, it is expected, that for a given parameter sub-box $P' \subseteq P$, several evaluations of the function, with possibly different variable sub-boxes $X' \subseteq X$, are performed. These are all the evaluations that happens for a currently processed frame during the processing of its items collection.

- When computing a sign covering using Algorithm 1, the parameter sub-box P' is again the same during the whole process, so at least the parameter sub-box components will always match.

Therefore, this interpreter always remembers its last input, and keeps all intermediate values of nodes from the last evaluation stored. When a new evaluation is requested, it first finds out, whether any components of the current and previous input match. If they do, this fact is noted, and subtrees of the function's AST, that make use only of these matched components, are not re-evaluated.

Every node is associated with a bit-string, providing information about which components (variables) values are required to successfully compute this node. The leaves get either $00\ldots0$, if they represent constants, or a bit-strings with exactly one 1, if they represent a variable. The tree root gets a bit-string $11\ldots1$. See Figure 5.2 for a simple example.

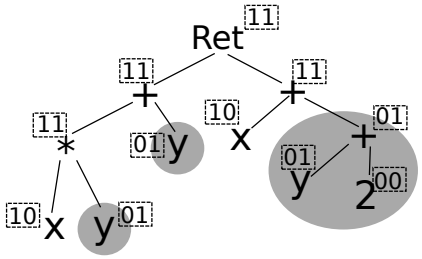


Figure 5.2: AST with the bit-string information for $f = (xy + y, x + y + 2)$. When two consecutive inputs have the same second component (variable y), then the colored sub-trees do not need to be reevaluated the second time, as the cached intermediate results can be directly used.

5.6 Frames and the parameter space decomposition

This section presents the classes related to the implementation of Algorithm 4 for parameter space decomposition and to the frame data structure its specializations that parametrize its functionality and were introduced in Chapter 3.

5.6.1 ISolver and derived classes

The `Solver` class is an abstract class encapsulating and implementing the parameter space decomposition in Algorithm 4). It has five derived classes (adapters), each of which is used to initiate the solving process with one concrete frame data structure design from Chapter 3.

These classes are stateless and once their instances are constructed, they can be used for the computation repeatedly, via the instance method `solve`. This method is the entry point of the solving process and has the following parameters.

- `const Box &vars`; Input variable domain box X in Algorithm 4.
- `const Box &params`; Input parameter domain box P in Algorithm 4.
- `const IBisector &bisector`; Instance of `IBisector` that encapsulates the parameter d (box width threshold) in Algorithm 4.
- `double faceRefModifier`; Face refinement modifier E in Algorithm 4.
- `int capacity`; Capacity of the items collection of frames. Every frame ever created during the computation will have the same capacity specified by the passed argument.

- `IFunction &function`; Instance of `IFunction` which represents the input function F_P in Algorithm 4 as well as the underlying interpreter used for its evaluations.
- `const IDegree &dc`; Instance of `IDegree` encapsulating the combinatorial part of the degree computation. For example, it stores the sign and index selection strategy to be used.
- `Boxlist &yesList`; List of boxes which is used to store the contents of the resulting list Y. It must be passed in empty.
- `Boxlist &noList`; List of boxes which is used to store the contents of the resulting list N. It must be passed in empty.

5.6.2 IFrame and derived classes

These are implementations of the different frame data structures designs introduced in Chapter 3, with `IFrame` representing their common interface.

- `class StaticFrame`; Static frame implementation from Section 3.6.
- `class BisectOnlyFrame`; Bisect-only frame implementation from Section 3.7.
- `class BisectAndKeepFrame`; Bisect-and-keep frame implementation from Section 3.8.
- `class TreeFrame`; Tree frame implementation from Section 3.9.
- `class GridFrame`; Grid frame implementation from Section 3.10.

5.6.3 Core structure

The core structure of frames is encapsulated in the base class `IFrame`. Instances of this class stores a private member of class `Box` representing the associated parameter sub-box F. Another private member is a reference to `IFunction` instance representing the underlying associated interval inclusion function $F_{P'}$.

For managing the ownership of frames within `ISolver` instances, we use typedef `UPtrIFrame`, which is a unique pointer (`std::unique_ptr`) to `IFrame`.

The base class contains the implementation of the frBisect function (recall the frame manipulating functions from Table 3.2). This function is used by `ISolver` instances when performing the parameter space decomposition.

- `UPtrIFrame bisect(const IBisector &bisector);`

  Bisects the underlying parameter sub-box P' into P_1 and P_2. This instance will be modified to now hold P_1, and then cloned to another

instance holding P_2, which will be returned via an (`UPtrIFrame`) instance.

The implementation of the other significant frame manipulating functions is left to the specialized classes. For efficiency purposes, all the derived clases are designed as mutable classes and the frame manipulating functions are implemented as private methods changing the actual state of the underlying frame.

The chain of the calls to frPrune, frHasRoot and frRefine frame manipulating functions is accessible by the public virtual method `hasSolution`, which returns and enumeration `Solution`, that can have the values `no, yes, unknown`.

- `virtual Solution hasSolution(IBisector &bisector, const IDegree &dc);`

  Executes the chain of frPrune, frHasRoot and frRefine frame manipulating functions, as expressed in Algorithm 4. The passed `bisector` encapsulates the face refinement threshold for sign covering computations and `dc` encapsulates the combinatorial phase of the topological degree computation.

  The return value maps to the conditions in Algorithm 4 as follows. If `no` is returned, the the items collection of the instance has become empty. If `yes` is returned, then a non zero degree was found for some box in the items collection. Otherwise neither is true.

The actual structure of the items collection is another thing specific to the derived classes. We will discuss it briefly in the following subsection.

5.6.4 Structure of the items collection

Classes `StaticFrame`, `BisectOnlyFrame` and `BisectAndKeepFrame` simply use an instance of `std::list<Box>` to store boxes in their items collection.

The use of `std::list` allows for easy iteration over the items collection and cheap inserting of new boxes into the list.

The items collection of `TreeFrame` is implemented as a unidirectional tree of nodes, where parents contains pointers to their children. Every node corresponds to one box in the items collection and is represented by the inner class `Node`, which stores an instance of `Box` and `std::list` containing pointers to children of this node. To express the parent-child relationship and the ownership of boxes by nodes, we make use smart pointers, `std::unique_ptr` and `std::shared_ptr`.

`GridFrame` instances store their items collection as a `std::list` of instances of their inner class `Node` (the same name as for `TreeFrame`). Atop

of this list, the grid structure of neighbours is built. Each `Node` again corresponds to a single box in the items collection and therefore stores one `Box` instance.

