Formal verification of upper bounds on translative packing densities

I.N. Mulder

Technische Universiteit Delft

TU Delft

Formal verification of upper bounds on translative packing densities

by

I.N. Mulder

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 30, 2019 at 14:00.

Student number:	4235541	
Project duration:	January 1, 2019 – August 30, 2019	
Thesis committee:	Dr. F. M. de Oliveira Filho,	TU Delft, supervisor
	Prof. dr. ir. K. I. Aardal,	TU Delft
	Dr. K. P. Hart,	TU Delft

This thesis is confidential and cannot be made public until August 30, 2019.

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

I have always been fascinated with reasoning and automation. I remember suggesting to 'build a program which could check a proof' for a project in my first year of mathematics (it seemed too hard, so I went with an alternative). Little did I know that I would be doing my thesis on this subject.

Coq is an amazing piece of machinery. However, one might spend non-trivial amounts of time to prove some 'trivial' results, especially when not familiar with Coq. When familiarity has been gained, when you know the proofs well enough, when your definitions are just right... Then developing proofs in Coq can be very satisfying. I hope that proof assistants will become more mathematician-friendly in the future, taking little more effort than the regular way of proving things. They should prevent errors and, when done right, should assist in explaining the reasoning steps in a proof.

In the mean time, conversing with proof assistants shall remain the hobby of those select few crazy enough to enjoy it.

I would like to give a few acknowledgements to some people who have made this thesis possible for me, in no specific order. Gailey, for the blessing of free coffee for the entirety of 2019. Bas, for sharing his food multiple times with a hungry Master's student. Fernando, for taking the time to read and correct this thesis during his holiday. The Squad, for productive starts at 9:15 AM and several non-productive breaks during the summer. Gloomhaveners, for a delightful distraction every week. Eveline, for enduring the never seen before appearance of Ike-who-actually-wants-to-get-up-early. Finally, countless other people for pleasant conversations, fun activities and unwise decisions during the course of my student life. Thank you all!

I.N. Mulder
Delft, August 2019

Abstract

Packing problems are concerned with filling the space with copies of a certain object, so that the least amount of space stays unoccupied. The famous Kepler conjecture asserts that the cannonball packing of spheres is the most efficient packing achievable, and was recently formally proven by Hales.

Dostert, Guzman, Oliveira Filho and Vallentin [5] investigated the packing problem for other shapes. They focused on translative packings, in which all objects are oriented in the same direction. They proved several new upper bounds on translative packing densities of various Platonic and Archimedean solids. However, their results rely heavily on complex computer calculations to ensure they satisfy the conditions of a theorem.

In this thesis, proof assistant Coq is used to formally verify these conditions. An introduction to Coq is provided, aimed at the working mathematician. Then, the theorems required for the proof are developed. Several results from multivariate calculus and convexity were required for the proof, but not available in Coq. The proof also requires a large amount of floating point calculations. A method is developed to efficiently perform floating point calculations on a large scale in Coq.

Using the developed techniques, the improved upper bound of Dostert et al. [5] on the translative packing density of the truncated tetrahedron is formally verified. These techniques can be reused to formally verify the other improved upper bounds of Dostert et al. [5].

Contents

1 Introduction **1**
- 1.1 Packing problems . . . 1
- 1.2 Results by Dostert, Guzmán, Oliveira & Vallentin . . . 2
- 1.3 Formal verification of results . . . 4
- 1.4 Preliminaries . . . 5
  - 1.4.1 Interval arithmetic . . . 5
  - 1.4.2 Fréchet derivatives . . . 6

2 Introduction to Coq **9**
- 2.1 Calculus of Inductive Constructions . . . 9
- 2.2 Curry-Howard correspondence . . . 10
- 2.3 Intuitionism and Coq . . . 11
- 2.4 Syntax of Coq . . . 12
  - 2.4.1 Gallina . . . 12
  - 2.4.2 Vernacular . . . 13
- 2.5 First proofs . . . 14
- 2.6 Recursion and induction . . . 17
- 2.7 Propositional logic . . . 21
- 2.8 Lists . . . 24
- 2.9 Equality in Coq . . . 28
- 2.10 Computation in Coq . . . 30
- 2.11 More tactics . . . 31
- 2.12 Real numbers . . . 31
- 2.13 Interacting with Coq . . . 33
- 2.14 Ltac: custom automation . . . 34

3 Formal verification in Coq **37**
- 3.1 Used packages . . . 38
  - 3.1.1 Interval package for Coq . . . 38
  - 3.1.2 Coquelicot: real analysis for Coq . . . 41
- 3.2 Building required functionalities . . . 42
  - 3.2.1 Interval arithmetic on large expressions . . . 42
  - 3.2.2 Multi-dimensional mean value theorem . . . 50
  - 3.2.3 Proving that ∇f is the derivative of f. . . . 55
  - 3.2.4 Lists of propositions . . . 58
  - 3.2.5 Convexity . . . 60
- 3.3 Specialized verification theorems . . . 68
  - 3.3.1 Proving ∇f is the derivative of f . . . 68
  - 3.3.2 Showing f is nonpositive in a cube outside the Minkowski difference . . . 70
  - 3.3.3 Showing f is nonpositive in a cube intersecting the Minkowski difference . . . 76
  - 3.3.4 Cubes cover the required region . . . 78
- 3.4 Auxiliary verification tools . . . 85
  - 3.4.1 Polynomial definition from a solution file . . . 86
  - 3.4.2 Alternative representation of polynomial . . . 86
  - 3.4.3 Manual expressions for interval_containers . . . 86
  - 3.4.4 Generating the verification files . . . 87
  - 3.4.5 Dividing the verification load . . . 87

4 Conclusion **89**
4.1 Results . . . 89
4.2 Discussion . . . 90
4.2.1 Validity of results . . . 90
4.2.2 Possible improvements . . . 92
4.2.3 Experience with Coq . . . 92
4.3 Further work . . . 92

A Auxilliary formal definitions and propositions **95**
A.1 Linear combinations . . . 95
A.2 Sums . . . 96
A.3 Affine functions . . . 97

B Verified results for the truncated tetrahedron **99**

Bibliography **103**

1

Introduction

1.1. Packing problems

Packing problems in mathematics are concerned with the question: How can we fill a space with a certain shape most efficiently? The most famous example of this problem considers spheres to fill $\mathbb{R}^3$. Johannes Kepler [10] was already thinking of this problem in 1611. He suspected that the best way to pack spheres in some container is by using the hexagonal close packing, or 'cannonball' packing, shown in Figure 1.1. This packing has a density equal to $\frac{\pi}{\sqrt{18}}$.

His conjecture, now dubbed the Kepler conjecture, turned out to be too hard to solve at the time. Two centuries later, in 1831, Gauss [6] proved a partial result; Kepler's conjecture was at least true for the (more regular) lattice packings. Only recently, in 1998, Thomas Hales announced he had a proof of the Kepler conjecture. His proof relied heavily on complex computer calculations, and this made it hard to verify for referees. This encouraged Hales to start the Flyspeck Project, which aimed to formally verify his proof using proof assistants; computer programs which can check mathematical proofs. The Flyspeck Project was completed in 2014, and published in 2017 [7].

Naturally, the packing problem was posed for different shapes and dimensions. The optimal sphere packing problem in $\mathbb{R}^8$ was recently solved by Maryna Viazovska [17], and, soon after, in $\mathbb{R}^{24}$ by Cohn et al [4], building on Viazovska's result. Their approach uses specific properties appearing in these dimensions. Jiao, Stillinger and Torquato [9] constructed packings of superballs in $\mathbb{R}^3$, shapes defined by

$$B_3^p = \{x \in \mathbb{R}^3 : |x_1|^p + |x_2|^p + |x_3|^p \leq 1\}$$

for various $p \geq 1$. Torquato and Jiao [15] continued this work and constructed packings for nonspherical objects such as the classic Platonic and Archimedean solids. Note that constructing a packing is the same thing as proving a lower bound on the highest density of packings. None of the packing problems have been 'solved' for the aforementioned shapes, so bounds are the best thing we have as of writing.

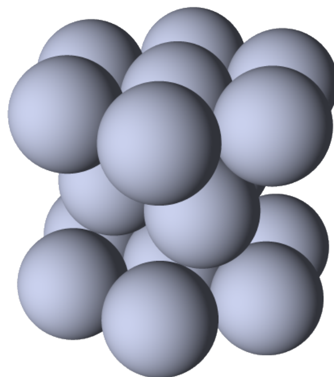


Figure 1.1: Hexagonal close packing of spheres

We distinguish three types of packings for a body $\mathcal{K}$ in $\mathbb{R}^n$, where $\mathcal{K}$ has non-empty interior. The first type is a *congruent* packing. Informally, this is a packing of congruent copies of $\mathcal{K}$, which are allowed to touch but not overlap. We can define this as

$$\mathcal{P} = \bigcup_{i \in \mathbb{N}} A_i \mathcal{K} + x_i,$$

where $x_i \in \mathbb{R}^n$ and $A_i \in \mathrm{SO}(n)$, and for $i \neq j$ we have

$$(x_i + A_i \mathcal{K}^\circ) \cap (x_j + A_j \mathcal{K}^\circ) = \emptyset.$$

Here $\mathrm{SO}(n) = \{A \in \mathbb{R}^{n \times n} : AA^T = I, \det(A) = 1\}$ is the group of rotations in $\mathbb{R}^n$, and $\mathcal{K}^\circ$ stands for the interior of $\mathcal{K}$.

The next type of packing is a *translative* packing. This is a more restrictive type of packing. Here we require that the copies of $\mathcal{K}$ are not only congruent, but identical; that is to say, all A_i are equal to the identity. The last type of packing is a *lattice* packing, as used by Gauss. The x_i can no longer be chosen arbitrary, but must form a lattice.

It was mentioned before that to prove a lower bound on the highest density of a packing of shapes, one needs to construct a single packing attaining that density. To prove an upper bound, we need to do something conceptually harder; we need to show that all possible packings have lower density. When considering translative or congruent packings, the lack of structure on the x_i makes this quite hard. We will use the following theorem by Cohn and Elkies [2].

Theorem 1. *Let $\mathcal{K}$ be a convex body in $\mathbb{R}^n$ and let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a continuous L^1-function (a continuous function whose absolute value is Lebesgue integrable). Define the Fourier transform of f at u by*

$$\widehat{f}(u) = \int_{\mathbb{R}^n} f(x) e^{-2\pi i u \cdot x} \mathrm{d}x.$$

Suppose f satisfies the following conditions:

(i) $\widehat{f}(0) \geq 1$
(ii) *f is of positive type, that is $\widehat{f}(u) \geq 0$ for every $u \in \mathbb{R}^n$*
(iii) *$f(x) \leq 0$ whenever $\mathcal{K}^\circ \cap (x + \mathcal{K}^\circ) = \emptyset$*

Then the density of every translative packing of $\mathcal{K}$ in $\mathbb{R}^n$ is at most $f(0)$ vol $\mathcal{K}$

We will not prove this result here. For a proof, see Cohn and Kumar [3]. The result was originally stated in Cohn and Elkies [2] for a smaller class of functions.

This theorem was used to prove the best known upper bounds of sphere packings in dimension 4 to 36. The bounds for dimension 8 and 24 were eventually proven tight by the previously mentioned result of Viazovska [17] and Cohn et al [4]. To simplify the theorem, one can restrict the function f to be a radial function (a function where $f(x)$ depends only on the norm of x). This is convenient for spheres, but less so for other convex bodies. Dostert, Guzmán, Oliveira, and Vallentin [5] showed how to use Theorem 1 to obtain better bounds for nonspherical objects.

1.2. Results by Dostert, Guzmán, Oliveira & Vallentin

Dostert et al [5] were able to use Theorem 1 to prove improved upper bounds for various nonspherical objects. They proved better upper bounds for translative and lattice packings of superballs, and better upper bounds for translative packings of Platonic and Archimedean solids with tetrahedral symmetry.

They noticed that it is natural to consider functions f which are invariant under the left action of $S(\mathcal{K} - \mathcal{K})$, where

$$S(\mathcal{B}) = \{A \in O(n) : A\mathcal{B} = \mathcal{B}\}$$

is the symmetry group of a convex body $\mathcal{B}$, and

$$\mathcal{K} - \mathcal{K} = \{k_1 - k_2 : k_1, k_2 \in K\}$$

is the Minkowski difference of $\mathcal{K}$ with itself. Note that for $A \in S(\mathcal{K}-\mathcal{K})$ we have

$$\begin{aligned}\left(A^{-1}x+\mathcal{K}\right)\cap\mathcal{K}=\emptyset &\iff (x+A\mathcal{K})\cap A\mathcal{K}=\emptyset\\ &\iff \forall k_1,k_2\in\mathcal{K}: x+Ak_2\neq Ak_1\\ &\iff \forall k_1,k_2\in\mathcal{K}: x\neq Ak_1-Ak_2\\ &\iff x\notin A(\mathcal{K}-\mathcal{K})\\ &\iff x\notin\mathcal{K}-\mathcal{K}\\ &\iff (x+\mathcal{K})\cap\mathcal{K}=\emptyset.\end{aligned}$$

Recall requirement (iii) of Theorem 1: $f(x) \leq 0$ whenever $(x+\mathcal{K}^\circ)\cap\mathcal{K}^\circ$. The derivation above shows that it is natural to require that $f(A^{-1}x) = f(x)$ for all $A \in S(\mathcal{K}-\mathcal{K})$.

If f is invariant under $S(\mathcal{K}-\mathcal{K})$, then so must $\widehat{f}$ be. To force f to be well behaved (i.e. continuous and in $\mathrm{L}^1(\mathbb{R}^n)$), Dostert et al. assume furthermore that $\widehat{f}$ can be written as

$$\widehat{f}(u) = g(u)e^{-\pi\|u\|^2},$$

where g is a polynomial. This forces $\widehat{f}$ to be a Schwartz function by the rapid decreasing of $e^{-\pi\|u\|^2}$. Since the Fourier transform is an automorphism on the Schwartz-space, this makes f a Schwartz function as well, and thus a continuous and L^1-function in particular. Also, since $\widehat{f}$ is invariant under $S(\mathcal{K}-\mathcal{K})$, so must the polynomial g be.

Dostert et al. focus on superballs B_3^p and on Platonic and Archimedean solids with tetrahedral symmetry. This is because for both of these groups, $S(\mathcal{K}-\mathcal{K}) = \mathsf{B}_3$, the octahedral group. The octahedral group B_3 is the symmetry group of the regular cube $[-1,1]^3$. This group identification is useful because we have information on B_3-invariant functions. It can be shown that all B_3-invariant polynomials are freely generated by three basic invariant polynomials:

$$s_1 = x_1^2+x_2^2+x_3^2, \qquad s_2 = x_1^4+x_2^4+x_3^4, \quad \text{and} \quad s_3 = x_1^6+x_2^6+x_3^6.$$

This means that our B_3-invariant polynomial g lies in the polynomial ring $\mathbb{R}[s_1,s_2,s_3]$.

Dostert et al. proceed by translating all requirements of Theorem 1 to linear or semidefinite programming conditions. Condition (i), $\widehat{f}(0) = g(0) \geq 1$, is of course a linear condition on the constant coefficient of g. To guarantee that $g(u) \geq 0$, which is equivalent to $\widehat{f}(u) \geq 0$, they require that g can be written as a sum of squares, a semidefinite condition. For condition (iii), f needs to be computed from $\widehat{f}$. They show this can be done using a linear transformation. Next, to ensure $f(x) \leq 0$ whenever $x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$, they require that $f(x)$ can be written as

$$f(x) = \left(-s(x)q_1(x) - q_2(x)\right)e^{-\pi\|x\|^2}$$

where $s(x)$ is usually taken to be $\|x\|^2 - r^2$, and q_1 and q_2 are again polynomials in $\mathbb{R}[s_1,s_2,s_3]$. If q_1 and q_2 can be written as sums of squares, we have $f(x) \leq 0$ for $s(x) \geq 0$ (so usually $\|x\| \geq r$) immediately. This requirement on q_1 and q_2 is another semidefinite condition. To make sure $f(x) \leq 0$ whenever $s(x) < 0$ and $x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$, additional linear conditions are posed at specific points in this domain. The idea is that, if f is required to be nonpositive at enough points, f will be nonpositive on the entire remaining domain.

Semidefinite programming solvers are then used to find good candidates minimizing $f(0)$. These should be treated as a 'black box'; we do not know exactly how the solver ends up at a particular solution, and thus the validity of solutions is questionable. The conditions entered into the solver *should* guarantee that solutions are valid, but floating point calculations are imprecise and prone to rounding errors. To verify their results, Dostert et al. wrote a program which verifies whether a given function f satisfies the condition of Theorem 1.

The first part of this program verifies whether the solution satisfies the semidefinite programming constraints. Solvers often end up at solutions that are nearly feasible due to floating point errors. If the solution is close enough to being feasible, it can sometimes be made feasible by slightly perturbing the solution. Note that once a feasible solution has been obtained, it does immediately satisfy the first two conditions of Theorem 1. The third condition requires more investigation.

Recall that for the third condition we need to check whether $f(x) = (-s(x)q_1(x) - q_2(x))e^{-\pi\|x\|^2}$ is nonpositive. The exponential factor always has positive sign, so we want to show that $-s(x)q_1(x) - q_2(x)$

is nonpositive. This is immediately satisfied for $s(x) \geq 0$, since q_1 and q_2 are sums of squares. The remaining area, where $s(x) < 0$ and $x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$, still needs to be investigated. Here q_1, q_2, s, and thus $-s(x)q_1(x) - q_2(x)$ are all B_3-invariant polynomials. Since the exponential factor is not relevant for verifying the third condition, we will disregard it and (re)define f as $f = -s(x)q_1(x) - q_2(x)$ for the next part. Thus, f is a B_3-invariant polynomial.

The semidefinite programming constraints guarantee nonpositivity at some finite set of points inside this area. It is plausible that, when the grid is fine enough, this guarantees nonpositivity between these points, but nonpositivity cannot be guaranteed when close to the border of $\mathcal{K} - \mathcal{K}$. Indeed, for some of the bodies the function f does not satisfy the requirements of Theorem 1 without compensating for this. An additional factor $\alpha > 1$ is required, for which $f(x) \leq 0$ whenever $(x + \alpha\mathcal{K}^\circ) \cap \alpha\mathcal{K}^\circ = \emptyset$. Still, even then it is not guaranteed by the conditions that $f(x)$ is actually nonpositive when $s(x) < 0$ and $x \notin \alpha(\mathcal{K}^\circ - \mathcal{K}^\circ)$. The second part of the program verifies whether this is true at all remaining points. Because of the B_3-invariance of f, it suffices to check $f(x_1, x_2, x_3) \leq 0$ when $0 \leq x_1 \leq x_2 \leq x_3$. This is easy to see when recalling the definitions of s_1, s_2, and s_3, the free generators of the ring of B_3-invariants. Next, the domain is divided into cubes, and nonpositivity is checked inside each cube.

Each cube $\mathcal{C}$ is given as a product of intervals $[y_1, y_1+\delta] \times [y_2, y_2+\delta] \times [y_3, y_3+\delta]$, where δ is the side of the cube. The gradient ∇f of f is computed, and evaluated on the cube using interval arithmetic. More will be explained about interval arithmetic in Section 1.4.1, but the result of this operation will be some interval for which

$$(l_1, l_2, l_3) \leq \nabla f(x) \leq (u_1, u_2, u_3)$$

whenever $x \in \mathcal{C}$. This also allows computation of a number $\nu_{\mathcal{C}}$ for which $\|\nabla f(x)\| \leq \nu_{\mathcal{C}}$. Next, the following consequence of the mean-value theorem in $\mathbb{R}^n$ is used:

Theorem 2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable on $\mathbb{R}^n$ and let $x, y \in \mathbb{R}^n$. If for all $0 \leq c \leq 1$ we have $\|\nabla f((1-c)x + cy)\| \leq D$ for some $D \in \mathbb{R}$, then*

$$|f(x) - f(y)| \leq D\|x - y\|.$$

This means that if f is evaluated at a number of points in $\mathcal{C}$, negativity at these points proves nonpositivity in some region around these points. The size of this region depends on the number D. So, we divide each side of the cube into N intervals for some integer $N \geq 1$. These gridpoints $\mathcal{C}_N$ are given by

$$\mathcal{C}_N = \{(y_1, y_2, y_3) + (a, b, c)\delta/N : a, b, c \in 0, \ldots, N\}.$$

Finally, we need to show that every $x \in \mathcal{C}$ for which $x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$, x is in the nonpositivity area around some gridpoint. Note that $\mathcal{C}$ is not completely inside $\mathcal{K}^\circ - \mathcal{K}^\circ$, since then we do not need to check nonpositivity in this cube. It is enough to estimate the distance of any $x \in \mathcal{C}$ which is not in $\mathcal{K}^\circ - \mathcal{K}^\circ$ to the closest gridpoint not in $\mathcal{K}^\circ - \mathcal{K}^\circ$. Suppose $\mathcal{C}$ intersects $\mathcal{K}^\circ - \mathcal{K}^\circ$. It is clear that every x is contained in some subcube with vertices contained in $\mathcal{C}_N$. If x is outside $\mathcal{K}^\circ - \mathcal{K}^\circ$, at least one of the vertices must be outside $\mathcal{K}^\circ - \mathcal{K}^\circ$ as well, since otherwise x is also inside by convexity of $\mathcal{K}^\circ - \mathcal{K}^\circ$. In the worst case, the closest gridpoint which is not in $\mathcal{K}^\circ - \mathcal{K}^\circ$ is the farthest vertex in the subcube, with distance at most $\sqrt{3}\delta/N$.

If the cube $\mathcal{C}$ does not intersect $\mathcal{K}^\circ - \mathcal{K}^\circ$, all vertices of the subcube containing x are outside $\mathcal{K}^\circ - \mathcal{K}^\circ$ in particular. The closest vertex x_N of the subcube satisfies $\|x - x_N\| \leq \sqrt{3}\delta/(2N)$.

Picking cubes and gridsizes is a challenge in itself. The program usually starts with cubes with sidelength equal to 1/300, and some specified maximum gridsize. If this is not sufficient, the cube is split into 8 subcubes with half the sidelength, until it is sufficient. Starting with a large sidelength saves on gradient computations, which are expensive, but causes the gradient bounds to become worse, possibly resulting in more expensive cube splittings.