To realize the concept of edges, we use `std::map` and `std::list` instances. Namely, each `Node` instance has associated instances of `EdgeMap` and `EdgeList` that are given as follows.

- `typedef std::shared_ptr<Box> SPtrBox;`
- `typedef std::map<Node*, SPtrBox> EdgeMap;`
- `typedef std::list<SPtrBox> EdgeList;`

`EdgeMap` represents edges whit actual endpoints. The first template argument is a pointer to the endpoint, while the second one is a pointer the wall over which the edge is realized (see Section 3.10).

`EdgeList` then represent edges without endpoints. The template argument is the wall along which there is no neighbour.

To support retrieving edges from a certain direction only (recall the functions loEdges and hiEdges from Section 3.10), every `Node` instance actually stores a separate instance of `EdgeMap` and `EdgeList` for every possible direction. When working with n-dimensional boxes, this makes the total of $2n$ instances of each. They are stored in an array and accessed via index, that we map to a particular direction.

This definitely could be done more efficiently, but we have focused more on the correctness and the concepts we created in Section 3.10 turned out to be implemented rather easily using this approach.

Chapter 6

Testing and experimental evaluation

This chapter presents the results of the experimental evaluation of our implementation of the main algorithm we designed in Chapter 3 together with all the specializations of the frame data structure presented there. Our custom implementation of the algorithm for computing the topological degree (Chapter 2) is evaluated as well. We also describe how we tested the correctness of our implementations.

6.1 Testing environment

All tests and experiments were carried out on a computer with Intel Core i7 5500U @ 2.40GHz (2 cores / 4 threads) with 8GB of RAM, running Ubuntu 16.04 LTS 64-bit.

Execution time was measured using the GNU *time* utility, combined with the program *chrt*, to run our measurements with higher priority than other processes. Typical command had the form of

```
sudo chrt -f 99 /usr/bin/time [flags] [command to measure]
```

Memory consumption was also measured with GNU time, as the maximum resident set size. When measuring execution time, we aimed to select difficult enough inputs, so that the results were not affected by the limited precision of the underlying timers. We also repeated every measurement, typically from 2 to 10 times.

6.1.1 Reproducibility of the experiments

To make our tests and measurements reproducible, we encapsulated them into Python scripts, located in the folder `./scripts`. These contain both the logic

for the given experiment, as well as the input data and all parameters, like path to executables, number of repeats etc. To run these, Python version 3.5 or above is required, because of the modules we are using in the scripts.

Tests, that need direct access to the implementation, like unit tests or our correctness test of the main algorithm, were compiled as the part of the main C++ project, and are stored in the file `tests/tests.o`.

6.2 Unit Testing

We have used Catch framework, discussed in Section 4.4, to write unit tests to ensure that our implementations of "building blocks" like boxes, bisectors or interpreters comply to their public interfaces.

6.2.1 Running the unit tests

Unit tests are compiled into the file `tests/tests.o`. To run the all tests, simply execute the file with no parameters. Use the flag `-h` to display help and lists of available Catch flags to run the tests with.

6.2.2 Tests overview

This is the overview of the unit tests. We divided their sources into separate files, containing related test cases. Each test file has its unique tag, that can be used to run `tests.o` with, to execute only tests having this flag, for example

```
./tests/tests.o "[box]"
```

runs the unit tests with the flag `[box]`. The complete list of source test is the following.

- filename `boxtest.cpp`; tag `[box]` This file contains tests for verifying the functionality of basic properties of our implementation of boxes and oriented boxes and related operations like computing faces, oriented boundary etc.

- filename `signvectortest.cpp`; tag `[signvec]` Tests the creation, manipulation and property setting/accessing of the sign vector implementation.

- filename `bisectortest.cpp`; tag `[bisector]` Tests the bisection related operations provided by `Bisector` and `Box` classes.

- filename `scannertest.cpp`; tag `[scanner]` Verifies, that inputs (expressions, function definitions) are scanned into correct sequences of tokens.

- filename `evaltest.cpp`; tag `[eval]` Tests the correctness of our implementation of our function interpreters against gaol numerical expression evaluation capability, [9, p. 43–45].

  We form strings representing scalar numerical expressions, covering all building block our functions can be build of (bin. operators, elementary functions, constants), and possibly containing variables, and let them evaluate using our implemented interpreters. Then we evaluate the same inputs directly via gaol, replacing particular variable values directly in the input string, and check, that the results represent the same set of numbers via `gaol::interval::set_eq`, see [9, p. 22].

- filename `signselectionstrategytest.cpp`; tag `[signstrategy]` Tests the implemented strategies for selecting the sign and index during the combinatorial phase of the topological degree computation. We implement the least frequent selection strategy mentioned in Subsection 2.3.4 and a default one, that always selects the sign + and the index 0. The default one, however, was used only during debugging.

- filename `signcoveringtest.cpp`; tag `[signcovering]` Verifies the contract of the implementation of Algorithm 1 for computing sign coverings, which is the core of performing the numerical phase of the topological degree computation. We execute test cases that check if the implementation correctly returns a sign covering of some permissible set and if the sign covering is wrt. to the given function and is either sufficient or not depending on the particular input.

6.3 Topological degree computation

Because we provided our custom implementation of algorithm [2] for computing the topological degree, for the reasons summarized in Section 2.5, we wanted to perform some tests regarding its correctness and execution speed.

6.3.1 Correctness

To test correctness of our implementation, we took the reference implementation [12], that accompanies [2], prepared a suite of inputs consisting of functions $f\colon \mathbb{R}^n \to \mathbb{R}^n$ and associated domains $B \subseteq \mathbb{R}^n$, computed $\deg(f, B, 0)$ using both implementations and compared the results.

This test together with the input suite is encapsulated in the script file `degree_test.py` for easy reproducibility. We took the sample inputs provided by [12] and added some other of our own, aiming to cover different types of functions f, like linear, polynomial, trigonometric, logarithmic etc.

We also kept in mind, that our implementation offers more parametrization than the reference one, so we introduced more test cases with the same f and

B, but different parameters for our implementation. Namely, we tested all of our three function interpreters and both sequential and parallel execution of the combinatorial phase of the algorithm.

In all cases, the results of both implementations were identical. You are free to rerun the test by executing `degree_test.py`, or just see its output located in the same folder as the script.

6.3.2 Execution time comparison

We compared the execution speed our implementation with the reference implementation [12] on certain examples, that were complex enough to take enough time on both implementations to surpass the possibly limited precision of timers provided by the GNU time utility.