This verification program was written in C++, and is no small program. This program makes the result more plausible, but the program could still contain software bugs, invalidating the result. The program could even be free of bugs, but any of the used libraries might still contain bugs. How can we be absolutely certain the result is true?

1.3. Formal verification of results

This thesis aims to formally verify the improved upper bounds on packing densities shown by Dostert et al. Like Hales, we will use a proof assistant to rigorously show the results are true.

For a complete verification, all used theorems need to be shown in the proof assistant. However, Theorem 1 itself is not in question. The functions found by Dostert et al. are also believed to satisfy the first two conditions. We will therefore focus on proving that the functions Dostert et al. found satisfy the bounded part of the third condition of Theorem 1, i.e. that

$$f(x) \leq 0 \qquad \text{whenever } x \notin \mathcal{K}^\circ - \mathcal{K}^\circ \text{ and } s(x) < 0.$$

We will use the same verification strategy, cubes and gridsizes in the proof assistant as those used in the C++ program, so that any difference in results can be scrutinized.

The author was not familiar with proof assistants before starting this thesis, and various proof assistants are available. To name a few: HOL Light (used by Hales), Isabelle, Agda, Mizar and Coq. We chose Coq because of its strong type system, the ability for a user to automate tasks, and a large mathematical library. For a comparison, see Wiedijk [18]. Coq's library also contains a package with support for interval arithmetic, which is frequently needed in our verification strategy.

Some of the readers might not be familiar with proof assistants in general, or Coq in particular. Presenting a formal proof in a language unknown to the reader is somewhat vacuous. So, to get up to speed, Coq is explained in some detail in Chapter 2. This should enable the reader to read proofs written in Coq when interacting with the proof assistant.

In Chapter 3, the various steps needed for full verification in Coq will be explained. This will first require us to develop various mathematical results in the proof assistant. Once this has been done, we can build results to verify nonpositivity of a function inside a cube. Finally, we will show in Coq that all cubes together cover the required area.

Chapter 4 will cover the results of this thesis. We will explain what has been verified, on what the verification is based and discuss further work.

1.4. Preliminaries

Before we start with the next chapter, we will first introduce two topics which will be used in this thesis: interval arithmetic and Fréchet derivatives.

1.4.1. Interval arithmetic

Interval arithmetic is useful when you have bounds on some real variables, and you want bounds on an expression over these variables. For example, suppose we know $x \in [a, b]$ and $y \in [c, d]$. It is easy to see that we must have $x + y \in [a + c, b + d]$. We could interpret this last interval as the sum of the previous two, suggestively writing

$$[a, b] + [c, d] = [a + c, b + d].$$

This definition of addition on intervals is nice, because if $x \in I_1$ and $y \in I_2$, then $x + y \in I_1 + I_2$. For subtraction we can define

$$[a, b] - [c, d] = [a - d, b - c].$$

Again, this gives us $x - y \in I_1 - I_2$ if $x \in I_1$ and $y \in I_2$. Note that this subtraction operation does not satisfy the group-equality $I_1 - I_1 = [0, 0]$, since

$$[a, b] - [a, b] = [a - b, b - a].$$

This is one of the weaknesses of interval arithmetic: although $x - x = 0$ is indeed contained in the interval $I_1 - I_1 = [a - b, b - a]$, these bounds are too rough. Interval arithmetic is not well suited for bounding expressions with internal dependencies: $x - x$ is bounded in the same way as $x - y$, where $y \in I_1$ as well.

These $+$ and $-$ operations satisfy the so-called containment property. For general functions f on real variables, we would like to construct functions operating on intervals with the same property. To do so, let us first define what this containment property means.

Borrowing notation from [11], we will write $\mathbf{x}$ for a (closed) interval enclosing the variable x. Furthermore, the lower and upper bounds will be denoted as $\underline{x}$ and $\overline{x}$ respectively, so that $x \in \mathbf{x} = [\underline{x}, \overline{x}]$. Suppose we have some function $f : \mathbb{R} \to \mathbb{R}$ and are interested in $f(\mathbf{x})$, that is

$$f(\mathbf{x}) = \{f(r) : r \in \mathbf{x}\}.$$

Since we have a general function f, it is possible that this set is near impossible to calculate; it depends on all values between $\underline{x}$ and $\overline{x}$. It might therefore be more efficient to instead evaluate some $\mathbf{f}(\mathbf{x})$, which contains $f(\mathbf{x})$ and should be easier to calculate. By containing, we mean that

$$\forall \mathbf{y} \subseteq \mathbb{R} : f(\mathbf{y}) \subseteq \mathbf{f}(\mathbf{y})$$

where the quantification is not over all subsets, but over all intervals in $\mathbb{R}$. The function $\mathbf{f}$ should furthermore be interval-valued; i.e. $\mathbf{f}(\mathbf{y}) = [\underline{\mathbf{f}(\mathbf{y})}, \overline{\mathbf{f}(\mathbf{y})}]$. Functions $\mathbf{f}$ satisfying this containment property are not unique, since any coarser interval also satisfies the containment property.

The following theorem allows us to build containing functions $\mathbf{f}$ for complicated f.

Theorem 3 (Fundamental theorem of interval analysis)**.** *Let $\mathbf{f}$ and $\mathbf{g}$ satisfy the containment property for real functions f and g respectively. Then $\mathbf{f} \circ \mathbf{g}$ satisfies the containment property for $f \circ g$.*

Proof. Let $x \in [\underline{x}, \overline{x}] =: \mathbf{x}$. By the containment property of $\mathbf{g}$, we have $g(x) \in g(\mathbf{x}) \subseteq \mathbf{g}(\mathbf{x})$. Since $\mathbf{g}(\mathbf{x})$ is an interval itself, we have $f(g(x)) \in f(\mathbf{g}(\mathbf{x})) \subseteq \mathbf{f}(\mathbf{g}(\mathbf{x}))$. This concludes the proof. □

This means that once we have containing functions for elementary functions (addition, multiplication, taking powers, etc.), we can compose them to get containing functions for the composed functions. Polynomials are composed of addition, multiplication, and powers, so we can build a containing function for any polynomial. This enables us to calculate bounds on a polynomial in a given interval.

1.4.2. Fréchet derivatives

For reasons which will be come clear later on, the regular notion of differentiability is not always suitable. By the regular notion of differentiability, we mean the following.

Definition 1. *Let $f : \mathbb{R} \to \mathbb{R}$ be a function and x a real number. We call f differentiable at x if:*

$$\lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

exists. If this limit is equal to $l \in \mathbb{R}$, we call l the derivative of f at x.

This works fine for functions $f : \mathbb{R} \to \mathbb{R}$, since we can do such a division with h. Note that

$$\lim_{h\to 0} \frac{f(x+h) - f(x)}{h} = l \iff \lim_{h\to 0} \left| \frac{f(x+h) - f(x)}{h} - l \right| = 0.$$

We would like a similar notion of differentiability on Banach spaces U, where we cannot divide by an element $h \in U$. We can divide by the norm of h, so this motivates rewriting the limit to:

$$\lim_{h\to 0} \frac{\|f(x+h) - f(x) - l \cdot h\|}{\|h\|} = 0$$

However, something like $l \cdot h$ is only available on Hilbert spaces, where we have an inner product $\langle l, h \rangle$. A more suitable notion would be to replace $l \cdot h$ with $A(h)$, where A is a linear operator. This motivates the following definition:

Definition 2 (Fréchet differentiability)**.** *Let U, V be Banach spaces and $f : U \to V$ be a function. We call f Fréchet differentiable at $x \in U$, if there exists a bounded linear operator $A : U \to V$, with*

$$\lim_{h\to 0} \frac{\|f(x+h) - f(x) - A(h)\|_V}{\|h\|_U} = 0.$$

This operator A is called the Fréchet derivative of f at x.

We can interpret this operator A as the *best linear approximation* of f at x. To link this back to the regular notion of differentiability: suppose we have a differentiable function $f : \mathbb{R} \to \mathbb{R}$, whose derivative is $f'(x)$ at $x \in \mathbb{R}$. The Fréchet derivative of f at $x \in \mathbb{R}$ is then given by $h \mapsto f'(x)h$. For differentiable functions $g : \mathbb{R}^n \to \mathbb{R}$, we get that the Fréchet derivative of g at $x \in \mathbb{R}^n$ is given by $h \mapsto \nabla g(x) \cdot h$, where $\cdot$ now denotes the Euclidean inner product on $\mathbb{R}^n$.

Fréchet derivatives also satisfy a version of the chain rule.

Theorem 4 (Chain rule for Fréchet derivatives)**.** *Let U, V, and W be Banach spaces, $f : U \to V$ and $g : V \to W$ be some functions. Suppose the Fréchet derivative of f at x is A, and the Fréchet derivative of g at $f(x)$ is B. Then $g \circ f$ is Fréchet differentiable at x, with Fréchet derivative $B \circ A$.*

Proof. We will show that

$$\lim_{h \to 0} \frac{\|g(f(x+h)) - g(f(x)) - B(A(h))\|_W}{\|h\|_U} = 0.$$

Fix $\epsilon > 0$. Using the Fréchet differentiability of f at x, pick δ_1 so that for $h \in U$,

$$\|h\|_U < \delta_1 \implies \frac{\|f(x+h) - f(x) - A(h)\|_V}{\|h\|_U} < \min\left(\frac{\epsilon}{2(\|B\|+1)}, 1\right),$$

where $\|B\|$ stands for the $V \to W$ operator norm of B. Then, using the Fréchet differentiability of g at $f(u)$, pick δ_2 so that for $h \in V$,

$$\|h\|_V < \delta_2 \implies \frac{\|g(f(x)+h) - g(f(x)) - B(h)\|_W}{\|h\|_V} < \frac{\epsilon}{2(1+\|A\|)}.$$

Let $\delta = \min\left(\delta_1, \frac{\delta_2}{1+\|A\|}\right)$. Then for $h \in U$ with $\|h\|_U < \delta$:

$$\begin{aligned}
\|f(x+h) - f(x)\|_V &= \|f(x+h) - f(x) - A(h) + A(h)\|_V \\
&\leq \|f(x+h) - f(x) - A(h)\|_V + \|A(h)\|_V \\
&\leq 1 \cdot \|h\|_U + \|A\|\|h\|_U \\
&= (1+\|A\|)\|h\|_U
\end{aligned}$$

Let us define $h_2 = f(x+h) - f(x)$. Then, using that B is linear,

$$\begin{aligned}
\frac{\|g(f(x+h)) - g(f(x)) - B(A(h))\|_W}{\|h\|_U} &= \frac{\|g(f(x+h)) - g(f(x)) - B(h_2) + B(h_2 - A(h))\|_W}{\|h\|_U} \\
&\leq \frac{\|g(f(x)+h_2) - g(f(x)) - B(h_2)\|_W}{\|h\|_U} + \frac{\|B(h_2 - A(h))\|_W}{\|h\|_U}.
\end{aligned}$$

Since $\|h_2\|_V \leq (1+\|A\|)\|h\|_U < \frac{1+\|A\|}{1+\|A\|}\delta_2 = \delta_2$, we know that

$$\begin{aligned}
\frac{\|g(f(x)+h_2) - g(f(x)) - B(h_2)\|_W}{\|h\|_U} &= \frac{\|g(f(x)+h_2) - g(f(x)) - B(h_2)\|_W}{\|h_2\|_V} \cdot \frac{\|h_2\|_V}{\|h\|_U} \\
&< \frac{\epsilon}{2(1+\|A\|)} \cdot \frac{(1+\|A\|)\|h\|_U}{\|h\|_U} = \frac{\epsilon}{2}.
\end{aligned}$$

For the other part, since $\|h\|_U < \delta_1$, we have

$$\begin{aligned}
\frac{\|B(h_2 - A(h))\|_W}{\|h\|_U} &= \frac{\|B(f(x+h) - f(x) - A(h))\|_W}{\|h\|_U} \\
&\leq \|B\| \cdot \frac{\|f(x+h) - f(x) - A(h)\|_V}{\|h\|_U} \\
&< \|B\| \cdot \frac{\epsilon}{2(\|B\|+1)} \\
&= \frac{\epsilon}{2} \cdot \frac{\|B\|}{\|B\|+1} < \frac{\epsilon}{2}.
\end{aligned}$$

Thus $\frac{\|g(f(x+h)) - g(f(x)) - B(A(h))\|_W}{\|h\|_U} < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon$ as required. □

2
Introduction to Coq

Proof assistants such as Coq have been around for a while. Still, they are not widely in use in the mathematical community. The reader might not be familiar with proof assistants in general or Coq in particular before reading this thesis. Since Coq will be the cornerstone of the formal verification we wish to perform, the following sections will try to make an introduction to Coq.

Several other tutorials exist, all aimed at different audiences. Coq is often used by theoretical computer scientists, for example to verify compilers. It is also used extensively in logic and other more foundational parts of mathematics, and this shows in the tutorials.

This introduction is aimed at mathematicians for whom the foundational part of Coq is less important. One does need some basic understandig of it, so we will touch upon the foundation and its motivation, but will try to keep ourselves to the parts which have direct influence on interacting with Coq. Then, we will see how one goes about translating a theorem and its proof to Coq. This entails knowing how to manipulate the proof environment, and knowing how to define types. The concept of computation and equality in Coq will also be discussed.

Next, we will discuss the real number library in Coq, as well as some other useful features. Some proof examples will also be given.

We conclude with an explanation of more advanced usage of Coq.

2.1. Calculus of Inductive Constructions

Coq's underlying formal language is called the Calculus of Inductive Constructions, or CiC for short. It is based on the λ-calculus, enriched with a notion of types. One might expect Coq to be based on the more known foundational basis of sets. Sets are intuitive objects for humans, but they are less convenient for computers. The λ-calculus is inherently computational, so more apt as a basis for a computer program which should check mathematical proofs.

The original λ-calculus contains 3 types of terms:

- Variables x;
- λ-terms, or function definitions: $(\lambda x.M)$. $(\lambda x.M)$ is a function sending an input x to M, where M possibly contains instances of x;
- Application: $(M\ N)$, applying the function M to the argument N.

A 'term' is a syntactically valid element of the language. One can reduce or rewrite terms in the λ-calculus, by using the following rule, called β-reduction,

$$(\lambda x.M)N = M[x := N]$$

where $M[x := N]$ signifies the term M, where all instances of x have been replaced with N.

The λ-calculus is strong enough to encode the natural numbers. This is usually done using Church numerals, for which the first three numbers are defined as follows:

$$\begin{aligned} 0 &= (\lambda f.\lambda x.x), \\ 1 &= (\lambda f.\lambda x.f\ x), \\ 2 &= (\lambda f.\lambda x.f\ (f\ x)). \end{aligned}$$

Each natural number n sends a function f to the function which applies f n times.

This original λ-calculus has a couple of shortcomings. For example, we are syntactically allowed to apply any function to any argument; there is no such thing as the domain of a function. This would be nice to have. Furthermore, the construction of the natural numbers is somewhat convoluted. It *encodes* the natural numbers in the existing language features, opposed to being able to define the natural numbers using the existing language.

CiC fixes both these shortcomings. Every term (syntactically valid element) in CiC has a type. We write $x : T$ to signify that the term x has type T. Functions can now only be applied to variables of the correct type. This can be enforced, since functions are terms and thus have a type. For example, $(\lambda x : T.x) : (T \to T)$.

Types can be defined using the $\to$ operator, where $X \to Y$ is the type of functions sending arguments of type X to images of type Y. However, for the natural numbers we would like a new type. CiC allows us to build such a type, in the following way.

```
Inductive nat :=
  | zero : nat
  | succ : nat → nat.
```

This is an inductive definition, meaning that we do not explicitly specify all the elements of our new type. Instead, we give one element `zero : nat`, and a successor function `succ : nat → nat`. Every element `x : nat` is then either equal to `zero`, or equal to `succ x'` for some other `x' : nat`. The 'either' in the previous sentence is exclusive; `zero` is distinct from `succ n` for any `n : nat`. In this way, we have automatically satisfied a good deal of the Peano axioms using just the definition.

In CiC, types are also terms themselves. For our natural numbers, we have `nat : Set`. `Set` aims to be (one of the) hierarchically smallest types, whose terms mention no other types. The hierarchy we mention can be seen when looking at the type of `Set`. In CiC, `Set : Type`(1) and `Type`(1) : `Type`(2). This hierarchy is needed, because an upper type `Type` for which `Type : Type` leads to a contradiction. This contradiction is called Girard's paradox [8], and is similar to Russel's paradox in set theory. When writing `Type` in Coq, it automatically infers the needed type-depth, so that the user does not need to worry about this. From the user point of view we then have `Type : Type`.

Another important basic type is `Prop`. Like `Set`, this type is also at the bottom of the type hierarchy. `Prop` aims to be the type of mathematical propositions. Propositions might or might not have a proof. For example, we have (`∀ n : nat, n = zero`) : `Prop` and (`zero = zero`) : `Prop`. A proof of a proposition `P : Prop` is a term `p : P`. This might seem weird at first, but this identification is natural by the Curry-Howard correspondence.

2.2. Curry-Howard correspondence

The Curry-Howard correspondence or isomorphism is a beautiful relation between two seemingly unrelated fields: programming and mathematical logic. It identifies propositions with types, and proofs of a proposition with terms of the corresponding type. Proving a proposition is then the same thing as giving an algorithm to construct a term of the correct type. We will cover three examples where this identification can be seen.

Let us first consider the statement $\exists x \in \mathbb{N}, x + 0 = 0$. This is quite trivial to prove, by substituting $x = 0$. Let us investigate the exact details of our proof. We wanted to prove existence, and we did so by giving a *witness*, for which the statement was true. One could see this as an algorithm, giving us an element which satisfies the statement.

Now consider the statement '$f(x) = 2x$ is continuous at 0'. By the ϵ-δ definition of continuity, we need to show that:

$$\forall\, \epsilon > 0, \exists\, \delta > 0 : |x - 0| < \delta \implies |f(x) - f(0)| < \epsilon$$

We would prove this as follows. Let $\epsilon > 0$, and take $\delta = \epsilon/2$. By substituting $f(x) = 2x$ and taking the 2 out of the absolute value, we immediately have $|x| < \epsilon/2 \implies |2x| < \epsilon$, so this δ suffices. Our proof now consists of a function $\delta(\epsilon) = \epsilon/2$, which sends a given ϵ to the δ for which the statement is true.

Finally, let us consider a proof by induction. We will prove '$\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2$'. For the base case $n = 0$, the left and right side evaluate to 0, so we are done. Suppose then $n = m + 1$, for

some $m \in \mathbb{N}$, and we know that $\sum_{i=0}^{m} i = m(m+1)/2$. In that case,

$$\begin{aligned}\sum_{i=0}^{n} i &= \sum_{i=0}^{m+1} i \\ &= \sum_{i=0}^{m} i + (m+1) \\ &= \frac{m(m+1)}{2} + \frac{2 \cdot (m+1)}{2} \\ &= \frac{(m+1)(m+2)}{2} = \frac{n(n+1)}{2}.\end{aligned}$$

Let us define the proposition $P(n) = \sum_{i=0}^{n} i = n(n+1)/2$. Our proof of $\forall n \in \mathbb{N}, P(n)$ consisted of two parts: the statement $P(0)$, and the implication that $\forall n \in \mathbb{N}, P(n) \implies P(n+1)$. This implication can be seen as an algorithm: given a proof of $P(n)$, we can construct a proof of $P(n+1)$. The induction principle does the rest. The induction principle itself can also be viewed as a recursive algorithm: given a number n, it constructs a proof of $P(n)$. If $n = 0$, we return the proof of $P(0)$. If $n = m + 1$, we use our implication function to turn a proof of $P(m)$ into a proof of $P(m+1)$. The proof of $P(m)$ can be found using our recursive algorithm.

We can also identify some functions with proofs. Consider the identity function for some type T, $\text{id}(t) = t$. We then have that $\text{id} : T \to T$, since id sends arguments of type T to output of type T. Instead of viewing this as a function type, we can also interpret this as a logical implication. This works if T is a *proposition*, like the ones mentioned before. In the previous paragraph, we gave a proof/algorithm that $\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2$. Due to the Curry-Howard correspondence, we can interpret this proof as being a term p with type $p : \forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2$. The identity function on this proposition then has type

$$\text{id} : \left(\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2\right) \to \left(\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2\right).$$

Using the Curry-Howard correspondence in the opposite direction, we can see id as a proof that

$$\left(\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2\right) \implies \left(\forall n \in \mathbb{N}, \sum_{i=0}^{n} i = n(n+1)/2\right).$$

This correspondence stands at the base of proving theorems in Coq. Proving a theorem `P : Prop` means providing a witness `p : P`. Verifying the validity of the theorem is then the same as verifying the type of the witness. The verification algorithm of Coq is thus a type-checking algorithm.

2.3. Intuitionism and Coq

Coq's logic is inherently constructive/intuitionistic. To prove a theorem, we need to provide an explicit witness of the type of the theorem. This provides some difficulty for the 'law of excluded middle'. This is an axiom in classical logic, stating that every proposition is either true or false. More formally, it states that for every proposition A we have:

$$A \vee \neg A$$

This useful statement is not provable using the standard axioms of Coq. To see why, let us first consider what it means for a proposition to be false in Coq.

Coq defines False as an inductive type without terms: an empty type. We can always perform case analysis on elements of inductive types. For example, if we have $n : \mathbb{N}$, then we know that either $n = 0$ or $n = m + 1$ for a different $m : \mathbb{N}$. We can also perform case analysis for empty types, but for these types there are no cases to consider. A proposition `P : Prop` is considered to be false if we have a term `notP : P → False`. This means we have a function which sends a proof of `P` to an element of `False`. Such an element of `False` should not exist, since it is defined to have no terms. Therefore, a proof of `P` should not exist, so the proposition `P` is false.

The law of excluded middle thus requires us to either provide a term of type P, or a function which sends such a type to an element of False. We cannot do this for general P: we have no way to start building either of these terms. Likewise, we cannot prove double negation elimination. This is also an axiom of classical logic, stating that if the negation of a proposition is false, the proposition is true.

The law of excluded middle is compatible with Coq, and it can be added as an axiom. We will not do so in this chapter, since all results we need to prove are able to be proven constructively. A law of excluded middle is usually not needed for every proposition, but only for specific propositions like real inequalities.