Again, all the measurements are reproducible by running the same script `degree_test.py`, as for the previous subsection. Here, we present the following examples. As in [2], we considered the function $f = (f_1, \ldots, f_n)$ defined as

$$\begin{aligned} f_1 &= x_1^2 - x_2^2 - \cdots - x_n^2 \\ f_2 &= 2x_1x_2 \\ &\cdots \\ f_n &= 2x_1x_n, \end{aligned}$$

for various n and the input box $B = [-1, 1]^n$. This function has a root $x = 0$, whose degree is 0 for n odd and 2 for n even([2, p. 21]).

Then, we considered the function $g = (\arctan(f_1), \ldots, \arctan(f_n))$, again for various n and the same input box B. The root $x = 0$ of this function has the same properties as the one of f.

We measured the time both implementation take to compute the topological degrees of f and g on domain B. Our implementation was run with recursive interpretation mode, with the combinatorial part of the algorithm executed sequentially. The results are presented in Table 6.1.

Our implementation turned out to be several times faster, especially when more function evaluating is involved (compare the results from Table 6.1 for functions f, g). As n gets higher and the combinatorial part therefore starts being the major part of the computation, the difference tends to lessen, but the ratio between the execution times is still significant.

6.3.3 Memory consumption

We also compared the memory consumption between ours and the reference implementation [12], in terms of the maximum resident set size, measured again by the GNU time utility. The general observation was, that for simple

Func.	n	*our* (s)	*ref* (s)	*ref*/*our*
f	8	0.220	2.43	11.03
	9	1.65	16.9	10.28
	10	13.0	102	7.835
g	8	0.223	4.20	18.82
	9	1.61	22.5	13.92
	10	13.1	119	9.057

Table 6.1: Execution time of the topological degree computation via our custom implementation (*our*) and the reference one [12] (*ref*).

Func.	n	*our* (MB)	*ref* (MB)	*ref*/*our*
f	8	8.14	16.7	2.05
	9	14.2	43.5	3.06
	10	27.9	109	3.90
g	8	8.23	16.7	2.03
	9	14.4	42.6	2.96
	10	28.0	109	3.89

Table 6.2: Max. resident set size during the topological degree computation via our implementation (*our*) and the reference implementation [12] (*ref*).

inputs, for which our implementation terminates almost immediately[8], our implementation tended to have its maximum resident set size around 4MB, which was around 2–4 more, than the reference implementation required for the same inputs.

For more complex inputs, where the computed sign coverings start getting larger and the combinatorial part takes more time to finish, both implementations' memory requirements naturally increase. However, in the experiments we performed, we found out, that the maximum resident set size grew slower for our implementation than for the reference one. If the input was complex enough, our implementation ended up requiring less memory than the reference one did.

We attribute this behavior to the way we manipulate lists in our implementation during the combinatorial phase. Mainly, we use several mechanisms provided by C++ to avoid unnecessary copying, like passing by reference or pointer wherever possible and splicing lists via `std::list::splice`.

In Table 6.2, we present measured results regarding functions f and g from Subsection 6.3.2, again for the input box $B = [-1, 1]^n$ and various dimensions n. As for previous subsections, the experiment can be reproduced by running `degree_test.py` script.

[8]meaning, that the running time was so small, that the timer provided by GNU time was unable to measure the exact time and returned 0.

6.4 Interpretation strategies

Among our unit tests (Subsection 6.2.2), we covered the verification of correctness of all our three function interpreters. We are further interested in comparing their performance in terms of computation speed. Although we perform all of our other experiments with each of our interpreters, so we have plenty of examples to compare, we want to also specifically choose some more difficult inputs, that put our interpreters under higher load, but that are otherwise not suitable for the rest of our experiments.

To this end, we browsed the demonstration database for PHCpack [31], containing systems of polynomial equations and their approximated solutions computed by PHCpack solver [32]. We took systems $f\colon \mathbb{R}^n \to \mathbb{R}^n$ presented there and computed sufficient sign coverings *sc* of some ∂B wrt. f. We tailored the input boxes B using the information about roots of f, that is part of the database, so that *sc* contained enough boxes for a single computation to take tens of seconds on average.

6.4.1 Examples

The script `script/interpreters.py` can be run to repeat all of our measurements regarding this section. Here we present the result of three selected experiments.

One of the function we picked was a function f titled "A neural network modeled by an adaptive Lotka-Volterra system, $n = 5$" (referred to as *noon5* in [31]). It is a function $\mathbb{R}^5 \to \mathbb{R}^5$ with components defined as follows,

$$\begin{aligned}
f_1 &= x_1x_2^2 + x_1x_3^2 + x_1x_4^2 + x_1x_5^2 - 1.1x_1 + 1\\
f_2 &= x_2x_1^2 + x_2x_3^2 + x_2x_4^2 + x_2x_5^2 - 1.1x_2 + 1\\
f_3 &= x_2x_1^2 + x_2x_3^2 + x_2x_4^2 + x_2x_5^2 - 1.1x_2 + 1\\
f_4 &= x_4x_1^2 + x_4x_2^2 + x_4x_3^2 + x_4x_5^2 - 1.1x_4 + 1\\
f_5 &= x_5x_1^2 + x_5x_2^2 + x_5x_3^2 + x_5x_4^2 - 1.1x_5 + 1.
\end{aligned}$$

For the input box $B_f = [-0.8, 0]^5$, the sufficient sign we covering *sc* we computed contained the total of 5 671 265 boxes.

Other function we chose was a function g known as "The 5-dimensional system of Reimer" (referred to as *reimer5* in [31], originating in [33]). This is a function $\mathbb{R}^5 \to \mathbb{R}^5$ having the following components,

$$\begin{aligned}
g_1 &= -1 + 2x^2 - 2y^2 + 2z^2 - 2t^2 + 2u^2\\
g_2 &= -1 + 2x^3 - 2y^3 + 2z^3 - 2t^3 + 2u^3\\
g_3 &= -1 + 2x^4 - 2y^4 + 2z^4 - 2t^4 + 2u^4\\
g_4 &= -1 + 2x^5 - 2y^5 + 2z^5 - 2t^5 + 2u^5\\
g_5 &= -1 + 2x^6 - 2y^6 + 2z^6 - 2t^6 + 2u^6.
\end{aligned}$$

Func.	Boxes in *sc*.	Interpretation	Exec. time (s)
f	5 671 265	Recursive	19.31
		Linearized	14.75
		Lin.-cached	10.07
g	11 117 615	Recursive	42.52
		Linearized	31.65
		Lin.-cached	19.47
h	11 117 615	Recursive	72.42
		Linearized	60.98
		Lin.-cached	49.87

Table 6.3: The time required for computing a sufficient sign covering *sc* wrt. functions f, g, h from Subsection 6.4.1 using different interpretation strategies.

For the input box $B_g = [0.1, 1]^5$, the sufficient sign we covering *sc* we computed contained the total of 11 117 615 boxes.

Finally, we considered function $h\colon \mathbb{R}^5 \to \mathbb{R}^5$, whose components are defined as

$$h_i = \sqrt[3]{\arctan(g_i)}$$

Function h is intentionally defined this way, so that the computation difficulty increases, without introducing new AST nodes with variables. Because at least one variable changes its value during back-to-back evaluations, the lin. cached interpreter must evaluate both the square root and the arctan function. For the input box $B_h = [0.1, 1]^5$, the number of boxes in the computed sufficient sign covering is the same as in the case of function g with the input box B_g.

Table 6.3 summarizes the measured execution times of the computations of sufficient sign coverings wrt. the presented functions on their respective domains. When experimenting with systems from [31], linearized interpretation strategy tended to be somewhat around 20–25% faster than the recursive one, like computations with f and g shows. The lin.-cached strategy is more dependent on the form of the function. Typically, it gave better results the more higher powers occurred in the function's prescription.

The experiment with function h indicates, that when the computation difficulty starts shifting from traversing the function's AST to the actual computation the interval arithmetic library performs, the differences in performance between the interpretation strategies tend to diminish.

6.5 The main algorithm

This section describes the tests and experiments related to our main algorithm from Chapter 3, which consists of the implementation of Algorithm 4 and all the specializations of the frame data structure we designed and implemented.

6.5.1 Correctness

The first thing we wanted to test was if our implementation complies to the output specification in Section 3.2. Namely, we wanted to verify (to certain extent, at least), that no point $p \in P$, where P is the input parameter box, ends up being incorrectly assigned to Y or N, when it actually does not belong there.

To this end, we wrote a simple black-box repetition test (see [34, p. 87]), which runs our main algorithm implementation with such input parametrized equation systems, for which we know their solutions (because they can be computed simply by hand, for example).

For each such particular input system $f_P(X) = 0$, the algorithm implementation is run and the resulting lists Y and N are saved. Then within a given time limit, the test repeatedly and pseudo-randomly chooses a point from some box in the list Y, and using our knowledge about the correct solution, it verifies, that the chosen point was correctly assigned to this list. The same then goes for the list N. These steps are repeated with every frame data structure specialization.

We did not want to encapsulate this test into a python script, because that would require serialization and deserialization of the results, possibly causing a loss of the inclusion property provided by interval arithmetic, to which we would no longer have access from the python script.

That is why we included this test directly in our C++ project, and make it build as a part of the unit tests executable `tests.o` (simply to avoid introducing a new separate executable). The repetition test can be run simply as follows.

```
./tests.o -reptest [time in seconds]
```

The given time limit specifies, how long should the repetition steps be performed for each input system and each frame specialization.

Let us demonstrate the idea of this test on a single example.

Example 6.1. On a simple example, we show the workflow of our repetition test. Consider the parametrized equation system $f_P(X) = 0$, with variables x, y and parameters a, b defined as

$$\begin{aligned} ax + 3y - 1 &= 0 \\ bx + 2y - 3 &= 0. \end{aligned}$$

For $3a = 2b$, the system has no solution, and for $3a \neq 2b$, it has exactly one solution

$$(x, y) = \left(\frac{7}{3b - 2a}, \frac{b - 3a}{3b - 2a} \right). \tag{6.1}$$

Our test first chooses a frame data structure specialization and uses it to execute the Algorithm 4 for f and certain input boxes X, P (these boxes are

part of the particular test case) and stores the resulting lists Y, N. Then it starts the actual testing, whose inner form depends on the particular input equations system. For the considered system $f_P(X)$, this is as follows.

For the list Y: Different points $p = [a', b']$ from different boxes in Y are repeatedly selected.

First, it is checked, whether $3a' = 2b'$. If this holds, the test logs fail, because that means the system does not have a solution for p, and p was therefore incorrectly placed into Y.

Otherwise, it needs to be checked, if the solution (x, y) for p, given by (6.1), is a part of the input box X. If so, then p was placed into Y correctly and the test logs success. Otherwise, p was misplaced and the test logs fail.

We do this check soundly using interval arithmetic. Both components of (x, y) are computed from (6.1) using interval arithmetic as intervals I_x, I_y, guaranteed to contain exact values of the respective components. The test then logs fail, if and only if $C = I_x \times I_y$ is guaranteed to lie completely outside of X (that is $C \cap X = \emptyset$).

For the list N: Different points $p = [a', b']$ from different boxes in N are repeatedly selected and steps, that are sort of complementary to processing Y are taken. First, if $3a' = 2b'$ holds, the test logs success. Otherwise, (x, y) is computed for p again as a box C, and the test logs fail, if and only if C is guaranteed to lie completely inside of X, that is $C \subseteq X$ holds, or in other words the system is ensured to have a solution for p inside the domain X.

6.5.2 Inputs for the further measurements

In the following subsections, we consider implementations of all five different designs of frame data structure from Chapter 3 and present and discuss measured results related to the quality of the computed solution, execution time and memory consumption. All these experiments can be rerun using the script `solver.py`, which contains both the logic and input suite. Here, we selected some concrete inputs from the suite, which we further work with. In the following, x, y, z denote variables and a, b denote parameters.

- The system of two linear equations f_{lin} with two variables and two parameters from Example 6.1; that is

$$\begin{aligned} ax + 3y - 1 &= 0 \\ bx + 2y - 3 &= 0 \end{aligned}$$

  The input domain boxes are $X_{lin} = [0.75, 1]^2$ and $P_{lin} = [-3, -1] \times [1, 2]$.

- System of equations f_{trig} with two variables and two parameters, containing the trigonometric function sin and the inverse trigonometric function arccos, defined as follows.

$$\begin{aligned} \arccos(ax + b) - 0.5 &= 0 \\ \sin(ay + b) - 0.5 &= 0 \end{aligned}$$

  The input domain boxes are $X_{trig} = [-2, 2]^2$ and $P_{trig} = [-1, 1]^2$.