There is another part where intuitionism pops up. We would like to perform case analysis on some propositions, like for example $A \vee B$. A proof of $A \vee B$ can either be a proof of A, or a proof of B. Coq allows us to perform this case analysis *only* when proving another proposition. It is not legal to perform case analysis on a proof of a proposition when defining something with a higher hierarchical type. This makes it more involved to define a function like

$$f(x) = \begin{cases} 1 & \text{x} > 0 \\ 0 & \text{x} \leq 0 \end{cases}$$

where we want to use that $x \leq 0 \vee x > 0$ for all $x \in \mathbb{R}$. This is still possible, since real inequalities are assumed to be decidable — we are assumed to be able to distinguish between real numbers.

2.4. Syntax of Coq

We are now almost ready to start using Coq to prove things. Three languages are involved when interacting with Coq, two of which are relevant to us now. The first one is called Gallina, which is the language of types and terms. Gallina is the language which implements CiC.

The second language is the Vernacular language. This is the top-level language of Coq. It is used to interface with Coq in various ways, such as requesting definitions or importing proofs from files. It also helps in creating proofs. Although it is possible to explicitly give a (Gallina) proof term when proving a theorem, this gets very complex when proving large theorems. If we enter 'proof mode', various commands will become available to prove a theorem in steps.

These two languages will be discussed in the following sections.

2.4.1. Gallina

As Gallina is the implementation language of the CiC, it is similar to the λ-calculus we discussed before. In this λ-calculus we used the notation $(f\ x)$ for applying the function f to the argument x, as opposed to the more conventional notation $f(x)$. Gallina also employs this $(f\ x)$ notation, and this is important to keep in mind when reading Gallina code. One of the benefits of this notation is that it is easier to use for functions with multiple arguments.

Suppose we have a function $F : \mathbb{R}^2 \to \mathbb{R}$. We can apply this function only to a pair of real numbers: $f(a, b) : \mathbb{R}$. It is often convenient to be able to access the function where we only vary the second coordinate and keep the first coordinate constant. To do so, we could interpret $\mathbb{R}^2 \to \mathbb{R}$ as $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, i.e. F is a function sending a real number to a function of type $\mathbb{R} \to \mathbb{R}$. Thus, for $a : \mathbb{R}$, we have $f(a) : \mathbb{R} \to \mathbb{R}$. If we apply this last function to another value b, we would need to write $f(a)(b)$, and the situation gets worse for higher values of n in $\mathbb{R}^n$. The $(f\ a\ b)$ notation saves parentheses for these cases.

Case analysis will often be used in proofs and functions, especially on inductively defined types. Gallina supports this with a match expression. Consider the following definition of a predecessor function, for the natural numbers we defined in Section 2.1.

```
Definition pred (n : nat) : nat :=
  match n with
  | zero ⇒ zero
  | succ m ⇒ m
  end.
```

The notation pred (n : nat) : nat means that the function pred, when applied to some n : nat, will have type nat. This means that pred : nat → nat. The := signifies the start of the actual definition of pred.

The current definition of pred sends a natural number to its predecessor. There is no predecessor for zero, so it maps zero to itself. The match expression is only considered valid if all cases of the inductive type are covered, so we have to specify what do to with zero. If we do not specify what to do with zero, the expression will not compile.

Another important keyword is fun (short for function), which plays the role of λ in the λ-calculus. We could have also defined pred as follows:

```
Definition pred' : nat → nat :=
  (fun n ⇒
    match n with
    | zero ⇒ zero
    | succ m ⇒ m
    end).
```

Specifying variables before the colon, like in the definition of pred, is shorthand notation for the function described by pred'.

For inductive types with just two terms, we have an if then else expression. To explain this, let us first define the boolean type.

```
Inductive boolean :=
  | true
  | false.
```

The boolean type has exactly two members: true and false. This means we can use an if then else expression on a term of type boolean. The first clause matches the first term in the definition of the type, likewise for the second. We could thus define some elementary boolean operations as follows:

```
Definition or (b1 b2 : boolean) : boolean :=
  if b1 then true else (if b2 then true else false).
Definition nand (b1 b2 : boolean) : boolean :=
  if b1 then
    if b2 then false else true
  else true.
```

This definition is more intuitive than the equivalent match expression. Note that if we would have defined the boolean type as:

```
Inductive boolean' :=
  | false'
  | true'.
```

the if then else expression used above would have done the exact opposite of what we expected.

The final keyword which shall often appear is let. It is a way to create a local definition in a term. This is useful when some subterm is repeated, or to improve the readability. It is quite helpful in the following example, where we build or from nand:

```
Definition or_from_nand (b1 b2 : boolean) : boolean :=
  let notb1 := nand b1 b1 in
  let notb2 := nand b2 b2 in
  nand notb1 notb2.
```

By naming the intermediate results, the definition is easier to read.

2.4.2. Vernacular

As discussed before, we talk with Coq using the Vernacular language. As a rule, every command starting with a capital letter and ending with a period is a Vernacular command. We have seen a few before, such as Inductive and Definition. Both are used to store a term or type with a given name, so that we can reuse it later.

To showcase Coq's proof mode, we will prove that our definition of nat satisfies Peano's eighth axiom. Peano's eighth axiom states that zero is not the successor of any other natural number. We write this in Coq in the following way:

```
Proposition Peano_ax8 (n : nat) : succ n ≠ zero.
```

This means that Peano_ax8 is a function sending an n : nat to the proof of succ n ≠ zero. We have not yet given the proof term, and we do not want to construct an explicit proof. We will therefore enter Coq's proof mode to help us construct a proof.

```
Proof.
```

```
1 subgoal

  n : nat
  ============================
  succ n <> zero
```

Once we enter proof mode, we have a proof context and (a number of) proof goals. The proof context contains all our hypotheses and variables. In this case, it contains the variable `n : nat`. The proof goals all have a specific type, and the proof can be finished with '`Qed.`' once we have constructed a term for each of the proof goal-types. In this case, the proof goal is `succ n ≠ zero`. It turns out we can prove this by entering

```
discriminate.
Qed.
```

`discriminate` is something called a tactic. Tactics are commands which are used to manipulate the proof context and to construct proofs. In this case, `discriminate` automatically proves the goal, but how `discriminate` does this is not clear. We will explain how `discriminate` proves this goal in Section 2.9. We will also discuss different useful tactics later.

Other useful vernacular commands are `Eval`, `Print`, `About` and `Check`. `Eval` can be used to perform computations in Coq:

```
Coq < Eval compute in 1 + 1.
     = 2
     : nat
```

`Check` supplies us with various information on a given term. It tells us that `Rle_0_1` contains a proof that $0 \leq 1$:

```
Coq < Check Rle_0_1.
Rle_0_1
     : 0 <= 1
```

`About` gives more information:

```
Coq < About Rle_0_1.
Rle_0_1 : 0 <= 1
Rle_0_1 is not universe polymorphic
Rle_0_1 is opaque
Expands to: Constant Coq.Reals.RIneq.Rle_0_1
```

`Print` also gives us more, but different information: it prints the explicit term contained by `Rle_0_1`:

```
Coq < Print Rle_0_1.
Fetching opaque proofs from disk for Coq.Reals.RIneq
Rle_0_1 = or_introl Rlt_0_1
     : 0 <= 1
```

The `or_introl` means that $0 \leq 1$, which is notation for $0 < 1 \vee 0 = 1$, has been proven by showing that $0 < 1$. A proof of $0 < 1$ can apparently be found in `Rlt_0_1`.

2.5. First proofs

We are now ready to prove some simple theorems in Coq. In the text below, some of Coq's responses will also be printed. This is to make the reader familiar with the effects that different tactics have on the Coq environment.

```
Proposition or_true_l : ∀ (b2 : boolean), or true b2 = true.
Proof.
```

We have entered proof mode. Our goal is something of type ∀. To prove it, we would like to move the quantifier variables to the local context. This is what the `intros` tactic does.

```
intros.
```

```
1 subgoal

  b2 : boolean
  ============================
  or true b2 = true
```

Let us recall the definition of `or`. We can do this by using the Vernacular `Print` command as follows:
`Print or.`

```
or =
fun b1 b2 : boolean => if b1 then true else if b2 then true else false
     : boolean -> boolean -> boolean
```

It seems that when we substitute `true` for `b1` in the `or` function, we should immediately get `true`. To perform this computation, we do:

```
compute.
```

```
1 subgoal

  b2 : boolean
  ============================
  true = true
```

This should be easy to prove, since equality is reflexive. Indeed, we can finish the proof with:

```
reflexivity.
```

```
No more subgoals.
```

This means we are done, so we can conclude with:

```
Qed.
```

This was still quite easy. We would also like to prove the following, which turns out to be a bit harder. We start with the same approach:

```
Proposition or_true_r : ∀ (b1 : boolean), or b1 true = true.
Proof.
intros.
compute.
```

```
1 subgoal

  b1 : boolean
  ============================
  (if b1 then true else true) = true
```

`compute` unfolds the definition of `or`, but cannot unfold it further than this. This is because the `if` expression cannot be evaluated for a general `boolean`, even though the `if` and `else` parts are equal. To distinguish the cases for `b1`, we use the `destruct` tactic.

```
destruct b1.
```

```
2 subgoals

  ============================
  true = true
subgoal 2 is:
 true = true
```

We now have two goals, corresponding to the two existing boolean terms: `true` and `false`. Both goals can be finished with the `reflexivity` tactic. To apply this tactic to all goals, we can use the `all`: prefix. This means we can finish as follows:

```
all: reflexivity.
Qed.
```

We can apply all we've learned so far when proving the following proposition.

```
Proposition or_eq_from_nand : ∀ (b1 b2: boolean), or b1 b2 = or_from_nand b1 b2.
Proof.
intros.
```

```
1 subgoal

  b1, b2 : boolean
  ============================
  or b1 b2 = or_from_nand b1 b2
```

After running `intros`, both variables `b1` and `b2` have moved to our local context. Our goal is now an equality of `or` and `or_from_nand`. These computations can only be performed on the explicit booleans `true` and `false`. To distinguish the different cases, we again use `destruct`.

```
destruct b1; destruct b2.
```

```
4 subgoals

  ============================
  or true true = or_from_nand true true
subgoal 2 is:
 or true false = or_from_nand true false
subgoal 3 is:
 or false true = or_from_nand false true
subgoal 4 is:
 or false false = or_from_nand false false
```

The semicolon indicates that we perform both `destruct`s in succession. Once the `destruct`s have been performed, we have four goals, corresponding to the four possible options for our two booleans. We can now evaluate the operations `or` and `or_from_nand`.

```
all: compute.
```

```
4 subgoals

  ============================
  true = true
subgoal 2 is:
 true = true
subgoal 3 is:
 true = true
subgoal 4 is:
 false = false
```

As before, we can conclude with `reflexivity`:

```
all: reflexivity.
Qed.
```

To recap, we saw the following tactics:

- `intros`, to move hypotheses or variables from the goal to the local context
- `destruct`, to perform case analysis on a term
- `compute`, to try to simplify the goal by expanding definitions
- `reflexivity`, to solve goals of type `x = x`.

We will now move to more advanced proofs on the natural numbers.

2.6. Recursion and induction

Let us continue developing the natural numbers we defined earlier. The next step would be to define a way to add two natural numbers. We would like to follow Peano's definition, which defines addition recursively for a b : nat as:

$$a + 0 = a,$$
$$a + S(b) = S(a + b).$$

We define this in Coq as:

```
Fixpoint plus (n m : nat) :=
  match m with
  | zero ⇒ n
  | succ m0 ⇒ succ (plus n m0)
  end.
```

By using Fixpoint instead of Definition, we indicate to Coq that the following definition will be recursive. This allows us to use plus inside the definition of plus itself. Coq does not accept just any definition. It detects that in this case, the syntactic size of the m term decreases, so the recursion will terminate.
We can now check that $1 + 1 = 2$, as follows:

```
Eval compute in plus (succ zero) (succ zero).

     = succ (succ zero)
     : nat

Proposition plus_0_r (n : nat) : plus n zero = n.
Proof.
reflexivity.
Qed.
```

The proof above is easy; the definition of plus can be unfolded and directly evaluates to n. We do not need to call the compute tactic, since reflexivity calls compute itself. This means we can directly conclude with reflexivity. The proof gets harder when we swap the arguments to plus.

```
Proposition plus_0_l (n : nat) : plus zero n = n.
Proof.
destruct n.

2 subgoals

  ============================
  plus zero zero = zero
subgoal 2 is:
 plus zero (succ n) = succ n
```

So far, so good. The first case can easily be solved once again with reflexivity. The second case can be unfolded once. In this case, compute will try to unfold plus too aggressively, so we use cbn. The tactic cbn unfolds definitions only when the resulting term is 'simpler'.

```
reflexivity.
cbn.

1 subgoal

  n : nat
  ============================
  succ (plus zero n) = succ n
```

This is where we get stuck. We wanted to prove plus zero n = n, and we need such a proof to conclude now. We need induction to prove this proposition. We will try again.

```
1 subgoal

  n : nat
  ============================
  plus zero n = n
```

We have reset our progress. To use induction on `n`, we use the `elim` tactic.

```
elim n.
```

```
2 subgoals

  n : nat
  ============================
  plus zero zero = zero
subgoal 2 is:
 forall n0 : nat, plus zero n0 = n0 -> plus zero (succ n0) = succ n0
```

The subgoals generated correspond to what we expect from a proof by induction. The first goal can again be immediately solved by `reflexivity`. For the second goal, we use `intros` to move the variables and hypotheses to the local context, and `cbn` to simplify the term.

```
reflexivity.
intros.
cbn.
```

```
1 subgoal

  n, n0 : nat
  H : plus zero n0 = n0
  ============================
  succ (plus zero n0) = succ n0
```

Compare this context with our previous attempt. We now have the induction variable `n0`, and the additional hypothesis `H` : `plus zero n0` = `n0`. By the Curry-Howard correspondence, we can interpret this as `H` being a proof of `plus zero n0` = `n0`. Coq has automatically generated a name for our hypothesis, we could have (and will subsequently) always name our hypotheses explicitly. This can be done by supplying intros with the names of the variables to be introduced.
Note that the left-hand side, `plus zero n0`, appears in our goal. We can use this proof of equality to perform a *rewrite* using the `rewrite` tactic:

```
rewrite H.
```

```
1 subgoal

  n, n0 : nat
  H : plus zero n0 = n0
  ============================
  succ n0 = succ n0
```

As we can see, the right hand side of `H` has been substituted in our goal. We are left with an equality we can prove with `reflexivity`.

```
reflexivity.
Qed.
```

We aim to prove eventually that `plus` commutes. This is easier to do once we have proved the following proposition:

```
Theorem plus_succ_l (n m : nat) : plus (succ n) m = succ (plus n m).
Proof.
```

In this case, we have two variables of type `nat`. As before, we will need induction to prove this. We do need to pick the right variable over which to perform induction! In this case, we need to do induction

over m. This is because we can unfold plus if we do case analysis on m. So, we proceed with the elim tactic.

```
elim m.

2 subgoals

  n, m : nat
  ============================
  plus (succ n) zero = succ (plus n zero)
subgoal 2 is:
 forall n0 : nat,
 plus (succ n) n0 = succ (plus n n0) ->
 plus (succ n) (succ n0) = succ (plus n (succ n0))
```

Note that for the first goal, we have two instances of a term of the form plus _ zero. We have proved an equality for this term in plus_0_r. After rewrites, the goal will have the form succ n = succ n. This is again easy to prove with reflexivity. This type of goal is easy enough for Coq to figure out how to prove it by itself. To tell Coq it should try to prove the goal itself, we prefix our tactic with now. now will only succeed if it can completely solve the goal, otherwise it will give an error. Remember that a semicolon indicates that the two tactics should be performed in succession. The now tactic will run after both tactics have been performed.

```
now rewrite plus_0_r; rewrite plus_0_r.
intros ? H.

1 subgoal

  n, m, n0 : nat
  H : plus (succ n) n0 = succ (plus n n0)
  ============================
  plus (succ n) (succ n0) = succ (plus n (succ n0))
```

The question mark indicates that we let Coq decide the name of the induction variable, but we explicitly name the hypothesis H. We simplify this goal once again with cbn. Note that it performs two simplifications, one on each side of the equals sign.

```
cbn.

1 subgoal

  n, m, n0 : nat
  H : plus (succ n) n0 = succ (plus n n0)
  ============================
  succ (plus (succ n) n0) = succ (succ (plus n n0))
```

Again, the left-hand side of H appears in our goal. After rewriting, we are left to prove a term of the form x = x. Coq will solve this for us when we prepend now to our rewrite:

```
now rewrite H.
Qed.
```

We are now ready to prove that plus defines a commutative operation. We will use all of our previously proven propositions.

```
Proposition plus_comm (n m : nat) : plus n m = plus m n.
Proof.
```

As before, we try to prove this with induction on m. In this case we could also have done this by doing induction on n, since the statement is symmetric with respect to n and m.

```
elim m.

2 subgoals

  n, m : nat
  ============================
  plus n zero = plus zero n
subgoal 2 is:
 forall n0 : nat,
 plus n n0 = plus n0 n -> plus n (succ n0) = plus (succ n0) n
```

The first goal can immediately be proven by rewriting with `plus_0_l` and `plus_0_r` and reflexivity. For the second goal, we again use `intros` and `cbn` to simplify.

```
now rewrite plus_0_l; rewrite plus_0_r.
intros ? H.
cbn.

1 subgoal

  n, m, n0 : nat
  H : plus n n0 = plus n0 n
  ============================
  succ (plus n n0) = plus (succ n0) n
```

We can now use `plus_succ_l` to rewrite the righthand side. We then obtain:

```
rewrite plus_succ_l.

1 subgoal

  n, m, n0 : nat
  H : plus n n0 = plus n0 n
  ============================
  succ (plus n n0) = succ (plus n0 n)
```

We conclude by rewriting with our induction hypothesis `H`.

```
now rewrite H.
Qed.
```

Done! We have seen some new tactics, and were able to prove that `plus` commutes using induction. But where did this induction principle come from? The induction principle is a term, just like anything else in Coq. For the natural numbers, it is called `nat_ind`, and was automatically generated when `nat` was defined. This happens for all inductive types, and we can actually expand it as an explicit lambda term. We will just look at the type of it, which should look familiar:

```
Check nat_ind.

nat_ind
     : forall P : nat -> Prop,
       P zero ->
       (forall n : nat, P n -> P (succ n)) -> forall n : nat, P n
```

Let us recount the tactics we saw and what they did:

- `intros`, has been used before, but one can also supply it with explicit names for variables and hypotheses;
- `cbn`, performs computation like `compute`, but only if it simplifies the term;
- `rewrite`, uses given equality to rewrite the goal;
- `now`, used as a prefix to another tactic to let Coq try to solve the goal automatically.

2.7. Propositional logic

We have previously discussed the `boolean` type, with elements true and false. There are also the propositions `True` and `False`. These are different and should not be confused with one another. The `boolean` type is meant for boolean arithmetic, so has no relation to what we can prove or disprove. `True` and `False`, however, are propositions — just like the equalities we proved on the natural numbers. We will investigate how they are defined.

```
Print True.
```

```
Inductive True : Prop :=  I : True
```

This definition means that there is only one element of type `True`, which is `I`. `I` can be interpreted as the 'proof of' `True`. It comes in handy when looking at the conjunction of multiple propositions, since it acts as a zero element for conjunction.

```
Print False.
```

```
Inductive False : Prop :=
```

The definition of `False` can be somewhat confusing. It states that *there are no elements* of type `False`; `False` is an empty type. This means that if we can get an element of `False`, we have reached a contradiction. This does not mean that Coq stops working or throws an error. Contradictory statements are fine, but we do get the classic result of *ex falso quod libet*: from falsehood we can derive anything. Indeed, we can prove this:

```
Proposition exfalso (P : Prop) : False → P.
Proof.
intros falsity.
```

```
1 subgoal

  P : Prop
  falsity : False
  ============================
  P
```

How to proceed? Since `False` is an inductive type, we can use the `destruct` tactic to perform a case analysis. Let us see what happens.

```
destruct falsity.
```

```
No more subgoals.
```

This means we are done. How did that happen? A case analysis usually requires us to consider the different cases and then prove `P` in each case. However, *there are no cases to consider* — and by considering no cases we have considered all the cases.

```
Qed.
```

The `False` type makes us able to define what it means for a proposition to be *not* true. A proposition `A` is not true (notation ¬`A` or ∼`A`) if there is a function sending a proof of `A` to an element of `False`.

```
Print not.
```

```
not = fun A : Prop => A -> False
     : Prop -> Prop
```

Since `False` is an empty type, elements of `False` should not exist. If we have a function sending proofs of `A` to elements of `False`, this means proofs of `A` should not exist: `A` is not provable/true.

In addition to negation, we also have a notion of conjunction and disjunction (*and* and *or*) for propositions.

```
Print and.
```

```
Inductive and (A B : Prop) : Prop :=  conj : A -> B -> A /\ B
```

This defines `and` as an inductive type with one element: `conj`. When given a proof `a : A` and a proof `b : B`, we get that `conj a b` has type `and A B`. By `destruct`ing an element of type `and A B`, we can recover proofs of `A` and of `B`. Here `A ∧ B` is notation for `and A B`.

Note that the type of `conj` is `A → B → A ∧ B`. The → symbol will always be right-associative in Coq, so `A → B → A ∧ B` should be read as `A → (B → A ∧ B)`. This means `conj` is a function sending elements of type `A` to functions sending elements of type `B` to elements of type `A ∧ B`. One can also interpret this as a logical implication: for propositions A and B, we have $A \implies (B \implies A \wedge B)$. This is another good example of the Curry-Howard correspondence.

```
Print or.
```

```
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

The inductive type for `or` has two elements, corresponding to two ways to construct an element of type `or A B`. This matches our expectations; we should be able to prove $A \vee B$ by either proving A or proving B. Once again `A ∨ B` is notation for `or A B`.

As before, the types of the elements of `or` can be read as logical implications. In this interpretation, `or_introl` says that $A \implies (A \vee B)$, while `or_intror` says that $B \implies (A \vee B)$.

Let us try to prove `A ∧ B` from a proof that `A` holds and a proof that `B` holds.