- System of equations f_{hyp2} with two variables and two parameters, given as follows.

$$\begin{aligned} a(x-1)^2 + by^2 - 1 &= 0 \\ bx^2 + a(y-1)^2 - 1 &= 0 \end{aligned}$$

  Each equation represents an ellipsis and every solution of this system is their intersection. The input domain boxes are $X_{hyp2} = [-2, 2]^2$ and $P_{hyp2} = [-2, 4]^2$.

- System of equations f_{hyp3} with three variables and two parameters, given as follows.

$$\begin{aligned} a(x-1)^2 + by^2 + z^2\sqrt{a^2+b^2} - 1 &= 0 \\ bx^2 + (y-1)^2\sqrt{a^2+b^2} + az^2 - 1 &= 0 \\ x^2\sqrt{a^2+b^2} + by^2 + a(z-1)^2 - 1 &= 0 \end{aligned}$$

  Each equation represents an ellipsoid. The input domain boxes are $X_{hyp3} = [-2, 2]^2$ and $P_{hyp3} = [-2, 2]^3$.

6.5.3 Measuring the quality of the solution

One of the main aspects we aim to measure and compare is the quality of the solution. We think that from the nature of our solved problem, it makes the best sense to measure the solution quality in terms of how much volume of the input parameter box P were covered by both of the resulting lists Y, N. Such approach allows us to compare two solutions to the same input and see, which one provides more quality.

For a solution (Y, N), we define

$$\text{pave}(Y) = \frac{\sum_{P' \in Y} \text{volume}(P')}{\text{volume}(P)}, \ \text{pave}(N) = \frac{\sum_{P' \in N} \text{volume}(P')}{\text{volume}(P)}, \tag{6.2}$$

and consider one solution (Y_1, N_1) to provide a more quality list Y for the given input than another solution (Y_2, N_2) for the same input, if $\text{pave}(Y_1) > \text{pave}(Y_2)$. Similarly for the N list. We also consider the size of lists Y, N (meaning the number of boxes in them). Out of two solutions providing more or less the same quality, we consider the one with fewer boxes better.

Parameter	Meaning
Frame type	Which frame design implementation to use.
Bisection threshold	Limits the depth of the parameter space decomposition (parameter d in Alg. 4).
Bisection ratio	The ratio in which parameter sub-boxes are bisected (parameter r in ALg. 4).
Face ref. modifier	Dictates the extent of refinement during sign covering computations (parameter E in Alg. 4).
Capacity	Capacity of the frame's items collection (see Subsec. 3.4.1)
Interpreter	The interpretation strategy to use.

Table 6.4: Relevant parameters for experiments involving our main algorithm.

6.5.4 Summary of relevant parameters

With respect to the designs from Chapter 3, our implementation allows certain parametrization. Table 6.4 summarizes those parameters, that are relevant to us in the following experiments.

6.5.5 Experiment comparing the different specialization of frames

We took each of the inputs presented in Subsection 6.5.2 and run the main algorithm with each specialization of the frame data structure. We collected data about the solution quality, execution time and memory usage (expressed as the maximum resident set size).

Other parameters were identical for each frame specialization. Bisection threshold was set to $d = 0.0125$, bisection ratio to $r = 0.5$, capacity to $c = 32$, face refinement modifier to $E = 1$, and recursive interpretation strategy was used (Table. 6.4). The result is in Table 6.5.

In general, the grid frame provides the most quality solution. Although its pave(Y) and pave(N) values can be close to those of bisect-and-keep and tree frames, grid frame design often provides a solution containing less boxes with bigger volume. The downside is the running time, which is usually the highest out of all frame types. However, we found out, that the grid frame specialization often requires less capacity of its items collection than the other frame designs to provide solution of more or less the same quality.

6.5.6 Experiment exploring the impact of limited capacity

Here, we focused on the tree and grid frame designs and the impact of the capacity of their items collection on the quality of their solutions and the execution time. Bisection threshold (Table. 6.4) was set to $d = 0.0125$, bisection

Input	Frame type	pave(Y) (%)	pave(N) (%)	count(Y)	count(N)	time (s)	mem (MB)
f_{lin}	Static	32.94	28.52	569	166	0.13	4.18
	Bisect-only	0.0	64.53	0	700	0.66	4.23
	Bisect-and-keep	32.94	64.60	569	709	0.32	4.28
	Tree	32.94	64.31	569	679	0.33	4.26
	Grid	32.94	64.75	569	605	0.64	4.60
f_{sqr}	Static	57.03	19.82	6660	1318	0.18	5.03
	Bisect-only	0.0	36.02	0	8836	7.97	5.01
	Bisect-and-keep	57.03	35.63	6660	9430	1.68	5.94
	Tree	57.03	35.0	6660	9434	2.47	5.85
	Grid	57.03	35.45	6600	9895	9.19	6.13
f_{trig}	Static	24.98	45.56	300	288	0.38	4.25
	Bisect-only	0.08	45.56	756	288	1.53	4.30
	Bisect-and-keep	52.60	45.56	1441	288	0.67	4.53
	Tree	52.60	45.56	1440	288	0.68	4.50
	Grid	53.27	45.56	724	288	1.03	4.88
f_{hyp2}	Static	26.71	13.07	2246	382	1.29	4.39
	Bisect-only	61.91	31.77	3391	2982	2.05	4.87
	Bisect-and-keep	67.17	31.70	1271	1820	1.03	4.43
	Tree	67.18	31.53	1215	2178	1.43	4.41
	Grid	67.29	31.95	813	1207	4.5	4.83
f_{hyp3}	Static	28.83	0.51	9301	133	31.2	5.18
	Bisect-only	87.22	5.01	10617	4760	24.8	5.72
	Bisect-and-keep	90.11	5.55	5683	3104	23.9	5.10
	Tree	90.30	6.28	5790	2561	21.02	5.16
	Grid	90.67	7.35	2514	2134	25.43	5.46

Table 6.5: Quality, time and max. resident set size, *mem*, for the main solving algorithm.

Input	Capacity	$\frac{\text{volume}(Y)}{\text{volume}(P)}$ (%)	$\frac{\text{volume}(N)}{\text{volume}(P)}$ (%)	Time (s)
f_{trig}	8	47.26	45.57	0.70
	32	52.60	45.56	0.68
	128	52.60	45.57	0.88
f_{hyp2}	8	66.42	30.54	0.91
	32	67.18	31.53	1.46
	128	67.21	31.71	4.38
f_{hyp3}	8	87.28	4.37	18.4
	32	90.30	6.28	21.0
	128	90.43	6.37	44.7

Table 6.6: The quality of the main solving algorithm for tree frames with various capacity.