```
Proposition both_implies_and (A B : Prop) : A → B → A ∧ B.
Proof.
intros proofA proofB.
split.
```

```
2 subgoals

  A, B : Prop
  proofA : A
  proofB : B
  ============================
  A
subgoal 2 is:
 B
```

```
all: assumption.
Qed.
```

There are two new tactics here: `split` and `assumption`. The tactic `split` transforms any goal whose type can only be constructed by providing multiple terms, into subgoals for each of these terms. Before `split`, the goal was `A ∧ B`. There is only one way to construct an element of type `and A B`: using the `conj` function, which requires an element of type `A` and an element of type `B`. This means that `split` transforms our previous goal of type `A ∧ B` to two new goals: one of type `A` and one of type `B`.

We have a proof of `A` and a proof of `B` in our proof context. The `assumption` tactic can be used when there is an assumption in the local context whose type matches the goal. If that is the case, `assumption` will automatically solve the goal.

Now compare the type of `both_implies_and` to the type of `conj`. They are the same! We have not shown anything new here, we have just created a new name for `conj`. We will now prove something new.

```
Proposition and_implies_or (A B : Prop) : A ∧ B → A ∨ B.
Proof.
intros H.
destruct H.
```

```
1 subgoal

  A, B : Prop
  H : A
  H0 : B
  ============================
  A \/ B
```

We will now use a new tactic to proceed: `left`.

```
left.
```

```
1 subgoal

  A, B : Prop
  H : A
  H0 : B
  ============================
  A
```

```
assumption.
Qed.
```

The `left` tactic transforms any goal whose type which can be constructed in exactly two ways, to the conditions for the first construction. The first way to construct `A ∨ B` was with `or_introl`. This requires an element of type `A`, so `left` transforms the goal `A ∨ B` to `A`. Likewise, there is the `right` tactic.

```
Proposition or_not_right_then_left (A B : Prop) : A ∨ B → ¬B → A.
Proof.
intros H1 H2.
destruct H1.
assumption.
```

```
1 subgoal

  A, B : Prop
  H : B
  H2 : ~ B
  ============================
  A
```

Note that `assumption` has solved the first subgoal generated by `destruct`. For the second subgoal, proving `A` directly is not possible. However, the assumptions we have seem to contradict each other, since we have both `B` and `¬B`. Remember that we can prove anything from `False`; we proved this in `exfalso`, and would like to apply it here. We can do this using the `apply` tactic. It is used to perform backwards reasoning. If we have a term whose conclusion matches our proof goal, it suffices to prove the hypotheses of this term. `A` is a proposition, and the conclusion of `exfalso` is any proposition, so `exfalso` can indeed be applied.

```
apply exfalso.
```

```
1 subgoal

  A, B : Prop
  H : B
  H2 : ~ B
  ============================
  False
```

Since `¬ B` is notation for `B → False`, we can again apply `H2`. This will transform the goal to type `B`, and we already have an element of type `B`! We can then use `assumption` to conclude.

```
apply H2.
```

```
1 subgoal

  A, B : Prop
  H : B
  H2 : ~ B
  ============================
  B
```

```
assumption.
Qed.
```

In regular proofs, we usually stop when we reach a contradiction. In Coq, this is not the case. Even if we reach a contradiction, we need to prove our original goal, and we can do so using `exfalso`.

Using `exfalso` and then contradicting hypotheses to prove a goal occurs with some frequency, which is why there is a tactic to help us. This tactic is called `contradiction` and it proves any goal from contradictory hypotheses. We use it when proving the following theorem.

```
Proposition or_not_left_then_right (A B : Prop) : A ∨ B → ¬A → B.
Proof.
intros H1 H2.
destruct H1.
- contradiction.
- assumption.
Qed.
```

Note the use of a hyphen to group parts of a proof. This is useful when a tactic generates multiple subgoals, and it might be unclear which tactics apply to which goal. In this case `contradiction` solves the first subgoal, while `assumption` solves the second.

Let us recap the new tactics we saw:

- `left` and `right`, for proving types which can be constructed in exactly two ways;
- `assumption`, for proving a goal when we have an assumption of the correct type;
- `apply`, to prove a goal by proving the hypothesis for another term, whose conclusion matches the goal;
- `split`, to prove a goal which can only be constructed by providing two or more terms;
- `contradiction`, to prove any goal from contradictory assumptions.

2.8. Lists

It is often useful to be able to represent zero or multiple elements of a type in one structure: a list. Lists of numbers can be summed, and we can define the conjunction or disjunction of a list of propositions. We could define a list type for each type where this makes sense, but Coq allows us to make a parametric type. A parametric type can be seen as a function of type `Type` → `Type`; it maps a `Type` to another `Type`. This is useful, since it enables us to prove things on lists of unspecified type. This means we can reuse all results on lists when working with a list of a specific type.

We can define the list type as follows:

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.
```

Here `nil` represents the empty list, and `cons` gives us a way to concatenate any element of type `A` to another list, creating a bigger list. The structure of `list` is in some way similar to that of the natural numbers, but we are able to store an element at every `cons`/`succ`.

To use `nil` and `cons`, we have to explicitly supply it with the type `A` when we use it. However, there is often just a single option, which can be inferred from the context. We can make the argument `A` implicit with the following Vernacular commands:

```
Arguments nil {A}.
Arguments cons {A}.
```

Now Coq will try to figure out which type to insert as argument. We will now start using our `list` structure. One intuitive operation is to append two `list`s, which creates a larger list. This would be defined as:

```
Fixpoint append {A : Type} (l1 l2 : list A) :=
  match l1 with
  | nil ⇒ l2
  | cons a l1 ⇒ cons a (append l1 l2)
  end.
```

Instead of writing `append` each time, we can create a notation. This can be defined as follows:

```
Notation "l1 +++ l2" := (append l1 l2) (at level 50).
```

The `level` 50 part is about the binding strength of this notation. Like +++, the equality symbol = is also just notation. The binding strength determines the order of operations of each notation. If we would have put the binding strength higher than equality, it will not behave the way we want it to.
Let us prove some propositions about this appending of `list`s. Note that `append nil l` will directly evaluate to `l`, just like adding zero to the right. More interesting is the swapped example:

```
Proposition append_nil {A : Type} (l : list A) : l +++ nil = l.
Proof.
elim l.

2 subgoals

  A : Type
  l : list A
  ============================
  nil +++ nil = nil
subgoal 2 is:
 forall (a : A) (l0 : list A),
 l0 +++ nil = l0 -> cons a l0 +++ nil = cons a l0
```

As with the natural numbers, we prove this by induction. The first goal can be proven by `reflexivity`. For the second goal, note that `cons a l0` +++ `nil` stands for `append` (`cons a l0`) `nil`, so that we can unfold `append` with `cbn` once. Then we can finish the proof by rewriting with the provided equality.

```
reflexivity.
intros ? ? H; cbn.
now rewrite H.
Qed.
```

Appending two lists is not a commutative operation. It is, however, associative, and we will prove this.

```
Proposition append_assoc {A : Type} (l1 l2 l3 : list A) :
    (l1 +++ l2) +++ l3 = l1 +++ (l2 +++ l3).
Proof.
elim l1.

2 subgoals

  A : Type
  l1, l2, l3 : list A
  ============================
  (nil +++ l2) +++ l3 = nil +++ (l2 +++ l3)
subgoal 2 is:
 forall (a : A) (l : list A),
 (l +++ l2) +++ l3 = l +++ (l2 +++ l3) ->
 (cons a l +++ l2) +++ l3 = cons a l +++ (l2 +++ l3)
```

Once again we need to pick over which variable we perform induction. This turns out to be `l1`. The first case is true by computation. The second case can be solved by unfolding `append` once using the `cbn` tactic. The goal can then immediately be solved once it has been rewritten with the induction hypothesis `H`.

```
reflexivity.
intros ? ? H; cbn.
now rewrite H.
Qed.
```

Other intuitive operations on lists are looking at the head and the tail of the list. The head of a list is the first element, so something of type `A`. The tail of a list is everything but the head, so this is another list of type `A`.

However, these operations only make sense for lists that are not equal to `nil`. To get around this, we provide the head operation with a default element, which it should return if the list is equal to `nil`.

```
Definition head {A : Type} (l : list A) (default : A) :=
  match l with
  | nil ⇒ default
  | cons a _ ⇒ a
  end.
```

For the tail operation we can just return `nil` when `nil` is supplied.

```
Definition tail {A : Type} (l : list A) :=
  match l with
  | nil ⇒ nil
  | cons _ l0 ⇒ l0
  end.
```

We will show that every list that is not equal to `nil`, is equal to the concatenation of its `head` and its `tail`. This is intuitively clear, but we should be able to prove it as well.

```
Proposition cons_head_tail_eq {A : Type} (l : list A) (default : A) :
  l ≠ nil → l = cons (head l default) (tail l).
Proof.
intros H.
destruct l.
```

```
2 subgoals

  A : Type
  default : A
  H : nil <> nil
  ============================
  nil = cons (head nil default) (tail nil)
subgoal 2 is:
 cons a l = cons (head (cons a l) default) (tail (cons a l))
```

The goal seems to be impossible to prove. However, our hypothesis `H` itself should not be possible. The `destruct` tactic forces us to consider all cases, but there would be a *contradiction* if it really were the first case, since obviously `nil` = `nil`. Since `nil` ≠ `nil` is notation for ¬ (`nil` = `nil`), this hypothesis is indeed contradictory and we can conclude with `contradiction`.

```
contradiction.
```

```
1 subgoal

  A : Type
  a : A
  l : list A
  default : A
  H : cons a l <> nil
  ============================
  cons a l = cons (head (cons a l) default) (tail (cons a l))
```

The remaining subgoal is easy, since once `head` and `tail` have been expanded, the terms on the left and right hand side match exactly. We therefore finish with `reflexivity`.

```
reflexivity.
Qed.
```

Note that to apply `cons_head_tail_eq`, we need to provide a default element. This is undesirable; it is never really used, we only need it to be able to use `head`, since it requires a default element. However, Coq enables us to work around this. The above procedure of mapping a list to its head or tail can be done in most programming languages. What we will do next can usually not be done: we will map a list to the head element, when we are provided with a proof that the list is not empty.

```
Definition head' {A : Type} (l : list A) : (l ≠ nil) → A.
Proof.
```

The commands above might look a bit weird, since we are used to giving an explicit definition with :=. However, we can enter proof mode just like for `Proposition`, and use tactics to construct a term. The only difference is that our 'proof goal' is now not a `Prop`, but some `Type`.

```
intros not_nil.

1 subgoal

  A : Type
  l : list A
  not_nil : l <> nil
  ============================
  A
```

We need to provide an element of type `A`. As before, we will use `destruct` to perform case analysis.

```
destruct l.

2 subgoals

  A : Type
  not_nil : nil <> nil
  ============================
  A
subgoal 2 is:
 A
```

As in `cons_head_tail_eq`, we have no way to produce an element of type `A`, but we do have a contradictory assumption. This means we conclude with `contradiction`.

```
contradiction.

1 subgoal

  A : Type
  a : A
  l : list A
  not_nil : cons a l <> nil
  ============================
  A
```

For the second goal, we do have an element of type `A` available: `a`. To solve the goal, we can use the `exact` tactic. This tactic solves the goal when provided with an element of the correct type.

```
exact a.
Defined.
```

Note that we finish with '`Defined`.' instead of '`Qed`.'. By using '`Defined`.', the definition of `head'` is transparent, or unfoldable. '`Qed`.' causes the definition to be opaque, so that we cannot actually perform the `head'` computation

We can now redo `cons_head_tail_eq`, with our new definition `head'`. This time, we do not need a default element of type A; we input the proof that the list is not `nil` directly into the `head'` function. After doing case analysis on `l`, we can solve the first goal with `contradiction`, and the second with `reflexivity`.

```
Proposition cons_head'_tail_eq {A : Type} (l : list A) (p : l ≠ nil) :
  l = cons (head' l p) (tail l).
Proof.
destruct l.
contradiction.
reflexivity.
Qed.
```

Let us recap once again the new tactics we saw:

- `exact`, to solve a goal with an assumption that matches the type of the goal.

2.9. Equality in Coq

Equality is represented by a type in Coq. It does not have any special role in the language, but it satisfies our notions of equality by construction.

The equality type is defined as follows:

```
Inductive eq {A : Type} (x : A) : A → Prop :=
  eq_refl : eq x x.
```

Let us take a second to figure out what this means. Like with lists, eq defines a parametric type. However, eq takes not just a type A, but an element of this type x, and maps it to something of type A → Prop. The curly brackets around A : Type indicate that the correct type is implicitly substituted by default. It then states that the only way to create an element of this Prop, is by using the constructor eq_refl, of type eq x x. So initially we just have $x = x$. To see how we get different equalities, consider the induction principle of equal:

```
Check eq_ind.

eq_ind
     : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, eq x y -> P y
```

This type signature is exactly what we want for an equality: being able to subtitute equal terms in propositions and functions.

Let us see how equal_ind is used in some simple proofs. Remember that tactics are just tools to help construct the correct terms. We can use tactics to construct a proof, and inspect the constructed lambda term later. We will now prove that equality is symmetric.

```
Proposition eq_sym {A : Type} (x y : A) : y = x → x = y.
Proof.
intros H.
now rewrite H.
Qed.
```

Proving this is quite straightforward. After rewriting, the goal becomes y = y, which is trivial. We can now investigate the exact proof term of eq_sym.

```
Print eq_sym.

eq_sym =
fun (A : Type) (x y : A) (H : y = x) =>
eq_ind_r (fun y0 : A => x = y0) eq_refl H
     : forall (A : Type) (x y : A), y = x -> x = y
```

eq_ind_r is used, which is a close neighbour of eq_ind. It has type:

```
About eq_ind_r.

eq_ind_r :
forall (A : Type) (x : A) (P : A -> Prop),
P x -> forall y : A, y = x -> P y
```

Note that x = y has been swapped with y = x in eq_ind_r. It is a good exercise to try to see how the proof of eq_sym works. The fun y0 ⇒ x = y0 term has type A → Prop, and is named P in eq_ind_r. P x evaluates to x = x, of which eq_refl is a proof. Since H is a proof that y = x, eq_ind_r gives us a proof of P y when given these arguments. P y will evaluate to x = y, as required.

We now know enough about equality in Coq to return to Peano's eighth axiom. We proved it was true in Section 2.4.2, where the discriminate tactic solved it automatically. How discriminate did this was unclear, so we will now prove it without discriminate.

```
Proposition Peano_ax8' (n : nat) : succ n ≠ zero.
Proof.
intros eq_succn_zero.
```

```
1 subgoal

  n : nat
  eq_succn_zero : succ n = zero
  ============================
  False
```

Remember from before that a ≠ b was notation for a = b → False. We need to somehow create an element of False with our hypothesis eq_succn_zero.

eq_succn_zero would allow us to do a rewrite, if zero or succ n was in the goal. However, False does not contain any such term. To continue, we make it depend on such a term.

```
set (P := (fun n ⇒ match n with | zero ⇒ True | _ ⇒ False end)).
change False with (P (succ n)).
```

```
1 subgoal

  n : nat
  eq_succn_zero : succ n = zero
  P := fun n : nat => match n with
                      | zero => True
                      | succ _ => False
                      end : nat -> Prop
  ============================
  P (succ n)
```

P is a tricky function, specially crafted to allow us a rewrite. P sends zero to True, other natural numbers to False. This means that P (succ n) will reduce to False, so these terms are interchangeable. The change tactic then swaps these interchangeable terms.

We can now use eq_succn_zero to perform a rewrite.

```
rewrite eq_succn_zero; cbn.
```

```
1 subgoal

  n : nat
  eq_succn_zero : succ n = zero
  P := fun n : nat => match n with
                      | zero => True
                      | succ _ => False
                      end : nat -> Prop
  ============================
  True
```

After rewriting our goal becomes P zero. By definition of P, this reduces to True. P was defined in such a way that eq_succn_zero allowed us to rewrite False to True. To prove True, we can exact its only proof I.

```
exact I.
Qed.
```

Let us recap the new tactic we saw:

- change allows us to replace a term with another term, if these two terms reduce to the same term.

We will now make formal what the rules are for reducing terms.

2.10. Computation in Coq

Up until now we have seen two forms of substitution: by using rewrites, and by expanding/unfolding definitions using `cbn` or `compute`. The difference between them is important. If we `rewrite` the goal, the resulting proof term will contain an application of the induction principle of the equality type. This means `rewrite` is particular to the definition of equality, thus not fundamental to the Calculus of Inductive Constructions. The substitutions caused by `cbn` or `compute` *are* fundamental to the Calculus of Inductive Constructions, inherent to the language itself. We will discuss the different kinds of substitution rules of Gallina, and how one can do a specific kind of these substitutions in this section. This is sometimes useful when `cbn` or `compute` does not do exactly what you need. For example, you might want to expand some definitions, while keeping others untouched.

Gallina comes with four kinds of substitution rules, each corresponding to a greek character:

- `delta`, unfolds named definitions. It is also possible to supply delta with specific definitions to unfold;
- `beta`, substitutes terms in functions;
- `zeta`, substitutes local definitions, i.e. `let _ in _` expressions;
- `iota`, unfolds `match`, `if` and recursive definitions.

These substitution rules (or reduction strategies) can be supplied as arguments to the `cbv` tactic. `cbv` behaves like `compute` when given no arguments, but its behaviour can be controlled when given substitution rules as arguments. We will see how these apply to the `or_from_nand` function we defined a couple of sections earlier. We will first recall the definition:

```
Print or_from_nand.

or_from_nand =
fun b1 b2 : boolean =>
let notb1 := nand b1 b1 in let notb2 := nand b2 b2 in nand notb1 notb2
     : boolean -> boolean -> boolean
```

When applied to `true` and `false`, this should evaluate to `true`. This is what we get when we use `cbv` without arguments:

```
Eval cbv in (or_from_nand true false).

     = true
     : boolean
```

The first thing that happens when computing this, is unfolding the definition of `or_from_nand`. We can restrict `cbv` to do only that by running the following:

```
Eval cbv delta [or_from_nand] in (or_from_nand true false).

     = (fun b1 b2 : boolean =>
        let notb1 := nand b1 b1 in
        let notb2 := nand b2 b2 in nand notb1 notb2) true false
     : boolean
```

We can substitute the arguments in the function by specifying the `beta` reduction strategy:

```
Eval cbv delta [or_from_nand] beta in (or_from_nand true false).

     = let notb1 := nand true true in
       let notb2 := nand false false in nand notb1 notb2
     : boolean
```

To unfold this term, we need to first unfold the definition of `nand`. We therefore add `nand` to the definitions to be unfolded by `delta`. Since we have already specified the `beta` strategy, the variables will also be substituted in the `nand` function.

```
Eval cbv delta [or_from_nand nand] beta in (or_from_nand true false).

     = let notb1 := if true then if true then false else true else true
         in
       let notb2 :=
         if false then if false then false else true else true in
       if notb1 then if notb2 then false else true else true
     : boolean
```

The `if else` expressions now become somewhat confusing. Since `if` is shorthand for a `match` expression, we can reduce these expressions using the `iota` reduction strategy.

```
Eval cbv delta [or_from_nand nand] beta iota in (or_from_nand true false).

     = let notb1 := false in
       let notb2 := true in
       if notb1 then if notb2 then false else true else true
     : boolean
```

Note that the last `if` expression has not been reduced, since the variables over which to match are `notb1` and `notb2` and these are still abstract. We can substitute their values in the let expression by adding the `zeta` flag. Since we already have the `iota` flag, the expression will be reduced to the final result: `true`.

```
Eval cbv delta [or_from_nand nand] beta iota zeta in (or_from_nand true false).

     = true
     : boolean
```

2.11. More tactics

There are loads of additional tactics other than the ones we have seen so far. We will name a couple which have been found useful for the rest of the thesis.

- `enough` and `assert`. These are used to prove the goal using a new assumption, and then requiring you to prove this assumption. With `assert` you need to prove this assumption immediately, while `enough` lets you prove the goal using this assumption (i.e. you show this assumption is indeed *enough*), and only then makes you prove the assumption.
- `exists` is used to prove an existential goal. To prove $\exists x, Px$, you first need to find some witness `x'`. Then you use `exists x'`. The proof goal will then become Px'.
- `set` and `pose` are used to define local variables with explicit definitions. The `set` tactic replaces any occurence of this definition in the goal with the new name of the variable, `pose` leaves the goal untouched. This is useful for decreasing the size of the goal, to make it easier to manipulate.
- `unfold` will, when given names of definitions, *unfold* these definitions in the goal. This is basically short hand for `cbv delta [names]`.
- `fold` needs to be supplied a term. It will search for an unfolded version of this term, and replace it with the provided term.
- `replace` has the exact syntax as `change`, but does not require the two terms to expand/unfold to the exact same term. Instead, it adds a new goal to your proof: that the two provided terms are equal.
- `eapply` and `eexists` are used just like `apply` and `exists`, when it is convenient for the user to specify the exact terms later. In the mean time, the goal or term is unspecified.
- `refine` allows you to supply the (skeleton of the) exact term solving the goal, but leave several holes unfilled. Coq will then generate a new subgoal for each unfilled hole.

2.12. Real numbers

Real numbers are tricky to define formally — trickier then, for example, the natural numbers. We were able to capture all that a natural number should be in an inductive definition, but it is unfortunately

not possible to do this directly for the real numbers. One of the ways to construct real numbers is to use Cauchy sequences to make $\mathbb{Q}$ complete. A real number is then defined as the equivalence class of rational sequences converging to this number. It is possible to formalize the reals in this way, which is how it is done in the C-CoRN library [12].

The standard library of Coq takes a different approach. Instead of defining the real number type inductively, the type is defined by an axiom. The rules for computing and comparing real numbers are also defined by axioms. This might be improved in the future. We will use the reals from the standard library in this thesis.

The axioms for the real numbers state (in more words) that $\mathbb{R}$ is a complete totally ordered commutative field. Furthermore, comparison is decidable. This means that for any two real numbers r_1 and r_2 we have $r_1 < r_2$, $r_1 = r_2$ or $r_1 > r_2$. This is stated in a way that does not use the standard propositional $\vee$, so that we can define functions like

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0; \\ 0 & \text{otherwise.} \end{cases}$$

Finally, $\mathbb{R}$ is defined to have the Archimedean property. This excludes two real numbers from having incomparable size; i.e. it forces $\mathbb{R}$ to be the standard real numbers. This means there are no infinitesimal numbers nor infinitely large numbers. This is equivalent to saying that for every real number r, there is a natural number n larger than r when viewed as a real number, since this means infinitely large numbers cannot exist. Coq assumes this last statement as an axiom.

To access the real numbers from Coq's standard library, we need to explicitly import these. To ensure that numeric characters are indeed interpreted as real numbers, we need to enter the relevant scope.

```
Require Import Reals.
Open Scope R_scope.
```

By importing the real numbers, results and types become available that were not at first. Additionally, tactics become available. We will now import another library called `ssreflect` from a package called `mathcomp`, for Mathematical Components. Among other things, they provide some extra tactics and add some convenient functionality to existing tactics.

```
From mathcomp Require Import ssreflect.
```

We will now prove that squares of real numbers are nonnegative. The proof will use existing results on the real numbers, which will be explained along the way. The idea of the proof is to distinguish the cases where a real number r is greater or equal than zero, or less than zero. We will then use that $a \cdot b \leq a \cdot c$ if $b \leq c$ and $0 \leq a$.

```
Proposition square_nonneg (r : R) : 0 ≤ r * r.
Proof.
destruct (Rle_dec 0 r) as [rge0 | rlt0].
```

```
2 subgoals

  r : R
  rge0 : 0 <= r
  ============================
  0 <= r * r
subgoal 2 is:
 0 <= r * r
```

`Rle_dec` is a term which says that for any two real numbers r_1 and r_2, we have either $r_1 \leq r_2$ or $r_1 \not\leq r_2$. By supplying it with r and 0 and destructing it, we get a subgoal for each case. The destruction will not cause the goal to be rewritten, but will introduce new hypotheses. To name these, we add *as* to the destruct tactic. We give a different name for the hypothesis depending on the case. The vertical bar distinguishes between cases. If there are n cases, you need exactly $n-1$ bars for valid syntax.

To solve the first case, we will first use that $0 = r \cdot 0$. We then need to prove that $r \cdot 0 \leq r \cdot r$. This is true if $0 \leq r$ (and $0 \leq r$). This is currently one of our assumptions, so we can conclude this subgoal.

```
- rewrite -(Rmult_0_r r).
  by apply Rmult_le_compat_l.

1 subgoal

  r : R
  rlt0 : ~ 0 <= r
  ============================
  0 <= r * r
```

We used `rewrite` before, but here we have supplied it some options. By using (`Rmult_0_r r`) the explicit equality to rewrite with becomes $r * 0 = 0$. For some patterns Coq will automatically substitute the right variables, on other occasions you have to help a bit. By adding a hyphen, we rewrite the other way around: we replace 0 with $r * 0$ in the goal. This functionality was added by importing `ssreflect`. Furthermore, `Rmult_le_compat_l` contains the proof that if $b \leq c$ and $0 \leq a$, then $a \cdot b \leq a \cdot c$. Our goal has the latter form, so by applying it we need to prove $0 \leq r$ and $0 \leq r$. This is one of our hypotheses, so we are done. The `by` tactic is similar to the `now` tactic, finishing the proof if it is easy enough.

For the second goal, note that we have $0 \not\leq r$ as an assumption now. This is of course the same as $r < 0$, or that $0 < -r$. This last statement is the most useful to us, so we transform it to this first. This is accomplised by doing an `apply` and some `rewrite`s *in* the hypothesis `rlt0`.

```
- apply Rnot_le_lt, Ropp_lt_contravar in rlt0.
  rewrite Ropp_0 in rlt0.

1 subgoal

  r : R
  rlt0 : 0 < - r
  ============================
  0 <= r * r
```

Note also that we did two applies in succession, by separating them with a comma. We are now in a position to try an approach similar to the first case. We will say that $-r \cdot 0 = 0$, and that $r \cdot r = -r \cdot -r$. This last statement is true since $a = -(-a)$, and $-(a \cdot b) = (-a) \cdot b = a \cdot (-b)$ for all a and b in $\mathbb{R}$. These properties are captured in `Ropp_involutive`, `Ropp_mult_distr_l` and `Ropp_mult_distr_r` respectively. Once this rewriting has been performed, we apply `Rmult_le_compat_l`. This transforms the proof into two instances of $0 \leq -r$. Since $0 \leq -r$ is the same as $0 < -r \vee 0 = -r$, we can conclude with `left` since $0 <$ `-r` is one of our assumptions.

```
  rewrite -(Rmult_0_r (-r)).
  rewrite -(Ropp_involutive (r * r)) Ropp_mult_distr_l Ropp_mult_distr_r.
  by apply Rmult_le_compat_l; left.
Qed.
```

Although the proof is not hard, the proof might not be as short as we want it to be. Manipulating real inequalities quickly becomes a hassle. In situations that are slightly harder than trivial, visualising what happens by just looking at the tactics being used is already quite hard. The best way to see what is going on is to step through the proof step by step, using Coq's IDE (intelligent development environment). Making proofs done in Coq readable without doing this is a challenge.

There are multiples ways to tackle this problem. One approach is to automate as much as possible. This means we use powerful tactics which (try to) solve specific types of goals immediately. Another approach is to choose a different way to display these proofs, as advertised by Tankink [13]. Tankink has created a way to step through these proofs without having access to the proof assistant yourself, by displaying the proofs in a 'movie' format on a web page. Another approach is to explain the proof with words, but in a more formal way — mentioning all the needed details.

2.13. Interacting with Coq

As a computer program, there are several ways to interact with Coq. One can use a command-line interface to enter proofs and definitions, or to query the existing results. However, one usually writes

these in a source file, which has a '.v' file extension. Verifying the proofs in a source file is the same as 'compiling' them to a binary file with the '.vo' extension. Other files can then reuse the definitions and results of these binary files. We have seen this in the previous section: we needed results on real numbers, so we imported these results from other files, into the environment.

For large verification efforts, one needs to properly structure the different source files to keep them maintainable. For example, one could put all results on affine functions in the same file. When proving results on convex sets in a new file, we can import all required results on affine functions from the first file.

2.14. Ltac: custom automation

Sometimes different propositions require the same or very similar tactics to prove. It also happens that we need to manipulate the proof environment in the same specific way, multiple times. It would be nice if we could exploit this instead of copying and pasting code.

This is what Ltac enable us to do. Ltac is the third language we mentioned in Section 2.4. In the Ltac language, we are able to define custom tactics which can be used just like `intros`, `assumption` or `destruct`. Writing tactics will feel similar to writing proofs, but the syntax is a bit different. Ltacs are close enough to Gallina to be able to define terms easily, yet far enough that they can reason about and make decisions based upon the current proof context. This means that Ltacs might do different things in different contexts. Ltacs also have the ability to fail, which is something regular definitions or propositions cannot (and should not!) do. Failure can result in the user being prompted, or in compilation failure. Other Ltacs might detect failure and depending on this, apply one tactic or another.

We will convert the proof presented in the previous section, that all squares of real numbers are positive, to an Ltac. Although this Ltac is not suitable for re-use, it will demonstrate the capabilities of Ltac well. We will show the definition of the Ltac, and explain the new syntax afterwards.

```
Ltac prove_square_nonneg :=
  match goal with
  | ⊢ 0 ≤ ?r * ?r ⇒
    idtac "Proving that square of"r"is nonnegative." ;
    let rge0 := fresh "rge0" in
    let rlt0 := fresh "rlt0" in
    destruct (Rle_dec 0 r) as [ rge0 | rlt0];
    [ idtac "Proving when"r"is nonnegative..." ;
      rewrite -(Rmult_0_r r);
      by apply Rmult_le_compat_l
    | idtac "Proving when"r"is negative..." ;
      apply Rnot_le_lt, Ropp_lt_contravar in rlt0;
      rewrite Ropp_0 in rlt0;
      rewrite -(Rmult_0_r (-r)) -(Ropp_involutive (r * r))
          Ropp_mult_distr_l Ropp_mult_distr_r;
      by apply Rmult_le_compat_l; left ]
  | ⊢ _ ⇒ fail "Goal does not equal nonnegativity of the square of a real number"
  end;
  idtac "Done." .
```

Note that Ltacs are defined using the := operator, and this definition stops at a period. The first line of the definition is `match goal`. This behaves like the `match` expressions we saw in Gallina, except that there are some special keywords on which we can also perform a match. One of these is `goal`, which (unsurprisingly) tries to match the goal to the given cases. The first vertical bar signifies the start of a new case. The ⊢ symbol is the 'turnstile' symbol from logic, where $A \vdash B$ means something like 'proposition B is provable from proposition A'. The meaning is related but not entirely the same in the `match goal` construct; $A \vdash B$ is matched when we are *trying* to prove B from hypothesis A. We have not specified any hypothesis here, which means this clause is activated when we are trying to prove something of the form $0 \leq ?r * ?r$.

The question marks are needed in Ltac, since it is not allowed to use variables that have not been introduced. By using ?`r`, we signify that if the goal has the specified form, we subsequently name the variable whose square is in the goal `r`. This is why it is legal to do a destruct on a term which uses `r`.

`idtac` is a tactic which we have not seen before. It is the 'identity' tactic, in the sense that composing a tactic with `idtac` is identical to running the tactic itself. In other words: `idtac` has no effect. However, we can supply `idtac` with a string, and this string will be printed. We have inserted the `r` term in the middle of this string, which will cause the name of the term to be printed. This is sometimes useful for debugging or for alerting the user.

When we wrote the proof of `square_nonneg`, we used a period after every tactic. To convert this proof to an Ltac, we need to remove all these periods. This is because a period signifies the end of an Ltac. This is where the semicolon comes in; tactics can be run after one another by putting a semicolon in between.

We have seen `let` expressions before, but we have not seen `fresh` before. The `fresh` tactic creates a new variable name which starts with the given string. If the provided name is already taken, a numeric character is appended to the name. Naming new hypothesis in this way is therefore safe, since it will never use/overwrite an existing hypothesis. Note furthermore that the scope of this let expression is the entirety of the remaining match clause.

The semicolon is nice for stitching together tactics which should be applied to all cases, but often each case needs its own treatment. The [|] construct (like the one we used for `destruct as`) solves this problem: the tactics which solve the first case are inserted left of the vertical bar, those which solve the second case are inserted right of the vertical bar. The tactics in each case correspond to the ones we used originally in the proof.

A final note is in order on what happens if the goal does not match $0 \leq$?`r` $*$?`r`. The match-expression will then try to match the goal to one of the other clauses. The second match clause `_` matches everything, so will always be matched if the goal is not nonnegativity of a real number. The `fail` tactic will then be executed, which will alert the user of a failure with the provided string. We will showcase our tactic in the following to propositions.

```
Proposition square_nonneg' (x : R) : 0 ≤ x * x.
Proof.
prove_square_nonneg.

Proving that square of x is nonnegative.
Proving when x is nonnegative...
Proving when x is negative...
Done.
No more subgoals.

Qed.
```

The `idtac`s provide messages on the execution of `prove_square_nonneg`. Note that while the real variable was called `r` in the Ltac, its real name is `x` in the proof above, and this shows in the printed messages. Now let us see what happens when we try to apply this tactic to an incompatible goal.

```
Proposition test_failure (x : R) : 0 ≤ x * 0.
Proof.
Fail prove_square_nonneg.

The command has indeed failed with message:
Tactic failure: Goal does not equal nonnegativity of the square of a real number.
```

Indeed, the tactic fails. We prefix the tactic with `Fail` to be able to be able to compile nonetheless; otherwise this would result in a compilation failure.

The Ltac we defined above defines a tactic, which can be used in a proof script. There is another type of Ltac, which defines a term. These Ltacs cannot be used inside proof scripts, but they can be used inside other Ltacs. This is useful when the construction of a term is needed to progress, but the construction of such a term might fail or be undefined in some cases. We will now construct an example of such an Ltac. This Ltac is not useful, but serves to demonstrate some capabilities of Ltac.

```
Ltac count_implications term :=
let rec countimpl curterm :=
  match curterm with
  | ?A → ?B ⇒
    let prev := countimpl B in
    constr:(S prev)
  | _ ⇒ constr:(0)
  end
in countimpl term.
```

The let expression here is different from the ones we've seen before. The expression `let name argument1 argument2 := _ in _` defines a function `name`, which takes two arguments. If the expression starts with `let rec`, like in this case, the function can be recursive. This is indeed the case for `count_implications`, since the inner `countimpl` function is called in `let prev := countimpl B`. Furthermore, the return values are wrapped in `constr:( )` tags. This means that the return value is not a tactic but a term.

The `countimpl` function tries to match `curterm` to a function type. If this fails, we return $0 \in \mathbb{N}$. If `curterm` is matched to `?A → ?B`, we determine the result $r \in \mathbb{N}$ of `countimpl B`, and return $r + 1$. The effect of `count_implications` is then indeed to count the implication symbols: `nat → nat → nat` should return $2 \in \mathbb{N}$. We cannot use this directly as a tactic, since it returns a term. We will therefore create another (simple) Ltac, which just prints the results.

```
Ltac print_goal_depth :=
match goal with
| ⊢ ?A ⇒
  let implications := count_implications A in
  idtac "Goal depth of"implications
end.
```

`print_goal_depth` contains no further surprises. And indeed, it works as expected:

```
Proposition deep3_goal (A B C : Prop) : A → B → C → A.
Proof.
print_goal_depth.
```

```
Goal depth of 3
```

Ltac is not without its flaws; there is no typechecking until runtime, and debugging is hard. It is quite powerful; combinations of the two types of Ltacs allow us to automate a lot of things. It is also very nice that this automation is, in a way, inside of Coq: one does not need to mess around with external programs.

3

Formal verification in Coq

This chapter aims to describe how we formally verified some of the results of Dostert et al. [5]. To do this, we will use Coq to verify that the functions found by Dostert et al. indeed satisfy the third condition of Theorem 1. If this condition is satisfied, we obtain upper bounds for the corresponding shapes $\mathcal{K}$. The main goal here is to verify that

$$f(x) \leq 0 \qquad \text{whenever } x \notin \mathcal{K}^\circ - \mathcal{K}^\circ \text{ and } s(x) < 0.$$

Here $s(x)$ is usually defined as $s(x) = \|x\|^2 - 1$, so that $s(x) < 0$ if and only if $\|x\| < 1$. We will first focus on proving

$$f(x) \leq 0 \qquad \text{whenever } x \notin \mathcal{K} - \mathcal{K} \text{ and } \|x\| \leq 1,$$

which will later be shown to be enough by a continuity argument. In Coq, this result will look like:

```
Theorem goal : ∀ x : R × R × R , norm x ≤ 1 → ¬ octahedron x → solution_poly x ≤ 0.
```

Here `octahedron` is the Minkowski difference $\mathcal{K} - \mathcal{K}$, for $\mathcal{K}$ equal to the tetrahedron. It will be defined as an intersection of half-spaces defined by linear inequalities, i.e.

```
Definition octahedron (v : R × R × R) : Prop :=
  v.1.1 + v.1.2 + v.2 ≤ 1 ∧ v.1.1 + v.1.2 + -v.2 ≤ 1 ∧
  v.1.1 + -v.1.2 + v.2 ≤ 1 ∧ v.1.1 + -v.1.2 + -v.2 ≤ 1 ∧
 -v.1.1 + v.1.2 + v.2 ≤ 1 ∧ -v.1.1 + v.1.2 + -v.2 ≤ 1 ∧
 -v.1.1 + -v.1.2 + v.2 ≤ 1 ∧ -v.1.1 + -v.1.2 + -v.2 ≤ 1.
```

Note that $R \times R \times R$ is notation for $(R \times R) \times R$, while .1 and .2 return the first and second coordinate of a pair respectively. This means `v`.1.1 is the first, `v`.1.2 the second, and `v`.2 the third coordinate of `v`.

`solution_poly` will be an explicit polynomial, but its coefficients will not be exactly defined. This is because they cannot be represented exactly as a floating point number — they can, however, be approximated to some interval. In Coq, we will assume the coefficients are contained in this interval.

```
Parameter c_0 : R.

Conjecture c_0_bound : 1 ≤ c_0 ≤ 1.

Parameter c_1 : R.

Conjecture c_1_bound : 1 ≤ c_1 ≤ 1.

Definition solution_poly (v : R × R × R) : R :=
  c_1 × (v.1.1 ^ 2 + v.1.2 ^ 2 + v.2 ^ 2) - c_0.
```

Note that `solution_poly` is just an example; it does not satisfy the conditions for $\|x\| \geq 1$.

These statements are reasonably concise, and one only has to check that these statements indeed correspond to the remaining conditions of Theorem 1 to use the results of the verification. This means that to trust the verification, an understanding of every detail of the proof is not required. By trusting the proof assistant itself, we trust all results it deems valid.

Even so, we will explain in this chapter the details of the formal proof. Formalization is no small effort and poses some interesting challenges.

Five main obstacles had to be overcome for the verification. The functionality required to do this was not found in Coq or any of Coq's user-contributed packages. Some of these obstacles may seem trivial to our human mathematical intuition, but turn out to be harder to formally verify.

- Computation of (intervals of) values of $f(x)$ and $\nabla f(x)$ needs to be done in reasonable time. If this computation is too slow, the verification might not be tractable.
- We need to prove the multidimensional mean-value theorem, at least in $\mathbb{R}^3$.
- To apply the mean-value theorem, we need to be able to verify that ∇f is the derivative of f.
- We need a way to manipulate and verify lists of propositions, so that we can perform verification at a list of gridpoints.
- We need some results on convexity. This is mainly needed for the verification of nonpositivity of f in cubes.

We will discuss how these obstacles were overcome in Section 3.2. We will first discuss some of the packages we used in Section 3.1. These packages provide useful results and Coq tactics, which we will need in Section 3.2.

With this extra functionality we were able to verify the results of Dostert et al. This verification was broken up in several steps.

- We need to prove that ∇f is the derivative of f. The extra problem here is that f can be represented in multiple polynomial bases: either in the regular x,y and z corresponding to the axes in $\mathbb{R}^3$, or in s_1, s_2 and s_3: the basis of B_3-invariant polynomials.
- We need to be able to verify that f is nonpositive in a cube which *does not* intersect $\mathcal{K}^\circ - \mathcal{K}^\circ$.
- We need to be able to verify that f is nonpositive in a cube which *does* intersect $\mathcal{K}^\circ - \mathcal{K}^\circ$.
- We need to verify that the union of all cubes cover the required region.

These steps are described in Section 3.3. To perform the verification, we need additional tools outside of Coq. These tools are covered in Section 3.4.

A note is in order on the presentation of proofs in Coq. Some proofs in Coq, especially large ones, are not able to convey the intuition behind them. It is hard to follow the effect tactics have without running an instance of Coq next to the explanation. Sometimes specific manipulations and rewrites are necessary to make Coq understand steps in the proof, while a human reader might see this directly.

We will show some of the proofs done in Coq, but will prefer regular proofs that are formal enough to be translated to Coq without too much difficulty. We will always mention the statements in Coq. If the statement in Coq is clear enough, we will sometimes omit the regular, mathematical version of the result.

Section 3.2.2 will explain the proof in Coq of the multidimensional mean value theorem. Section 3.2.1, on speeding up interval arithmetic, will also contain Coq implementation details, since the main challenge was the implementation in Coq. The rest of the sections will use the regular but formal proof approach.

3.1. Used packages

While Coq provides various results on real numbers, it lacks two things we require. These are support for interval arithmetic, and some results in real analysis. However, these are available in two user-contributed packages. These will be discussed below.

3.1.1. Interval package for Coq

The interval package for Coq was developed by Érik Martin-Dorel and Guillaume Melquiond [11]. This package provides a tactic '`interval`' to prove real inequalities of various functions, provided we have some bound on the variables. It uses interval arithmetic and floating point computations to prove these inequalities. We will first discuss some examples, and then explain some of the internal workings of the package. This last part is relevant, since in our case we can (and need to) improve the speed of the `interval` tactic.

3.1.1.1. Interval package: examples

Below are two examples of how the `interval` tactic is used to prove inequalities. The first example is an obvious use of the interval package. It starts with the assumption x is in the interval $[3/5, 4/5]$. The

Interval package supports various functions, one of which is tan. Note that tan is applied to an interval, not a single value. The function is also not evaluated exactly, but approximated. However, the interval package also proves the accuracy of these approximations. This allows it to prove something about the exact value.

```
Coq < Theorem example_ineq1 (x : R) : 3/5 <= x <= 4/5 -> 2/3 <= tan x <= 34/33.

Coq < Proof.

Coq < intros.
1 subgoal

  x : R
  H : 3 / 5 <= x <= 4 / 5
  ============================
  2 / 3 <= tan x <= 34 / 33

Coq < interval.
No more subgoals.

Coq < Qed.
```

The next inequality does not show any associated interval. However, the interval package computes both $\sin(13)$ and $1/\sqrt{5}$ using floating point arithmetic. This is not an exact computation, but it is possible to keep track of the errors with intervals. That is, we have numbers l_1, u_1 such that $\sin(13) \in [l_1, u_1]$, and in this case we hope that l_1 is close to u_1. In the same way we get numbers l_2, u_2 such that $\sqrt{5} \in [l_2, u_2]$. If $u_1 < l_2$, the proof is finished.

```
Coq < Theorem example_ineq2 : sin 13 < 1 / sqrt 5.
1 subgoal

  ============================
  sin 13 < 1 / sqrt 5

Coq < Proof.

Coq < interval.
No more subgoals.

Coq < Qed.
```

3.1.1.2. Interval package: explanation

There is actually quite a lot going behind the `interval` tactic. As discussed in Section 1.4.1, we need to find a function $\mathbf{f}$ satisfying the containment property, to provide a bound on an expression $f(x)$. We can build such an $\mathbf{f}$ if we can decompose f into compositions of elementary functions.

This is exactly the strategy used by the interval package. Armed with containment functions for several elementary functions, it decomposes a given function f into compositions of elementary functions. This gives us an $\mathbf{f}$, and thus a bound by evaluating $\mathbf{f}(\mathbf{x})$.

Decomposing a function in Coq is no small feat. There is no way to know beforehand whether a given function can be decomposed into the known elementary functions. This makes proving solid theorems about this decomposition method hard. To overcome this, `Ltac`s are used. Their ability to manipulate the Coq environment makes them ideally suited to the task.

Suppose we wish to prove that $f(x) \leq 0$ when $x \in \mathbf{x}$. When running the `interval` tactic, the procedure that is run consists of the following steps:

- The function $f(x)$ of type $\mathbb{R}$ is 'reified' intro an expression F, an abstract syntax tree, of type `expr`. This expression F consists of constants and elementary operations on these constants.
- The expression F is then converted to a list of stack-based computations `f` of type `list term`. This `f` can be mapped to functions on different domains. For example, if mapped to a function on $\mathbb{R}$, we will recover f. We will suggestively denote this as $\texttt{f}(\mathbb{R}) = f$. We can also map `f` to a function on intervals, denoted by $\texttt{f}(I)$. By construction, this function $\texttt{f}(I)$ will now satisfy the containment property on f.