Input	Capacity	$\frac{\text{volume}(Y)}{\text{volume}(P)}$ (%)	$\frac{\text{volume}(N)}{\text{volume}(P)}$ (%)	Time (s)
f_{trig}	8	53.26	45.56	0.15
	32	53.27	45.56	1.03
	128	53.27	45.56	9.86
f_{hyp2}	8	67.18	30.83	0.97
	32	67.29	31.95	4.50
	128	67.30	32.08	54.6
f_{hyp3}	8	89.48	4.55	14.2
	32	90.67	7.35	25.4
	128	90.79	7.81	233

Table 6.7: The quality of the main solving algorithm for grid frames with various capacity.

ratio to $r = 0.5$, face refinement modifier to $E = 1$ and recursive interpretation strategy was used.

The results are summarized in Tables 6.6 and 6.7. The grid frame specialization seems to scale worse with growing capacity, but in our experiments, it typically required lesser capacity to provide results comparable to other frame specializations with bigger capacity.

Conclusion

In the scope of this thesis, we developed and implemented an algorithm for characterizing the sets of points $p \in P$, for which there exists a real-valued solution x to the given parametric system of equations $f_P(x, p) = 0$, where $f_P \colon X \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function formed by arithmetic expressions and X is a box (product of closed real intervals).

The algorithm is designed to make use of floating point based interval arithmetics for computing sound estimates of function images and for ensuring robustness with respect to rounding errors caused by finite precision of floating point numbers. The top-level of the algorithm is a branch and bound based parameter space decomposition that sets the core workflow of the algorithm, but can be highly parametrized.

We observed, that the overall quality of our solving algorithm is determined by the way the domain X is manipulated. We therefore introduced a data structure called frame intended to encapsulate such manipulation behind a generic interface, and then designed several concrete specializations of this structure with a specific behavior. The most complex one, the grid frame specialization, maintains a grid of sub-boxes of X, that can be temporarily merged into various oriented cubical sets. When explaining the concepts we developed, we aimed to provide both informal description, often accompanied by visual aids, as well as more technical details.

Our algorithm further uses the solution existence test based on the solvability property of the topological degree. For the actual computation of the topological degree, we made use of the algorithm [2]. However, the application was not straightforward as we had to figure out certain details specific to our problem domain.

Most notably, our working with parametrized functions led us to enhance previous work related to practical computation of the topological degree by creating a custom sound method of determining, if for arbitrary boxes $X' \subseteq X$, $P' \subseteq P$ the degree $\deg(f_p, X', 0)$ exists and has the same value for every $p \in P'$. This consists of computing sign coverings that are wrt. the whole set of func-

tions $\{f_p; p \in P'\}$ and reusing the combinatorial phase of the algorithm [2].

For this and other reasons summarized in Section 2.5, we provided a custom implementation of the existing algorithm [2], that according to our experiments outperforms its reference implementation [12] in terms of execution time. Our implementation proved to be around ten times faster on average in our experiments. For more complex inputs, our implementation also uses less memory compared to the reference one. In addition, our implementation can execute the combinatorial phase of the algorithm in parallel using threads.

We provided the implementation of our solving algorithm as well as the algorithm for computing the topological degree in C++. We created a CLI for both and for our solving algorithm also a GUI written in Java Swing [15], capable of visualizing the results. We used Gaol [9] as the underlying interval arithmetic library and Catch framework [25] for writing unit tests. On the enclosed DVD, an Oracle VM VirtualBox [35] image with our precompiled ready-to-use application is available.

One of the implementation highlights is that using the underlying interval arithmetic library, we wrote our custom interval inclusion functions interpreters. The most complex one is specifically tailored to our problem domain, and makes use of the fact, that inputs for consecutive evaluations of the same function often share some components, which allows to bypass evaluation of some subtrees in the expression's AST and use cached values instead. In addition, it is not recursive, but instead linearizes the function's AST during its initialization and then performs any evaluation using a simple linear code. In our experiments, this strategy proved to be faster than evaluating without caching.

We performed several tests and experiments related to the correctness, solution quality and time/memory requirements of our implementation. We encapsulated them into Python scripts so that they can be reused anytime.

As a part of our experiments, the quality and resource requirements of all frame data structure specialization were measured. The grid frame specialization provided particularly good results in terms of quality, but it also often took the longest time, because of all the manipulation within the grid structure. In general, our implementation is capable of producing quality results and terminate in reasonable time for various non-linear systems up to three equations, possibly containing combination of elementary functions like exp, log, sin, cos, arcsin, arccos, n-th root, ... In higher dimensions, the execution time starts to be a limiting factor, if we aim to preserve the solution quality.

As one possibility of future work, we suggest increasing the efficiency of the implementation of the grid frames. We aimed mainly for correctness and tended to use higher abstractions (mostly in form of structures provided by the C++ standard library), to express our ideas and intentions more clearly. The downside to that is the overhead this brings. Implementing some sort of copy on write mechanism, when passing the frames to child nodes in the parameter space decomposition process could also be beneficial.

Bibliography

[1] O'Regan, D.; Cho, Y. J.; et al. *Topological Degree Theory and Applications*. Boca Raton, FL: Chapman & Hall/CRC, 2006, ISBN 978-1584886488.

[2] Franek, P.; Ratschan, S. Effective Topological Degree Computation Based on Interval Arithmetic. *CoRR*, volume abs/1207.6331, 2012. Available from: `http://arxiv.org/abs/1207.6331`

[3] Muller, J. M. *Handbook of Floating-Point Arithmetic*. Boston: Birkhäuser, c2010, ISBN 978-0-8176-4705-6.

[4] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, Aug 2008: pp. 1–70, doi:10.1109/IEEESTD.2008.4610935.

[5] von Gudenberg, J. W. OOP and Interval Arithmetic—Language Support and Libraries. In *Numerical Software with Result Verification: International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 19-24, 2003. Revised Papers*, edited by R. Alt; A. Frommer; R. B. Kearfott; W. Luther, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN 978-3-540-24738-8, pp. 1–14, doi:10.1007/978-3-540-24738-8_1. Available from: `https://doi.org/10.1007/978-3-540-24738-8_1`

[6] Floating-Point Environment. June 2015, [Online; Accessed 21 June 2017]. Available from: `http://en.cppreference.com/mwiki/index.php?title=c/numeric/fenv&oldid=78965`

[7] Jaulin, L.; Kieffer, M.; et al. Interval Analysis. In *Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics*, London: Springer London, 2001, ISBN 978-1-4471-0249-6, pp. 11–43, doi:10.1007/978-1-4471-0249-6_2. Available from: `https://doi.org/10.1007/978-1-4471-0249-6_2`

[8] Moore, R. E.; Kearfott, R. B.; et al. *Introduction to Interval Analysis*. Philadelphia, PA: Society for Industrial and Applied Mathematics, c2009, ISBN 978-0-89871-669-6.