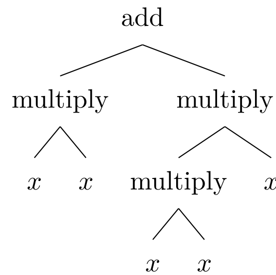


Figure 3.1: Abstract syntax tree for $x \cdot x + x \cdot x \cdot x$

- All the relevant intervals are extracted out of the local proof context (i.e. $\mathbf{x}$).
- A theorem is applied that says it is enough to prove $\overline{\mathtt{f}(I)(\mathbf{x})} \leq 0$, since $f(x) \leq \overline{\mathtt{f}(I)(\mathbf{x})}$.
- The $\mathtt{f}(I)(\mathbf{x})$ computation is performed and compared to conclude.

The notation $\mathtt{f}(\mathbb{R})$ and $\mathtt{f}(I)$ might seem mathematically a bit sketchy. However, it properly reflects what is going on in Coq. The list of (abstract) computations $\mathtt{f}$ is really evaluated at an algebraic structure with a definition for each of the elementary operations. The result of this evaluation is a function of this algebraic structure to itself: $\mathtt{f}(I)$ sends intervals to intervals and $\mathtt{f}(\mathbb{R})$ sends real numbers to real numbers.

Remember that the current definition of real numbers in Coq has no computational meaning, unlike for example the natural numbers. This computation could therefore not be done if we were still using real numbers. That is to say, $\mathtt{f}(I)$ is not a function on intervals of real numbers: instead, it is a function on intervals of floating point numbers. These *do* have computational meaning and are thus comparable.

We will illustrate what the abstract syntax tree F and the list of stack-based computations $\mathtt{f}$ mean with an example. Suppose we wish to perform interval arithmetic on the expression $x \cdot x + x \cdot x \cdot x$. Such an expression in Coq contains hidden brackets. The actual expression is $(x \cdot x) + ((x \cdot x) \cdot x)$. This expression would be converted to the abstract syntax tree in Figure 3.1. Note the repetition of a subtree which multiplies x with x. It is possible to directly construct a function $\mathtt{f}$ satisfying the containment property from an abstract syntax tree F, but repeated subexpressions would have to be calculated repeatedly. By constructing $\mathtt{f}$ first, the interval package makes the resulting computation more efficient. The construction of $\mathtt{f}$ guarantees that no subexpressions of f are computed twice, which makes $\mathtt{f}(I)$ as efficient as possible.

The list of computations $\mathtt{f}$ for our expression $x \cdot x + x \cdot x \cdot x$ would be the sequence of the following three computations. The computations operate on a stack, which contains just the element x at the start of the computation. The state of the stack after each operation is shown to the right of the computation.

$$\begin{aligned} \text{multiply } 0{,}0 \quad & (x \cdot x, x) \\ \text{multiply } 0{,}1 \quad & ((x \cdot x) \cdot x, x \cdot x, x) \\ \text{add } 1{,}0 \quad & ((x \cdot x) + ((x \cdot x) \cdot x), (x \cdot x) \cdot x, x \cdot x, x) \end{aligned}$$

Note that this list of computations has integers as operands, indicating the location of the actual operand in the stack. When the computation is finished, the answer can be found at the head of the stack.

The algorithm which constructs $\mathtt{f}$ out of F is of exponential complexity in the number of subexpressions of F. This is because for every subtree of F, all existing computations are searched to determine whether this expression has been calculated before. This way of doing things is only efficient when the time gained by efficient floating point calculations exceeds the time lost by constructing $\mathtt{f}$ out of F. This might no longer be true when f is a large expression.

It might be possible to improve this algorithm. Currently, an expression in the tree is compared to every existing computation. However, a leaf-expression will never be equal to an expression higher in the tree. It might be possible to leverage this in a more complicated algorithm, where we also store the tree-depth of every expression. New expressions need only be compared to expressions of the same tree-depth.

3.1.2. Coquelicot: real analysis for Coq

The Coquelicot package provides several results and tactics for real analysis. It was developed by Sylvie Boldo, Catherine Lelay and Guillaume Melquiond [1]. The Coq standard library defines differentiability and integrability for functions $\mathbb{R} \to \mathbb{R}$. However, we need a notion of differentiability on functions $\mathbb{R}^3 \to \mathbb{R}$. Coquelicot provides this, and several other results on real analysis. We will first discuss the structures Coquelicot defines, and then continue with its approach to differentiability.

3.1.2.1. Coquelicot: structures

Coquelicot defines several abstract structures with some associated operators and axioms, such as rings and modules. Results are then proven for these abstract structures. Note that this does not differ from the regular approach in mathematics, but this is not straightforward to implement in Coq. Coq enforces strict typing rules, and does not allow us to just substitute types. If we have shown some results on a general normed vector space U, we are not able to just substitute $\mathbb{R}^3$ for U. However, this can be solved by using 'Canonical Structures'. Canonical structures can be used to automatically associate types with related types. For example, they can be used to associate $\mathbb{R}$ with the usual ring structure $(\mathbb{R}, +, \cdot)$ on $\mathbb{R}$. Coquelicot performs this association by first building the ring type for $(\mathbb{R}, +, \cdot)$. This requires a proof of the ring-axioms for $\mathbb{R}$ with these operators. By declaring the obtained ring to be Canonical, Coq will use this ring structure whenever a ring structure is needed on $\mathbb{R}$, and none other is specified. This makes it easy to use, since as a user of the package you do not need to worry about this association: it all happens automatically.

Coquelicot defines several abstract structures. The structures most relevant to us are the following: abelian groups, (absolute) rings, module spaces, uniform spaces and normed modules. Abelian groups and rings are defined as expected. Absolute rings are rings equipped with an absolute value. An absolute value on a ring $\mathbb{K}$ must satisfy:

- $|\cdot|$ is a function from $\mathbb{K}$ to $\mathbb{R}$
- Axiom 1: $|0_{\mathbb{K}}| = 0$
- Axiom 2: $|-_{\mathbb{K}} (1_{\mathbb{K}})| = 1$
- Axiom 3: for all x, y in $\mathbb{K}$, we have $|x \cdot_{\mathbb{K}} y| \leq |x||y|$
- Axiom 4: for all x, y in $\mathbb{K}$, we have $|x +_{\mathbb{K}} y| \leq |x| + |y|$
- Axiom 5: for all x in $\mathbb{K}$, we have that $|x| = 0$ implies $x = 0_{\mathbb{K}}$

If one requires $|x \cdot_{\mathbb{K}} y| = |x||y|$ instead of axiom 3, axiom 2 is not needed. However, this is the approach taken in Coquelicot.

Module spaces are abelian groups G equipped with a scalar multiplication operation $R \times G \to G$, where R is some ring. These are also defined as expected. Remember that a module space is not the same as a vector space; vector spaces are defined over fields, module spaces are defined over rings.

Uniform spaces are spaces U with a notion of open balls. This is slightly more general space than a metric space. Balls on uniform spaces need to satisfy the following axioms:

- $B(x, \epsilon) \subseteq U$ for all $x \in U$ and $\epsilon \in \mathbb{R}$. Note that in Coq this will have type `U` → `R` → `U` → `Prop`.
- Axiom 1: for all $x \in U$ and all $\epsilon > 0$, we have $x \in B(x, \epsilon)$.
- Axiom 2: for all $x, y \in U$ and $\epsilon \in \mathbb{R}$, we have that $x \in B(y, \epsilon)$ implies $y \in B(x, \epsilon)$
- Axiom 3: for all $x, y, z \in U$ and ϵ_1, ϵ_2 in $\mathbb{R}$, we have that if $y \in B(x, \epsilon_1)$ and $z \in B(y, \epsilon_2)$, then $z \in B(x, \epsilon_1 + \epsilon_2)$.

The last type of space we will discuss are normed modules. Coquelicot defines these as spaces U which are module spaces over some absolute ring $\mathbb{K}$, uniform spaces, and are additionally equipped with a norm. Some compatibility is also required between the norm and the ball. The norm must satisfy the following conditions:

- $\|\cdot\|$ is a function form U to $\mathbb{R}$
- Axiom 1: for all x, y in U, we have $\|x +_U y\| \leq \|x\| + \|y\|$
- Axiom 2: for all $k \in \mathbb{K}$ and $x \in U$, we have $\|k \cdot_U x\| \leq |k|_{\mathbb{K}} \|x\|$
- Axiom 3: for all $x, y \in U$ and ϵ, we have that $\|y -_U x\| < \epsilon$ implies $y \in B(x, \epsilon)$
- Axiom 4: There exists some 'norm factor' $N_U \in \mathbb{R}$ satisfying the following property. For all $x, y \in U$ and $\epsilon > 0$, if $y \in B(x, \epsilon)$ then $\|y -_U x\| < N_U \cdot \epsilon$
- Axiom 5: for all $x \in U$, we have that $\|x\| = 0$ implies $x = 0_U$.

Note that a normed module is now endowed with two topologies; that of the uniform space, and that of the norm. These topologies are equal, but do not necessarily have the same notion of open balls. For $\mathbb{R}^n$ with $n > 1$, the ball of the uniform space is an open cube, while the ball of the norm is the open norm-ball.

In Coq, we shall use the product type `R * R * R` for $\mathbb{R}^3$. This is seamlessly interpreted as a normed module, because of the Canonical structure given to products of normed modules. This means the norm is not explicitely specified for a product type `U * V`, but built from the norms on `U` and `V`. The product norm is defined as

$$\|(u,v)\|_{\mathsf{U*V}} = \sqrt{\|u\|_{\mathsf{U}}^2 + \|v\|_{\mathsf{V}}^2},$$

which will coincide with the regular Euclidean norm on $\mathbb{R}^n$. The product ball is, however, defined as

$$B((u_0,v_0),\epsilon)_{\mathsf{U*V}} = \{(u,v) : u \in B(u_0,\epsilon)_{\mathsf{U}}, v \in B(v_0,\epsilon)_{\mathsf{V}}\}$$

which will coincide with an open cube in $\mathbb{R}^n$.

Coquelicot comes with a definition for $\mathbb{R}^n$ for general n. Although it is possible to use this definition when working with $\mathbb{R}^3$, it is less convenient to use than when seeing $\mathbb{R}^3$ as the product of $\mathbb{R}^2$ and $\mathbb{R}$. Elements of $\mathbb{R}^n$ for general n are seen as functions of the natural numbers to the real numbers, for which only the first n values are relevant, while an element of `R * R * R` is precisely a triple of real numbers.

3.1.2.2. Coquelicot: differentiability

An approach for differentiability on $\mathbb{R}^n$ would be to define a Jacobian, a matrix consisting of all the partial derivatives. However, this requires results on vectors and matrices. To avoid this, Coquelicot takes a different approach — that of the Fréchet derivative, as mentioned in Section 1.4.2. This does not require matrices, only abstract linear operators, and these are easier to define. Furthermore, it works not just for $\mathbb{R}^n$, but for every normed module. We need normed modules here to ascertain that the operators are *bounded* linear operators.

Fréchet derivatives of functions $\mathbb{R} \to \mathbb{R}$ are functions $h \mapsto f'(x) \cdot h$. This makes it is possible to extract the original derivative out of the operator A. Coquelicot proves that the differentiability on $\mathbb{R} \to \mathbb{R}$ of the Coq standard library and the Fréchet derivative are equivalent. Furthermore, it provides a convenient tactic `auto_derive`, which is able to automatically prove that some function is indeed the derivative of another function.

3.2. Building required functionalities

Let us repeat the functionalities we need to build to be able to verify the results of Dostert et al [5].

- Computation of (intervals of) values of $f(x)$ and $\nabla f(x)$ needs to be done in reasonable time. If this computation is too slow, the verification might not be tractable.
- We need to prove the multidimensional mean-value theorem, at least in $\mathbb{R}^3$.
- To apply the mean-value theorem, we need to be able to verify that ∇f is the derivative of f.
- We need a way to manipulate and verify lists of propositions, so that we can perform verification at a list of gridpoints.
- We need some results on convexity. This is mainly needed for the verification of nonpositivity of f in cubes intersecting the minkowski difference $\mathcal{K} - \mathcal{K}$.

Each of these functionalities will be discussed in a separate subsection. The first two subsections will alternate between text and pieces of Coq code, to make the reader understand both the motivation of the code and the code itself. The remaining two subsections will mention the results in Coq, but discuss the proofs in a regular fashion.

3.2.1. Interval arithmetic on large expressions

As mentioned in Section 3.1.1, constructing a list of interval computations for large expression in Coq is quite costly. This becomes evident when performing interval arithmetic for the polynomials found by Dostert et al [5]. The `interval` tactic takes about 40 seconds to prove a bound on the polynomial for the truncated tetrahedron. We have estimated that the total verification time needed would then be around 34 processor-years, so improvements were needed.

A significant amount of time was therefore spent on investigating and improving the speed of the interval arithmetic in Coq. For our polynomials p, the calculation itself would take about half a second. Mapping p to P to p and getting the bounds out of the local context would take the remaining 39.5 seconds. We should be able to take advantage of two things: p (and so p as well) is equal in all our evaluations, and we know the bounds on the coefficients of the polynomials in advance. We should be able to somehow 'save' p and bounds of the expression to some structure, and reuse this structure.

Coq is, of-course, a proof assistant, but it can also be used as a programming language. It is therefore quite possible to define such a structure, although its meaning is more computational than mathematical. This section will describe how this structure and the procedures using it were defined. The idea of reusing this structure is not hard to grasp for a human, but making Coq grasp it is not a trivial task, so some implementation details are in order.

We will proceed with the construction below, and will also create various tactics which use the structure to perform the required interval arithmetic.

The `interval_container` is the structure we have been talking about. It contains all information to perform an interval arithmetic computation of the function `term : U → R`. Here `U` is usually some product of `R`, such as `R × R × R`. The function `vars : U → list R` maps an element of the input type to a list of real numbers, for which bounds have to be supplied. In the case where `U` is `R × R`, we would set `vars` to be `fun v : R × R ⇒ [ :: v.1 ; v.2 ]`; a list of the coordinates.

```
Record interval_container {U : Type} := {
  vars : U → list R;
  term : U → R;
  evalProg : list Interval_bisect.term;
  allVars : U → list R;
  progValid :
    ∀ v : U, List.nth 0 (eval_real evalProg (allVars v)) 0 = term v ;
  constBounds : list Private.A.bound_proof
}.
```

The vernacular `Record` command provides a way to define a new data structure consisting of several elements. Elements are specified between curly brackets, and separated by a semicolon. The provided names can be used to access the elements of a given `interval_container`.

`evalProg` contains the way to compute `term`: it is the list of stack-based computations mentioned in Section 3.1.1.2. In our previous notation: `term` is the function p, and `evalProg` is the function p. Remember that once p finishes, the result of the computation is at the top of the stack. `List.nth` 0 takes the element at the top of the stack, `eval_real` interprets `evalProg` as a real expression. This means that `progValid` is a proof that when interpreting `evalProg` as a real expression, the result is precisely `term`. Or, using the suggestive notation of Section 3.1.1.2, `progValid` is a proof that $\mathsf{p}(\mathbb{R}) = p$. Finally, `allVars` maps an input element to all used variables and constants, whereas `constBounds` (indeed) contains proofs of all the bounds of the used constants.

Let us consider an example to become more familiar with the different elements of this structure. Suppose we want to create an `interval_container` for the function $x\colon \mathbb{R} \mapsto x \cdot x + x \cdot x \cdot \pi$. In this case, `U` is equal to `R`. Then `vars` is equal to `fun r ⇒ [:: r]`, and `term` is equal to the function. Note that π is a constant in the function, but it is one of the 'variables' on which the list of computations will operate. This means `allVars` will be equal to `fun r ⇒ [::r; PI]`. Recalling the similar example in Section 3.1.1.2, `evalProg` will be equal to `[:: add 1,0 ; multiply 0,2 ; multiply 0,0]`, operating on `allVars`. Then `progValid` is a proof that this `evalProg` evaluated as a real expression is indeed equal to our function. Finally, `constBounds` should be a list containing a proof of a bound on `PI`. For example, it could contain an element `pi_bound` which has type $3 \leq$ `PI` ≤ 4 (so is a proof that this inequality holds).

To construct an `interval_container` directly, we would need to supply each of the required elements at once. This is okay for `vars` and `term`, but the other four elements should be constructed out of `vars` and `term`. We will therefore create a tactic which automates this process, which is discussed in Section 3.2.1.1. After having constructed an `interval_container`, we need a tactic to use this structure. This will be discussed is Section 3.2.1.2.

3.2.1.1. Constructing an `interval_container`

If our `interval_container` uses constants with known bounds, we want to supply this to the tactic beforehand. Intuitively, we want to give a list of proofs that constants are in a certain interval. However,

we cannot use a regular list. In a list all elements have the same type. For example, we could build a list of real numbers, where all elements of the list have type *R*. The list we want to build should contain proofs of different propositions. Recall the previous example, where constBounds could contain an element pi_bound : $3 \leq$ PI ≤ 4. For a different expression, we might want the list to contain an additional element e_bound : $2 \leq$ e ≤ 3. Note that pi_bound and e_bound are proofs of different propositions, so they have different types!

We will therefore store these bounds in a new kind of list; a proof_list. The elements we want to put together in a list do not have the same type, but they are all proofs of some Prop. In other words, the type of the type of the elements are Props. This motivates the following definition of proof_list:

```
Inductive proof_list : Type :=
  | empty_proof_list : proof_list
  | proof_cons {statement : Prop} : statement → proof_list → proof_list.
```

Next up is converting a provided proof_list into a list Private.A.bound_proof. This conversion is necessary, since Private.A.bound_proof is the type used by the interval package for proofs that some term lies in some floating point interval.

A given proof_list is allowed to contain bounds for constants which do not appear in our expression. However, all the constants which are in our expression should be in the provided proof_list. Therefore, we will first construct an Ltac which will extract a Private.A.bound_proof for a term t by searching through our proof_list, which we dub our non_local_bounds.

```
Ltac get_bounds_constant t non_local_bounds :=
  match t with
  | INR ?n ⇒ get_nat_bound n
  | _ ⇒ findBound t non_local_bounds
  | Private.I.T.toR ?v ⇒
     constr:((let f := v in Private.A.Bproof t
               (Private.I.bnd f f) (conj (Rle_refl t) (Rle_refl t)), @None R))
  | _ ⇒
     constr:((Private.A.Bproof t
               (Private.I.bnd SFBI2.nan SFBI2.nan) (conj I I), @Some R t))
  end.
```

We first check whether t is some $n \in \mathbb{N}$ mapped to $\mathbb{R}$ with INR (Injection of Naturals to Reals). These usually do not appear in user-provided functions, but they do appear in generated derivatives. This is because the power function has type *R* → nat → *R*, which means that x^{n+1} is encoded as pow x (S n). This expression will then have derivative $(n+1)x^n$ = INR (S n) × pow x n. The interval package itself provides no way to extract bounds for such a term. They do provide support to extract bounds for terms like IZR z, which is some $z \in \mathbb{Z}$ mapped to $\mathbb{R}$. We could rewrite our function beforehand to convert all INR n terms to IZR z terms, but this is not easy. Using rewrites in a definition will create an ugly term, since the rewrite is not forgotten. Rather, it is part of the definition: to get your term, you need to rewrite it using some rule. Attempts to change the generation of derivatives to create an IZR term were met with comparable difficulties.

Another option is to just perform the INR computation. However, this will greatly increase the size of the expression, and thus the size of the abstract syntax tree. This is because INR maps natural numbers to sums of $1 \in \mathbb{R}$: INR 3 will be mapped to $1 + 1 + 1$, etc. Since larger natural numbers might be present in our term, this is not what we want. This would cause a significant slowdown in the construction of the list of interval computations for our term.

Therefore, we hide the INR to INZ conversion inside the proof of the bound for INR. This also enables us to use the bound extraction for INZ provided by the interval package. This is what get_nat_bound does.

For other terms, we rely on the interval package to figure out how to extract the bound. This is done in findBound. It might fail on terms of the form Private.I.T.toR v. These are floating-point values v mapped back to $\mathbb{R}$, and bounds can therefore be easily recovered. If it does fail, match will try to match the term to other match statements, and try them instead. This means they will enter the third term, and the proper bound will be output.

Finally, if none of the above statements succeed, the $[-\infty, \infty]$ bound is returned. A value of SFBI2.nan (Not A Number) is used to signify this infinite interval. Note that the return value is actually a pair, where the second part of the pair is an option *R*, which can either be None or Some value. If no bound was found, Some is returned. After all variables have been extracted, all Some values

are collected. Only if bounds have been found for all terms, so no Some values are present, we finish building the interval_container.

```
Ltac get_bounds_constants l non_local_bounds :=
  let rec aux l lw :=
    match l with
    | nil ⇒ constr:((@nil Private.A.bound_proof, @nil R))
    | cons ?x ?l ⇒
      let i := get_bounds_constant x non_local_bounds in
      match aux l lw with
      | (?m, ?lw) ⇒
        match i with
        | (?i, @None R) ⇒ constr:((cons i m, lw))
        | (?i, @Some R ?aw) ⇒ constr:((cons i m, cons aw lw))
        end
      end
    end in
  aux l (@nil R).
```

get_bounds_constants applies get_bounds_constant to all constants in the list l. It again returns a pair, the first value of which contains the list Private.A.bound_proof, the second value contains a list of values for which no bound was found. Note the use of a recursive let expression, signified by let rec. Note also that this Ltac, like the previous one, constructs a term. We would like to know if get_bounds_constants failed to find bounds for some constant. We could edit the previous match clause to not match @Some R ?aw, but this will give us a generic 'No match clause' error. Instead, we implement a graceful failure in the following Ltac:

```
Ltac poseConstantBounds boundname constants non_local_bounds:=
  let raw_constants := (eval cbv -[Private.I.T.toR INR] in constants) in
  let raw_bounds := (eval cbv -[Rle] in non_local_bounds) in
  match get_bounds_constants raw_constants raw_bounds with
  | (?bounded_vars, ?unbounded_vars) ⇒
    match unbounded_vars with
    | (@nil R) ⇒
      pose (boundname := bounded_vars)
    | ?var_list ⇒
      fail "No bounds found for" var_list
    end
  end.
```

poseConstantBounds is thus the toplevel tactic for extracting bounds for constants from a provided proof_list. Provided constants and non_local_bounds are first further evaluated, to unfold their definitions. If any constants exist for which no bounds could be found, we fail with a message.