[9] Goulalard, F. *Gaol 4.2.0 Documentation*. May 2015, [Online; Accessed 7 Septemer 2017]. Available from: `http://frederic.goualard.net/software/gaol-4.2.pdf`

[10] Sutherland, W. A. *Introduction to Metric and Topological Spaces*. Oxford: Oxford University Press, second edition, 2009, ISBN 978-0-19-956307-4.

[11] Berg, G.; Julian, W.; et al. The Constructive Jordan Curve Theorem. *Rocky Mountain J. Math*, volume 5, 1975. Available from: `http://projecteuclid.org/euclid.rmjm/1250130636`

[12] Franek, P.; Ratschan, S.; et al. TopDeg. Program for Topological Degree Calculation. August 2012, [Online; Accessed 28 September 2017]. Available from: `http://www.cs.cas.cz/~franek/topdeg/topdeg.html`

[13] Thareja, R. *Data Structures Using C*. New Delhi: Oxford University Press, second edition, 2014, ISBN 978-0-19-809930-7.

[14] Franek, P.; Ratschan, S.; et al. Quasi-decidability of a Fragment of the First-Order Theory of Real Numbers. *Journal of Automated Reasoning*, volume 57, no. 2, Aug 2016: pp. 157–185, ISSN 1573-0670, doi:10.1007/s10817-015-9351-3. Available from: `https://doi.org/10.1007/s10817-015-9351-3`

[15] Oracle and/or its affiliates. Swing (Java™ Foundation Classes). 2018, [Online; Accessed 11 January 2018]. Available from: `https://docs.oracle.com/javase/7/docs/technotes/guides/swing/`

[16] Boost. Boost Downloads. December 2017, [Online; Accessed 20 November 2017]. Available from: `http://www.boost.org/users/download/`

[17] Boost. Boost Software License. 2017, [Online; Accessed 16 December 2017]. Available from: `http://www.boost.org/users/license.html`

[18] Melquiond, G. Interval Arithmetic Library. December 2006, [Online; Accessed 30 November 2018]. Available from: `http://www.boost.org/doc/libs/1_66_0/libs/numeric/interval/doc/interval.htm`

[19] Lerch, M.; Tischler, G.; et al. Filib++, a Fast Interval Library Supporting Containment Computations. *ACM Transactions on Mathematical Software (TOMS)*, volume 32, no. 2, 2006: pp. 299–324, doi:10.1145/1141885.1141893. Available from: `https://dl.acm.org/citation.cfm?doid=1141885.1141893`

[20] IAMath—a 100% Java, Verifiable Implementation of Interval Arithmetic. 2003, [Online; Accessed 16 December 2017]. Available from: http://interval.sourceforge.net/interval/java/ia_math/README.html

[21] Hickey, T. J.; Qju, Z.; et al. Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. *Reliable Computing*, volume 6, no. 1, Feb 2000: pp. 81–92, ISSN 1573-1340, doi:10.1023/A:1009950630139. Available from: https://doi.org/10.1023/A:1009950630139

[22] Goulalard, F. Gaol: Not Just Another Interval Arithmetics Library. November 2016, [Online; Accessed 9 November 2017]. Available from: https://sourceforge.net/projects/gaol/

[23] ENSTABretagneRobotics. Visualizer for Intervals and Boxes. 2017, [Online; Accessed 4 December 2017]. Available from: http://enstabretagnerobotics.github.io/VIBES/

[24] Drevelle, V.; Nicola, J. VIBes: A Visualizer for Intervals and Boxes. *Mathematics in Computer Science*, volume 8, no. 3, Sep 2014: pp. 563–572, doi:10.1007/s11786-014-0202-0. Available from: https://doi.org/10.1007/s11786-014-0202-0

[25] Nash, P. Catch2 Repository. 2017, [Online; Accessed 16 December 2017]. Available from: https://github.com/catchorg/Catch2

[26] Nash, P. Catch2 Tutorial. 2017, [Online; Accessed 9 December 2017]. Available from: https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md

[27] Freeman, E.; Robson, E.; et al. *Head First Design Patterns*. Beijing: OReilly, 2014, ISBN 978-0596007126.

[28] Meyers, S. *More Effective C++*. Boston: Addison-Wesley, 2014, ISBN 978-0201633719.

[29] Kerrisk, M. Linux Programmer's Manual. February 2018, [Online; Accessed 14 February 2018]. Available from: http://man7.org/linux/man-pages/man7/pthreads.7.html

[30] Garshol, L. M. BNF and EBNF: What are they and how do they work? August 2008, [Online; Accessed 14 February 2018]. Available from: http://www.garshol.priv.no/download/text/bnf.html

[31] The Database of Polynomial Systems. 2017, [Online; Accessed 10 November 2017]. Available from: http://homepages.math.uic.edu/~jan/demo.html

[32] Verschelde, J. *PHCpack Documentation. Release 2.4.47*. 2017, [Online; Accessed 16 November 2017]. Available from: `http://homepages.math.uic.edu/~jan/PHCpack.pdf`

[33] Noonburg, V. W. A Neural Network Modeled by an Adaptive Lotka-Volterra System. *SIAM Journal on Applied Mathematics*, volume 49, no. 6, 1989: pp. 1779–1792, ISSN 0036-1399, doi:10.1137/0149109. Available from: `http://epubs.siam.org/doi/10.1137/0149109`

[34] Patton, R. *Software Testing*. Indianapolis: Sams, 2001, ISBN 0-672-31983-7.

[35] Oracle and/or its affiliates. Oracle VM VirtualBox. 2017, [Online; Accessed 1 February 2018]. Available from: `http://www.oracle.com/technetwork/server-storage/virtualbox/overview/index.html`

[36] Kitware. CMake. 2018, [Online; Accessed 20 January 2018]. Available from: `https://cmake.org/`

Appendix A

Compiling and running the implementation

In the following text, we show how the user can try out our final application. Significant attention is given to the description of our provided GUI.

Our application is intended to be run on GNU/Linux. It depends on Gaol interval arithmetics library [9] and POSIX threads [29]. To quickly and conveniently try out our application, we recommend to use the Oracle VM VirtualBox image [35] we prepared, that contains the application already precompiled and ready to be run. The image is located on the enclosed DVD.

Another option is to build the project from sources, which is intended to be done via CMake [36]. Prior to the compilation, the libraries mentioned above must be present. In order to install Gaol, follow the steps in [9]. Once gaol is installed, navigate to the source folder of our project and type

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

If everything works fine, three executables will be generated.

A.1 List of executables

After a successful build, the build folder contains following executables.

- `./degree`; Command line interface for computing the topological degree of a given function on a given domain.
- `./solver`; The main part of our implementation, solving the problem from Chapter 3. It provides a command line interface for executing Algorithm 4 and allows its parametrization, including to choose the specialization of the frame data structure to be used.

- `./tests/tests`; Unit tests (Section 6.2) and the repetition test from Section 6.5.

Description of the command line interfaces can be found on the enclosed DVD. Apart from those executables, the file `./gui/solvergui.jar` (relative to the source dir.) is our graphical user interface serving as an adapter to `./solver`. It is written in Java and allows the user to collect the input via Java Swing [15] components, execute the solver, and wait for its output, which it then visualizes either via VIBes (Subsection 4.3.1), or via a simple built-in painting method we provided.

The folder `./scripts` contains Python scripts that can be run to reproduce our tests and measurements. These contain both the logic as well as all the input data. The top lines of the scripts can be edited to adjust the paths to the compiled executables etc.

A.2 Description of the provided GUI

To try out the solver, we recommend doing it so via the provided user interface in `./gui/solvergui.jar`. Upon running, the main window (Figure A.1) is displayed, where the input can be entered. The purpose of the particular

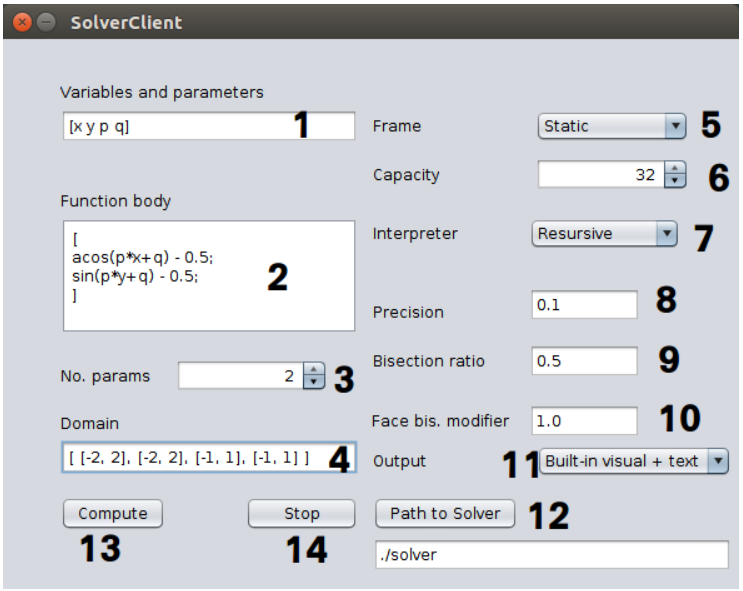


Figure A.1: GUI for our implemented solver.

components numbered in the figure is following.

1. Text field to enter the list of identifiers of variables and parameters. They are separated by space, comma or semicolon and the whole list is

enclosed in square brackets. An Identifier name can contain any combination of letters and numbers and must start with a letter.

2. Text area to enter the function (equation system) body. This is made of a list of arithmetical expressions, each representing one function component, separated by a comma or semicolon. The whole list is enclosed in square brackets. To build expressions, you can use the following.

   - Identifiers of the variables and parameters.
   - Binary operators $+$, $-$, $*$, $\hat{}$, unary operators $+$, $-$, parentheses $(,)$.
   - Strings to denote the elementary functions: exp, log, sin, cos, tan, acos, asin atan, abs, sqr (sqr(x) is equivalent to x^2), sqrt, sqrtn (to compute n-th root; replace the letter "n" with an actual positive integer).
   - Constants π, e, numerical punctual constants (like $0.2, -4$, etc.).
   - Intervals, whose endpoints are expressions, like $[-1, 1]$, $[x + 2, x + 8]$, etc.

   See also Section 5.4.1.

3. Spinner to choose how many of the introduced identifiers represent parameters, counting from the end of the list. In Figure A.1, the number is 2, meaning that the last two identifiers, p and q are parameters, while x and y are variables.

4. Text field to enter domains as list of intervals, separated by commas or semicolons. The list must be enclosed in an extra set of square brackets. The n-th interval in the list is the domain of the n-th defined identifier.

5. Combo box for selecting which frame type from Chapter 3 to use.

6. Combo box for selecting the capacity of the items collection of frames (Section 3.4).

7. Combo box for choosing the type of interpreter to use (Section 5.5).

8. Text field to dictate the depth of the parameter space decomposition (parameter d in Algorithm 4). This number represents the minimal width of parameter domain sub-boxes to be further bisected. Lesser values tend to increase solution quality, but the execution time as well.

9. Text field to select the ratio, in which parameter domain sub-boxes will be bisected (the parameter r in Algorithm 4). This must be strictly between 0 and 1.

10. Text field to set the modifier of the face refinement threshold (the parameter E in Algorithm 4, see also Subsection 3.5.1). It must be greater than 0.

11. Combo box to select the output mode. The result can either be presented via the built-in combination of visual and textual information we implemented, or send to VIBes server to receive an interactive plot (Subsection 4.3.1). Note that the second option requires the VIBes server to be already running.

12. Button to bring up a file dialog to choose path to solver executable, that is to the file `./solver`.

13. Button that starts computation on a worker thread.

14. Button to stop the currently running computation.

Appendix B

Acronyms

AST Abstract syntax tree

IA Interval arithmetics

CLI Command line interface

DFS Depth-first search

GUI Graphical user interface

LSB Least significant bit

Appendix C

Contents of enclosed CD

- `app` ..... main project directory
  - `src` ..... application source files
  - `test` ..... unit tests source files
  - `script` ..... scripts encapsulating experiments and tests
  - `build` ..... directory with executables
  - `gui` ..... GUI client written in Java Swing
- `readme.txt` ..... description of this DVD's content
- `thesis` ..... directory containing the thesis
  - `src` ..... LaTeX source code of the thesis and resources
  - `thesis.pdf` ..... the thesis in pdf format
- `virtual` ... VirtualBox image with pre-compiled ready-to-run application