We need one more thing before we are ready to construct an interval_container. When computing the list of variables in term, the first couple of variables will be exactly that: the variables as specified in vars. The remaining part of the list should be constants, for which a bound should be provided in a proof_list. We need a way to strip away the first variables, or equivalently, to get the tail of the list of variables at the nth place: nthtail. The definition is pretty straightforward; recursively traverse the tail of the list until $n = 0$, and then return the list. Recall that the tail of an empty list is defined as the empty list, while the tail of a non-empty list is the list without the head.

```
Fixpoint nthtail {A : Type} (n: nat) (l: list A) : list A :=
  match n with
  | 0 ⇒ l
  | S n ⇒ (nthtail n (tail l))
  end.
```

We are now ready to build an interval_container. The build_container tactic will do so, and this tactic is printed below. Of course, we need to supply the functions term and vars, the way of mapping input elements to *R*. Here element will be any element of the input type U of the function term. This element is needed, since we are only able to compute evalProg and allVars as a *function* of an element of type U. These functions take a constant value, but these constant values can only be recovered if we can evaluate the function at an element.

The argument extract_algo will be a tactic which can extract a list of computations f to compute term. The interval package comes with one such method, we have constructed several others. These

generally compute a less efficient list, but the computation itself aims to be faster. These alternatives might be useful for large computations which only need to be done a couple of times. It is also possible to directly provide a list of computations `f` for `term`. This will be done later on; we will use Python to do the computation, then save the list of instructions in a newly generated Coq source file, which uses these computations to build an `interval_container`.

The `let name := fresh "name"` can be skipped over when reading. Recall from Section 2.14 that in `Ltac`s, unlike in regular Coq, it is invalid to use a new name without declaring it first. The `fresh` declares a new name, so that it is valid to use this name in the `Ltac`.

```
Ltac build_container term vars element boundlist extract_algo print_extract:=
  match (type of element) with
  | (?U) ⇒
    let extraction := fresh "extraction" in
    simple refine (let extraction :=
          (_ : U → (list Interval_bisect.term) × (list R)) in _) ; [
```

We first compute the list of computations, and call this `extraction`. This will also store the variables of the computation.

```
      let u := fresh "u" in
      intro u;
      let extractRaw :=
        (eval cbv -[powerRZ Rabs sqrt Rsqr IZR INR fst snd pow] in (term u)) in
      let theterm := fresh "theterm" in
      pose (theterm := extractRaw);
      change R1 with (IZR 1%Z) in theterm;
      change R0 with (IZR 0%Z) in theterm;
      let varsRaw := (eval cbv -[fst snd] in (vars u)) in
      let termRaw1 := (eval cbv delta [theterm] in theterm) in
      clear theterm;
      let termRaw := (eval cbn [fst snd] in termRaw1) in
```

All let expressions above prepare `term` to be in a state where `extract_algo` can safely construct our list of computations. If it was specified that we need to `print_extract`, we use `idtac` to print the result.

```
      let result:= extract_algo termRaw varsRaw in
      match print_extract with
      | true ⇒ idtac result
      | false ⇒ idtac
      end;
      exact result | ];
```

Now that we have our `extraction`, we can define `evalProg` and `allVars`. We then need to prove that our extraction actually is equal to our original, and provide a list of bounds on our constants.

```
      apply (Build_interval_container U vars term
        (fst (extraction element)) (fun v ⇒ snd (extraction v))); [
      let v := fresh "v" in
      intro v;
```

Proving equality is actually easy, because the two terms should reduce to the same term. This means we do not need any rewrites, and can directly conclude with `refl_equal`.

```
      exact (refl_equal (term v)) |
      let thebounds := fresh "thebounds" in
```

We apply our method for getting bounds for constants, after using `nthtail` to cut off the variables.

```
      poseConstantBounds thebounds
        constr:(nthtail (length (vars element))
                (snd (extraction element))) boundlist;
      exact thebounds ]
  end.
```

3.2.1.2. Using the interval_container

To start using our interval_container, we need some more infrastructure. In this section we will build two tactics, use_interval_container and intro_interval_container. Both will perform interval arithmetic using an interval_container, the first to solve a goal, the second to bring the bound of an expression into the local context.

To make this work, we will recreate the existing interval tactic, replacing computations that have already been performed with the corresponding values in the interval_container structure. A global overview of the interval tactic procedure has already been given in Section 3.1.1. We have skipped over an intermediate step there. The implication

$$\mathsf{f}(I)(\mathbf{x}) \leq 0 \implies f(x) \leq 0$$

is proven with the intermediate step

$$\mathsf{f}(I)(\mathbf{x}) \leq 0 \implies \mathsf{f}(\overline{\mathbb{R}})(x) \leq 0 \implies f(x) \leq 0,$$

where $\overline{\mathbb{R}}$ is the extended real number line. This is done because the domain I of intervals of floating-point numbers includes infinite intervals. We have seen SFBI2.nan (Not A Number) used this way in the previous section. Because of this NaN value, floating-point arithmetic is similar to arithmetic on the extended real number $\overline{\mathbb{R}}$ line. This intermediate step makes the theorems easier to handle.

Note that our interval_container contains a proof that $\mathsf{f}(\mathbb{R}) = f$. The interval package already provides proofs of the form $\mathsf{f}(\overline{\mathbb{R}}) \leq 0 \implies \mathsf{f}(\mathbb{R}) \leq 0$. We can therefore put these two steps into one, like in the following example.

```
Lemma prog_suff_gt_zero {U : Type} (container : interval_container) : ∀ (v : U),
  Private.A.check_p Private.A.positive_check (List.nth 0
    (eval_ext (evalProg container) (map Xreal (allVars container v))) Xnan) →
    term container v > 0.
Proof.
intros.
rewrite -progValid.
by apply Private.xreal_to_positive.
Qed.
```

In this example, there is a Private.A.check_p Private.A.positive_check. The interval package defines a record called check. A check contains two compatible predicates P_i and P_r: one for intervals and one for extended real numbers, together with a proof that they are compatible. By compatible, we mean that for all $y \in \overline{\mathbb{R}}$ and $\mathbf{y} \in I$, we have:

$$y \in \mathbf{y} \implies P_i(\mathbf{y}) \implies P_r(y)$$

One of these checks is Private.A.positive_check, for which $P_i(\mathbf{y}) := 0 < \underline{\mathbf{y}}$ and $P_r(y) := 0 < y$. Of course, if $y \in \mathbf{y}$, we have $\underline{\mathbf{y}} \leq y$, and so these statements are compatible.

The lemma aims to prove that $\mathsf{f}(\overline{\mathbb{R}})(v) > 0$ implies $f(v) > 0$. In the proof of this lemma the rewrite rewrites term container v, which is $f(v)$. Remember that progValid is the proof that $f = \mathsf{f}(\mathbb{R})$. The goal after rewriting is then equal to $\mathsf{f}(\mathbb{R})(v) > 0$, while the assumption is that $\mathsf{f}(\overline{\mathbb{R}})(v) > 0$. That our assumption implies our goal was shown in the interval package, and we can use this result to conclude.

Similar lemmas were proven for the other comparisons with 0. We will use these inside the final use_interval_container tactic.

The following Ltac contains the general method for using an interval_container to prove a check:

```
Ltac use_interval_container_aux container var lemma check :=
  apply (lemma _ container var) ;
  let allbounds:= get_all_bounds container var in
  change (map Xreal (allVars container var)) with
         (map Private.A.xreal_from_bp allbounds) ;
  let precision := (eval vm_compute in (Private.prec_of_nat 256%nat)) in
  refine (Private.interval_helper_evaluate
          allbounds check (evalProg container) precision 0%nat _);
  vm_cast_no_check (refl_equal true).
```

We will use a nonpositivity check to demonstrate what the Ltac does. We start with the goal $f(x) \leq 0$, where $\mathsf{f}(\mathbb{R})(x) = f(x)$. After applying the lemma, our goal is now $\mathbf{f}(\overline{\mathbb{R}})(x) \leq 0$. We will now compute the bounds on x, which should be available in the local context. The `change` tactic then prepares the goal to match the `Private.interval_helper_evaluate` lemma. This lemma contains the implication $\mathsf{f}(I)(\mathbf{x}) \leq 0 \implies \mathsf{f}(\overline{\mathbb{R}})(x) \leq 0$. The `change` is needed for the lemma to automatically see that $x \in \mathbf{x}$.

Note that we are always using 256-bit precision here. This could be improved, by making this a configurable parameter.

After the lemma has been applied, our goal is $\mathsf{f}(I)(\mathbf{x}) \leq 0$. In the general `check` form, this is written as $P_i(\mathsf{f}(I)(\mathbf{x})) =$ `true`. Now $\mathsf{f}(I)(\mathbf{x})$ has computational meaning, and can be reduced to some interval. The interval check P_i maps this interval to a boolean value, saying whether the interval satisfies the check or not. After this reduction has been made, we hope that the goal `true` $=$ `true` remains, since this is trivial to prove. To that end, we apply the `vm_cast_no_check` tactic. 'no check' defers the reduction checking to be done at `Qed`., which saves time. The last line basically means: the type of our goal is convertible to `true` $=$ `true`, which we can prove with the reflectiveness of equality.

The general procedure just described is used by the following `use_interval_container` tactic. This tactic tries to match the goal to one of the existing checks, and then runs the auxilliary tactic with this check.

```
Ltac use_interval_container container :=
let rawTerm := (eval cbn in (term container)) in
lazymatch goal with
  | ⊢ rawTerm ?v ≤ 0 ⇒
    use_interval_container_aux container v prog_suff_le_zero nonpositive_check
  | ⊢ rawTerm ?v < 0 ⇒
    use_interval_container_aux container v prog_suff_lt_zero negative_check
  | ⊢ rawTerm ?v > 0 ⇒
    use_interval_container_aux container v prog_suff_gt_zero
                                                    Private.A.positive_check
  | ⊢ rawTerm ?v ≥ 0 ⇒
    use_interval_container_aux container v prog_suff_ge_zero nonnegative_check
  | ⊢ ?a ≤ rawTerm ?v ≤ ?b ⇒
    let a_float := Private.get_float a in
    let b_float := Private.get_float b in
    let float_range := constr:(Private.I.bnd a_float b_float) in
    change (contains (Private.I.convert float_range)
                     (Xreal (term container v)));
    use_interval_container_aux container v prog_suff_contains
                                           (Private.A.subset_check float_range)
  | ⊢ _ ⇒ fail "Goal does not match easy comparison with" rawTerm
end.
```

Note that goals $a \leq f(x) \leq b$ require some more work. They are only excepted if both a and b can directly be converted to floats. This requirement makes life a bit easier, since the terms a and b could, in the worst case, depend on v. We then change the goal to match a subset check, and run the general procedure with this subset check.

We need to be able to handle such goals if we want to get proofs of bounds in our local context, for later use. This is often quite useful for debugging purposes, or if we need further processing. This introduction of bounds to our local context is performed in the following `intro_interval_container` tactic.

```
Ltac intro_interval_container container v :=
  let allbounds := get_all_bounds container v in
  let mappedbounds := constr:(map Private.A.interval_from_bp allbounds) in
  let prec := (eval vm_compute in (Private.prec_of_nat 256%nat)) in
  let calculation_term := constr:(List.nth 0
        (Private.A.BndValuator.eval prec (evalProg container) mappedbounds)
                                Private.I.nai) in
  let b := eval vm_compute in calculation_term in
```

After these let statements, `b` is equal to the reduced interval of $\mathsf{f}(I)(\mathbf{x})$. This is now a floating point

interval, while we would like a real interval in our local context. This is done in the next part: we check whether the floating point interval corresponds to some real interval, and then convert the type of the hypothesis. We then move the hypothesis back to the goal using revert, for later processing.

```
match goal with
| ⊢ ?P ⇒
  refine (
   (_ : contains (Private.I.convert b) (Xreal (term container v)) → P)
    _) ; [
  let H := fresh "H" in
  intro H ;
  match eval cbv -[IZR Rdiv] in (Private.I.convert b) with
  | Ibnd ?l ?u ⇒
    match l with
    | Xreal ?l ⇒
      match u with
      | Xreal ?u ⇒ change (l ≤ term container v ≤ u)%R in H
      end
    end
  | _ ⇒ fail "Finding bounds unsuccessful"
  end ;
  let rawTerm := (eval cbn in (term container)) in
  change (term container) with rawTerm in H;
  revert H |
```

Note that the refine clause has two gaps. This will create two subgoals. We apply several tactics to manipulate the first subgoal, as described above, but we do not solve this goal. Solving this goal is up to the user, who now has additional hypothesis in the local context. We do solve the second subgoal, which is a proof obligation to show that $\underline{b} \leq f(x) \leq \overline{b}$. By doing the computation, we have 'lost' the information that calculation_term = b, and thus that $\underline{b} \leq f(x) \leq \overline{b}$. For now, we fulfill this proof obligation using the use_interval_container tactic. Some clever substitutions could probably remove this repeated calculation, but this will suffice for now.

```
  match eval cbv -[IZR Rdiv] in (Private.I.convert b) with
  | Ibnd ?l ?u ⇒
    match l with
    | Xreal ?l ⇒
      match u with
      | Xreal ?u ⇒ change (l ≤ term container v ≤ u)%R
      end
    end
  | _ ⇒ fail "Finding bounds unsuccessful"
  end ;
  let rawTerm := (eval cbn in (term container)) in
  change (term container) with rawTerm ;
  use_interval_container container ]
end.
```

Similar introduction tactics were created for when we just need the lower or upper bound in the context, not the proof of it. This is easier, since this does not oblige us to prove such a bound.

The following tactics (along with some variations) are made available to other Coq files.

```
Tactic Notation "build_container"
  constr(term) constr(vars) constr(element) constr(boundlist) :=
  IntervalContainer.build_container term vars element boundlist
                          Private.extract_algorithm false.
```

The false signifies not to print the computations.

```
Tactic Notation "interval_container" constr(container) :=
              IntervalContainer.use_interval_container container.
Tactic Notation "interval_container_intro" constr(container) constr(v) :=
              IntervalContainer.intro_interval_container container v ; intro.
```

The `interval_container` tactic uses a provided container to (try to) solve the goal, while the `interval_container_intro` tactic uses provided container to put a (proof of a) bound on the term in the local context.

3.2.2. Multi-dimensional mean value theorem

We need a multi-dimensional mean value theorem to use the gradient for providing bounds on our polynomials. What we need is something like:

Theorem 5. *Let $f : \mathbb{R}^n \to \mathbb{R}$ and x, y in $\mathbb{R}^n$. If f is differentiable on $\mathbb{R}^n$, then there exists $c \in \mathbb{R}$ with the property that $0 < c < 1$ and*

$$f(y) - f(x) = \nabla f((1-c)x + cy) \cdot (y - x).$$

Note that $\cdot$ is an inner product here. We need to mold this theorem into a form compatible with Coquelicot, so that we can utilize its infrastructure.

Remember that differentiability in Coquelicot is defined using Fréchet derivatives. This will cause the inner product to be replaced by an application of a linear operator. Furthermore, it is inconvenient to work with $\mathbb{R}^n$ in Coquelicot. The proof of Theorem 5 also does not use specific properties of $\mathbb{R}^n$, so it should be no problem to replace $\mathbb{R}^n$ with a normed module U over $\mathbb{R}$.

Recall the one dimensional version of the mean value theorem:

Theorem 6. *Let $f : \mathbb{R} \to \mathbb{R}$ and x, y in $\mathbb{R}$ with $x \leq y$. Suppose f is differentiable on (x, y) and continuous on $[x, y]$. Then there exists some $c \in (x, y)$ with*

$$f'(c) = \frac{f(y) - f(x)}{y - x}.$$

The proof of Theorem 5 uses Theorem 6. Looking at the conditions of Theorem 6, it seems that the assumption of f being differentiable on $\mathbb{R}^n$ can also be weakened. We should only need that f is differentiable on the open line between x and y, and continuous on the closed line. We know that differentiability implies continuity, so we actually only need continuity at the endpoints x and y. With these considerations in mind, we can start proving in Coq.

To start off, we prove a preliminary proposition: the mean value theorem on $\mathbb{R}$ between 0 and 1. This is done because the exact statement of Coquelicot's mean value theorem on $\mathbb{R}$ is somewhat hard to manipulate, because instead of requiring $x \leq y$, x is replaced with $\min(x, y)$ and y with $\max(x, y)$. This makes it harder to work with when proving a more general version of the mean value theorem. The Coq standard library also proves a version of the mean value theorem, but uses its own notion of differentiability. We would like to use Coquelicot's notion of differentiability, so we use their mean value theorem.

Coquelicot's notion of differentiability on functions of type `R → R` is called `is_derive`. The statement `is_derive f x v` means that the derivative of function `f` at point `x` is equal to value `v`. Furthermore, `continuity_pt f x` means that the function `f` is continuous at point `x`.

In the following statement, we have replaced `f 1 - f 0 = df c * (1 - 0)` with `f 1 - f 0 = df c`.

```
Proposition MVT_gen_R01 (f : R → R) (df : R → R) :
    (∀ x : R, 0 < x < 1 → is_derive f x (df x)) →
    (∀ x : R, 0 ≤ x ≤ 1 → continuity_pt f x) →
    ∃ c : R, 0 ≤ c ≤ 1 ∧ f 1 - f 0 = df c.
Proof.
intros.
```

Because of the mentioned replacement, we are not able to directly apply Coquelicot's mean value theorem `MVT_gen`. Rewriting will also not work, since we cannot access the term `c` - it is contained in the existential statement, and not in the local context. To overcome this, we show it is enough to prove a term which is compatible with `MVT_gen`. To make it compatible, we will also replace 0 and 1 with $\min(0, 1)$ and $\max(0, 1)$ respectively. These functions are represented in Coq by `Rmin` and `Rmax`.

```
enough (∃ c : R, Rmin 0 1 ≤ c ≤ Rmax 0 1 ∧ f 1 - f 0 = df c * (1 - 0))
    as [c [cH1 cH2]].
∃ c; split.
- by rewrite (Rmin_left _ _ Rle_0_1) (Rmax_right _ _ Rle_0_1) in cH1.
- rewrite cH2; ring.
```

The `Rmin_left` rewrites $\min(0,1)$ to 0 if given a proof that $0 \leq 1$. Such a proof can be found in `Rle_0_1`. Similarly, `Rmax_right` rewrites $\max(0,1)$ to 1. For the second goal, we need to prove $f(1) - f(0) = f'(c)$, while we have that $f(1) - f(0) = f'(c) \cdot (1 - 0)$. This is a valid ring equality, so we use `ring` to conclude.

We can now apply Coquelicot's `MVT_gen`. The new proof goals generated correspond with our hypothesis, except for the fact that they again contain `Rmin` and `Rmax`. We rewrite these in the same way and conclude.

```
apply MVT_gen;
by rewrite (Rmin_left _ _ Rle_0_1) (Rmax_right _ _ Rle_0_1).
Qed.
```

We will now remove superfluous conditions from `MVT_gen_R01` in our next proposition. As we said, the differentiability of `f` on the open interval implies continuity on the open interval. This means only continuity on the endpoints is required.

Note also that we write `continuous` instead of `continuity_pt`. The former is Coquelicot's notion of continuity, the latter that of Coq's standard library. These are equivalent, which is proven by `continuity_pt_continuous`.

```
Proposition MVT_gen_R01' (f : R → R) (df : R → R) :
    (∀ x : R, 0 < x < 1 → is_derive f x (df x)) →
    continuous f 0 → continuous f 1 →
    ∃ c : R, 0 ≤ c ≤ 1 ∧ f 1 - f 0 = df c.
Proof.
intros dfDeriv ? ?.
apply MVT_gen_R01; [ assumption | ].
```

At this point, we need to prove that `∀ x : R, 0 ≤ x ≤ 1 → continuity_pt f x`. We will do this by distinguishing the cases where $x = 0$, $0 < x < 1$ and $x = 1$. The first and the last case are hypotheses, the middle case is true because f is differentiable there. The `ex_derive_continuous` term states that the implication `(∃ v, is_derive f x v) → continuous f x` holds: differentiability of `f` at `x` implies that `f` is continuous at `x`.

```
intros x [[xgt0 | xeq0] xle1].
- destruct xle1 as [xlt1 | xeq1].
```

case $0 < x < 1$:

```
  * apply continuity_pt_continuous.
    apply (ex_derive_continuous f).
    ∃ (df x).
    by apply dfDeriv.
```

case $x = 1$:

```
  * apply continuity_pt_continuous.
    by rewrite xeq1.
```

case $x = 0$:

```
- apply continuity_pt_continuous.
  by rewrite -xeq0.
Qed.
```

We are now ready to prove the multidimensional mean value theorem. Recall that we replaced $\mathbb{R}^n$ with an abstract normed module U over $\mathbb{R}$. The main idea is to use our function $f : U \to \mathbb{R}$ to create a function $g : [0,1] \to \mathbb{R}$. This g is defined as

$$g(r) = f((1 - r)x + ry).$$

This function has the property that $g(0) = f(x)$ and $g(1) = f(y)$. Furthermore, the derivative of g is

$$g'(r) = F((1 - r)x + ry)(y - x).$$

Here we have written $F(a)(b)$ for the Fréchet derivative of f at a (which is a linear operator), applied to the value b. We used the chain rule for Fréchet derivatives here (see Section 1.4.2). We can write g as $f \circ h$, where $h : \mathbb{R} \to U$ is defined as $h(r) = (1-r)x + ry$. Intuitively, the derivative of h would be $y - x$, so the Fréchet derivative of h is the linear operator $t \mapsto t \cdot (y - x)$. Thus the Fréchet derivative of $g = f \circ h$ at r is

$$\begin{aligned} F(h(r)) \circ (t \mapsto t(y-x)) &= t \mapsto F(h(r))(t(y-x)) \\ &= t \mapsto t\, F(h(r))(y-x) \end{aligned}$$

where we used that $F(h(r))$ is a linear operator to move the multiplication with t outside the operator. Since g is a function of $\mathbb{R} \to \mathbb{R}$, we get that the (usual) derivative of g is equal to $g'(r) = F(h(r))(y-x)$.

We then apply the mean value theorem on $\mathbb{R}$ to g, and map the results back to f. The statement of our theorem is thus as follows.

Theorem 7 (MVT_general)**.** *Let U be a normed module over $\mathbb{R}$, x and y elements of U, $f : U \to \mathbb{R}$ a function, and F a function sending elements of U to linear operators of U to $\mathbb{R}$. Suppose the following conditions hold:*

- *For every $0 < t < 1$, the Fréchet derivative of f at $(1-t)x + ty$ is equal to (the linear operator) $F((1-t)x + ty)$;*
- *f is continuous at x;*
- *f is continuous at y.*

Then there exists some $0 \le c \le 1$ such that $f(y) - f(x) = F((1-c)x + cy)(y - x)$.

The Fréchet derivative assumption in Coq is stated as `filterdiff f (locally x) A`. This means precisely that the operator `A` is the Fréchet derivative of `f` at `x`. This `filterdiff` is only defined on `NormedModules`, so the assumption that U is a normed module is indeed required.

The name `filterdiff` stems from the fact that Fréchet derivatives in Coquelicot are defined using the *filter* concept from topology. In this context, `locally x` is the neighbourhood filter of `x`. Note that `filterdiff f (locally x) A` does not mean that `A` is the Fréchet derivative of `f` in a neighbourhood around `x`; rather, it means that for every $\epsilon > 0$ there is a neighbourhood around x on which the error that `A` makes as a linear approximation of `f` is smaller than ϵ. We omit the details of this construction, since it is somewhat involved and we do not need an intimate understanding of filters to prove what we need.

```
Theorem MVT_general (f : U → R) (x y : U) (F : U → U → R) :
  (∀ t : R, 0 < t < 1 →
     filterdiff f (locally (plus (scal (minus one t) x) (scal t y)))
                          (F (plus (scal (minus one t) x) (scal t y)))) →
  continuous f x →
  continuous f y →
  ∃ c : R, 0 ≤ c ≤ 1 ∧
       f y - f x = F (plus (scal (minus one c) x) (scal c y)) (minus y x).
Proof.
```

Notice the type of our Fréchet derivative `F`: it sends values of `U` to operators `U` → `R`. The plus, scal, minus are all operations in the normed module `U`. We then define the function g:

```
intros Fderiv ? ?; cbn.
set (h := fun r ⇒ plus (scal (1 - r) x) (scal r y)).
set (g := fun r ⇒ f (h r)).
set (g':= fun r ⇒ F (h r) (minus y x)).
```

Again, we cannot rewrite statements depending on the existential variable `c`, since `c` is not in the context and cannot be moved there yet. We therefore show it is enough to prove another existential statement, which happens to be equal to the conclusion of `MVT_gen_R01`.

```
enough (∃ c, 0 ≤ c ≤ 1 ∧ g 1 - g 0 = g' c) as [ c [cH1 cH2]].
∃ c; split; [ assumption | ].
fold (1 - c) (h c) (g' c).
rewrite -cH2; unfold g, h, Rminus.
by rewrite Rplus_opp_r Ropp_0 Rplus_0_r scal_one scal_zero_l plus_zero_l
                                         scal_one scal_zero_l plus_zero_r.
```

The rewrites above take care of proving the equality `f y - f x` = `g 1 - g 0`.

Before we continue, we will first show that `h` is Fréchet differentiable everywhere, with Fréchet derivative at point r_0 equal to

$$H(r_0)(h) = h \cdot (y - x).$$

We `assert` this now, since we will use this fact in multiple parts of the proof. Note that the expression $h \cdot (y - x)$ does not depend on r_0. This is equal to saying: the closest linear approximation of our function does not depend on where we evaluate our function, ie `h` is a linear function with constant (regular) derivative.

```
assert (∀ r0, filterdiff h (locally r0) (fun r ⇒ scal r (minus y x))) as hDeriv.
  intros.
  unfold h, minus.
  rewrite plus_comm.
  apply (filterdiff_ext_lin _ (fun r ⇒ plus (scal (-r) x) (scal r y)));
          [ | intros; by rewrite scal_distr_l scal_opp_r -scal_opp_l ].
  apply (filterdiff_plus_fct (fun r ⇒ scal (1 - r) x)).
  - apply (filterdiff_ext (fun r : R ⇒ minus (scal 1 x) (scal r x)));
          [ intros; unfold minus; by rewrite scal_distr_r -scal_opp_l | ].
    apply (filterdiff_ext_lin _ (fun r : R ⇒ minus zero (scal r x)));
           [ | intros; unfold minus; by rewrite plus_zero_l -scal_opp_l].
    apply (filterdiff_minus_fct (fun r ⇒ scal 1 x)).
    * apply (filterdiff_const (scal 1 x)).
    * apply filterdiff_linear; apply (is_linear_scal_l x).
  - apply filterdiff_linear; apply (is_linear_scal_l y).
```

The above proof splits up `h` in three parts: the constant part of `scal 1 x`, and the two variable parts `scal (-r) x` and `scal r y`. The derivative of `h` is then the sum of the derivatives of these expressions.

To show this is true, we sometimes need to rewrite the function or the derivative. However, we do not have functional extensionality: even if functions agree on all values, they are not provably equal. However, Coquelicot shows that it is enough for functions or derivatives to agree on all values to allow for a substitution. These are the various `filterdiff_ext_lin` and `filterdiff_ext` applications.

Once the necessary rewrites have been performed, we have theorems giving us the derivatives of each of the simpler expressions, and we are done.

Remember that our current goal was an existential statement matching the conclusion of the intermediate result `MVT_gen_R01'`. To continue, we apply this theorem and prove the required hypotheses.

```
apply MVT_gen_R01'.
```

We now have to show that g' is differentiable on $(0, 1)$:

```
- intros.
  unfold is_derive, g, g'.
  apply (filterdiff_ext_lin _
      (fun r0 : R ⇒ F (plus (scal (1 - x0) x) (scal x0 y)) (scal r0 (minus y x)))).
  * apply (filterdiff_comp' _ _ _ (fun r0 : R ⇒ scal r0 (minus y x)) _).
    by apply hDeriv.
    by apply Fderiv.
  * intros.
    rewrite (linear_scal); [ trivial | ].
    by apply Fderiv.
```

The `filterdiff_ext_lin` is necessary since the `is_derive` condition slightly rewrites our original derivative g'. This is because `is_derive f x diff` means that `diff` is the slope of the best linear approximation of `f` at `x`. This is thus the same as `filterdiff f (locally x) (fun r ⇒ scal r diff)`. We want to use the chain rule on Fréchet derivatives to continue. To do so, our Fréchet derivative needs to be of the form `fun r ⇒ lf (lg r)`, while it is now of the form `fun r ⇒ scal r (lf c)`. We can, however, rewrite `scal r (lf c)` to `lf (scal r c)`, since `lf` is a linear function. This allows us to finish this part of the proof using the chain rule.

We now have to show that g is continuous at 1.

```
- unfold g.
  apply continuous_comp.
  * apply (filterdiff_continuous h _).
    eexists.
    apply hDeriv.
  * unfold h.
    by rewrite Rminus_0_r scal_zero_l scal_one plus_zero_r.
```

This can be done because g is a composition of continuous functions. The function h is continuous because it is differentiable. The filterdiff_continuous lemma requires us to show that a derivative exists, which our earlier assertion enables us to.

The last part of the proof is to show that g is continuous at 0. This is basically the same as for at 1:

```
- unfold g.
  apply continuous_comp.
  * apply (filterdiff_continuous h _).
    eexists.
    apply hDeriv.
  * unfold h, Rminus.
    by rewrite Rplus_opp_r scal_zero_l scal_one plus_zero_l.
Qed.
```

Of course, the conditions for the mean value theorem are also satisfied if the function is differentiable on the closed line. This sometimes requires less work to verify, so the following lemma is also shown:

```
Lemma MVT_easy (f : U → R) (x y : U) (F : U → U → R) :
  (∀ t : R, 0 ≤ t ≤ 1 →
      filterdiff f (locally (plus (scal (minus one t) x) (scal t y)))
                        (F (plus (scal (minus one t) x) (scal t y)))) →
    ∃ c : R, 0 ≤ c ≤ 1 ∧
             f y - f x = F (plus (scal (minus one c) x) (scal c y)) (minus y x).
```

We have omitted the proof, since it follows the same lines to prove continuity using differentiability as the previous theorem.

A note is in order on the exact statement of MVT_general. Note that the conclusion of the theorem says that there exists some $c \in \mathbb{R}$ with $0 \leq c \leq 1$ — instead of the proposed $0 < c < 1$. Evaluating the expression for $\mathtt{c} = 0$ or $\mathtt{c} = 1$ will result in the right hand side being fd x or fd y respectively. This is strange, since the assumptions do not even require that fd x or fd y are really the Fréchet derivatives of f at x or y: they could be any value.

This happened because Coquelicot's MVT_gen lemma, the mean value theorem on $\mathbb{R}$ was formulated to also be applicable in the degenerate case where the interval is a singleton. In that case, $c \in (a, a)$ is of course not possible, but $c \in [a, a]$ is possible. By loosening the requirements on c we get an expression that is true for this degenerate case. Note that the conclusion for the non-degenerate case is still true: we only prove that we must have $0 \leq c \leq 1$, while the stronger statement $0 < c < 1$ is actually true in the non-degenerate case.

We could retrace this exact dependency and remove it, but the current statement suffices for our use.

With our mean value theorem ready, we can now use it to prove the result we actually need. Coquelicot calls this 'bounded variation', and although this term also has a different meaning in analysis, we will use Coquelicot's terminology.

```
Lemma bounded_variation_general (f : U → R) (x y : U) (F : U → U → R) (D : R) :
  (∀ t : R, 0 < t < 1 →
      filterdiff f (locally (plus (scal (minus one t) x) (scal t y)))
                          (F (plus (scal (minus one t) x) (scal t y)))) →
  (∀ t : R, 0 ≤ t ≤ 1 →
    ∀ v : U,
      norm (F (plus (scal (minus one t) x) (scal t y)) v) ≤ D * (norm v)) →
  continuous f x →
  continuous f y →
  norm(f y - f x) ≤ D * (norm (minus y x)).
```

The proof is now quite easy: once we substitute our mean value theorem witness, the proof is finished nearly directly. Note that instead of bound on the (vector) norm of the gradient along the line, we require a bound on the operator norm of the Fréchet derivatives along the line.

```
Proof.
intros Fderiv bounded_opnorm cont_x cont_y.
destruct (MVT_general f x y F Fderiv cont_x cont_y) as [c [c_bound cH]].
rewrite cH.
by apply bounded_opnorm.
Qed.
```

To accompany the 'easy' version of the mean value theorem, we have also shown an easy version of bounded variation. This is exactly Theorem 2 mentioned in the first chapter.

```
Lemma bounded_variation_easy (f : U → R) (x y : U) (fd : U → U → R) (D : R) :
  (∀ t : R, 0 ≤ t ≤ 1 →
    filterdiff f (locally (plus (scal (minus one t) x) (scal t y)))
                     (fd (plus (scal (minus one t) x) (scal t y))) ∧
    ∀ v : U,
      norm (fd (plus (scal (minus one t) x) (scal t y)) v) ≤ D * (norm v)) →
    norm(f y - f x) ≤ D * (norm (minus y x)).
```

The proof is omitted. Note that we are able to pack all conditions into one statement.

3.2.3. Proving that ∇f is the derivative of f

To fulfill the conditions of our `MVT_general`, we need to give a proof that the Fréchet derivative of our polynomial p is some expression for all x on some line. Since the polynomial will be a function of $\mathbb{R}^3$ to $\mathbb{R}$, the Fréchet derivative will be equal to $h \mapsto h \cdot \nabla p$, where $\cdot$ denotes the inner product. The corresponding expression in Coq will be something like `filterdiff f x (grad_p x)`. An explicit proof of this, involving epsilons and deltas, is unwanted because of the size of our polynomial p. Coquelicot provides automation for proving derivatives of functions $\mathbb{R} \to \mathbb{R}$. This means we can verify the validity of partial derivatives of p. Existence of partial derivatives will give us existence of a derivative under the right conditions. In this section, we will prove a theorem which will allow us to do this for our polynomial.

In Coquelicot, $\mathbb{R}^3$ is represented as `(R * R) * R`. It is the product type of $\mathbb{R}^2$ and $\mathbb{R}$, and Coquelicot knows this is a normed module together with the product norm. If we state our theorem to apply to cases of the form `U * R`, where `U` is some normed module over $\mathbb{R}$, we can apply it to $\mathbb{R}^3$. After applying it once, we will need to provide a Fréchet derivative of a function $\mathbb{R}^2 \to \mathbb{R}$, and after applying it twice we arrive at the easier case $\mathbb{R} \to \mathbb{R}$.

A note is in order on what partial derivatives mean in this context of Fréchet derivatives. We say a function $f : U \times \mathbb{R} \to \mathbb{R}$ is partially Fréchet differentiable in the first coordinate at a point (x_0, y_0) if there is an operator $A : U \to \mathbb{R}$ so that the function $f_{y_0}(x) = f(x, y_0)$ is Fréchet differentiable at x_0 with Fréchet derivative A. For partial differentiability in the second coordinate we can resort to the usual notion of differentiability. We are now ready to state the theorem we wish to prove

Theorem 8 (partials_imply_gradient)**.** *Let $f : U \times \mathbb{R} \to \mathbb{R}$ be a function, $(x_0, y_0) \in U \times \mathbb{R}$ and U a normed module over $\mathbb{R}$. Suppose we have the following:*

- *F_x is the partial Fréchet derivative in the first coordinate of f at (x_0, y_0);*
- *There exists a $\delta > 0$ so that $\frac{\partial f}{\partial y}$ exists in the ball of radius δ around (x_0, y_0);*
- *$\frac{\partial f}{\partial y}$ is continuous at (x_0, y_0).*

Then f is Fréchet differentiable at (x_0, y_0) and its Fréchet derivative is equal to

$$F(h) : U \times \mathbb{R} \to \mathbb{R}$$

$$(h_1, h_2) \mapsto F_x(h_1) + h_2 \frac{\partial f}{\partial y}(x_0, y_0)$$

Note that Theorem 8 states that existence of the partial Fréchet derivative in the first coordinate at one point is sufficient, while we do need stronger conditions on the partial derivative in the second coordinate. Translated to Coq, the theorem becomes:

```
Theorem partials_imply_gradient :
  ∀ (f : U * R → R) (x0 : U) (y0 : R) (Fx : U → R) (fy : U * R → R),
  filterdiff (fun x : U ⇒ f (x, y0)) (locally x0) Fx →
  locally (x0,y0) (fun v : U * R ⇒ is_derive (fun y ⇒ f (v.1, y)) (v.2) (fy v)) →
  continuous fy (x0,y0) →
  filterdiff f (locally (x0, y0))
      (fun t : U * R ⇒ plus (Fx t.1) (scal (t.2) (fy (x0,y0)))).
```

As before, the `filterdiff` expression signifies that `Fx` is indeed the partial Fréchet derivative in the first coordinate at (`x0`, `y0`). The `locally` (`x0`,`y0`) `_` statement means that in a neighbourhood around (`x0`,`y0`), $\frac{\partial f}{\partial y}$ is given by `fy`.

This theorem has been proven in Coq, but we will explain the proof with words.

Proof. We need to prove that, given some $\epsilon > 0$, there exists some $\delta > 0$, so that for every pair $(h_1, h_2) \in U \times \mathbb{R}$ with $\|(h_1, h_2)\| < \delta$ we have

$$|f(x_0 + h_1, y_0 + h_2) - f(x_0, y_0) - F(h_1, h_2)| \leq \epsilon \|(h_1, h_2)\|.$$

We will first add a term $f(x_0+h_1, y_0)$ to the left hand side, so we can use that F_x is the partial Fréchet derivative of f in the first coordinate. Then, by the triangle inequality, we have:

$$\begin{aligned} &|f(x_0 + h_1, y_0 + h_2) - f(x_0, y_0) - F(h_1, h_2)| \\ &\qquad \leq \left| f(x_0 + h_1, y_0 + h_2) - f(x_0 + h_1, y_0) - h_2 \frac{\partial f}{\partial y}(x_0, y_0) \right| + |f(x_0 + h_1, y_0) - f(x_0, y_0) - F_x(h_1)|. \end{aligned}$$

We can find some δ_1 so that $|f(x_0 + h_1, y_0) - f(x_0, y_0) - f_x(h_1) \leq \frac{\epsilon}{2}\|h_1\|$ whenever $\|h_1\| < \delta$ because of the Fréchet differentiability of f in the first coordinate. We can obtain a bound for the first term by rewriting it in the following way:

$$\begin{aligned} &\left| f(x_0 + h_1, y_0 + h_2) - f(x_0 + h_1, y_0) - h_2 \frac{\partial f}{\partial y}(x_0, y_0) \right| \\ &\qquad = \left| f(x_0 + h_1, y_0 + h_2) - f(x_0 + h_1, y_0) - (y_0 + h_2 - y_0) \frac{\partial f}{\partial y}(x_0, y_0) \right| \\ &\qquad = \left| \left(f(x_0 + h_1, y_0 + h_2) - (y_0 + h_2) \frac{\partial f}{\partial y}(x_0, y_0) \right) - \left(f(x_0 + h_1, y_0) - y_0 \frac{\partial f}{\partial y}(x_0, y_0) \right) \right|. \end{aligned}$$

We now define the function $g(t) = f(x_0 + h_1, t) - t\frac{\partial f}{\partial y}(x_0, y_0)$, so that we can rewrite the last term as

$$\left| \left(f(x_0 + h_1, y_0 + h_2) - (y_0 + h_2) \frac{\partial f}{\partial y}(x_0, y_0) \right) - \left(f(x_0 + h_1, y_0) - y_0 \frac{\partial f}{\partial y}(x_0, y_0) \right) \right| = |g(y_0+h_2) - g(y_0)|.$$

It is easy to see that we can apply bounded variation to obtain a bound on our expression. Note that g is differentiable in a neighbourhood of y_0 since f is partially differentiable in the second coordinate in a neighbourhood of (x_0, y_0). This means we have

$$g'(t) = \frac{\partial f}{\partial y}(x_0 + h_1, t) - \frac{\partial f}{\partial y}(x_0, y_0)$$

for $t \in B(y_0, \delta_2)$, for some $\delta_2 > 0$. Now g' has a convenient form, since it allows us to use the continuity of $\frac{\partial f}{\partial y}$ at (x_0, y_0). We can pick $\delta_3 > 0$ such that if $\|(h_1, h_2)\| < \delta_3$, then

$$\left| \frac{\partial f}{\partial y}(x_0 + h_1, y_0 + h_2) - \frac{\partial f}{\partial y}(x_0, y_0) \right| < \frac{\epsilon}{2}$$

and so by bounded variation we get

$$|g(y_0 + h_2) - g(y_0)| \leq \frac{\epsilon}{2} \cdot |h_2|.$$

We define $\delta = \min(\delta_1, \delta_2, \delta_3)$. For $\|(h_1, h_2)\| < \delta$, we have

$$\begin{aligned}|f(x_0+h_1, y_0+h_2) - f(x_0, y_0) - f'(h_1, h_2)| &\leq \frac{\epsilon}{2}|h_2| + \frac{\epsilon}{2}\|h_1\| \\ &\leq \frac{\epsilon}{2} \cdot (\max(|h_2|, \|h_1\|) + \max(|h_2|, \|h_1\|)) \\ &= \epsilon \cdot \max(|h_2|, \|h_1\|) \\ &\leq \epsilon\|(h_1, h_2)\|\end{aligned}$$

as required. We also need that f' is a linear function, but this is easy to prove. □

One of the conditions of the `partials_imply_gradient` theorem is continuity of the partial derivative in *both* coordinates. We could prove continuity by proving differentiability in both coordinates. This would then take another application of `partials_imply_gradient`, which would again require a proof of the continuity of the partial derivative of the partial derivative, etcetera. This approach would work for our polynomials, but would require large proof terms since the highest powers appearing in our polynomials is 26.

We devised another approach to tackle this issue. Another sufficient condition was found which provides continuity in both coordinates, and which is provable with available techniques.

Theorem 9 (locally_bounded_diff_implies_2d_continuous)**.** *Let $f : U \times \mathbb{R} \to \mathbb{R}$ be a function, U be a normed module over $\mathbb{R}$ and (x_0, y_0) coordinates in $U \times \mathbb{R}$. Suppose we have the following:*

- *f is continuous in the first coordinate at (x_0, y_0), i.e. $x \mapsto f(x, y_0)$ is continuous at x_0;*
- *There exists a $\delta_1 > 0$ so that $\frac{\partial f}{\partial y}$ exists in the ball of radius δ_1 around (x_0, y_0);*
- *There exists a $\delta_2 > 0$ and a $D > 0$ so that for $(u, v) \in B((x_0, y_0), \delta_2)$ we have*

$$\left|\frac{\partial f}{\partial y}(u, v)\right| < D.$$

Then f is continuous at (x_0, y_0).

We will first translate the statement to Coq, and then give a proof in words. The first two conditions can be translated using things we have seen before. Continuity is encoded with `continuous`, and a statement of the form `locally (x0,y0) _` can be used to encode the second condition. For the third condition we created a definition in Coq. The statement `locally_bounded f x` means

$$\exists M \in \mathbb{R}\ \exists \delta_2 > 0\ \forall y \in B(x, \delta_2) :\ \|f(y)\| < M,$$

i.e. there is an $M \in \mathbb{R}$ and a $\delta_2 > 0$ so that all y in the ball around x with radius δ_2 have $\|f(y)\| < M$. Note that we do not need to require $M > 0$ since this follows automatically from $0 \leq \|f(x)\| < M$, and $x \in B(x, \delta_2)$ for all $\delta_2 > 0$.

The statement in Coq thus becomes

```
Proposition locally_bounded_diff_implies_2d_continuous :
  ∀ (f: U * R → R) (x0 : U) (y0 : R) (fy : U * R → R),
  continuous (fun x ⇒ f (x, y0)) x0 →
  locally_bounded fy (x0, y0) →
  locally (x0, y0) (fun v : U * R ⇒ is_derive (fun y ⇒ f (v.1, y)) v.2 (fy v)) →
  continuous f (x0, y0).
```

We continue with the proof.

Proof. We need to prove that, given some $\epsilon > 0$, there exists some $\delta > 0$, so that for every pair $(h_1, h_2) \in U \times \mathbb{R}$ with $\|(h_1, h_2)\| < \delta$ we have

$$|f(x_0+h_1, y_0+h_2) - f(x_0, y_0)| < \epsilon.$$

As before, we introduce a term of $f(x_0 + h_1, y_0)$:

$$\begin{aligned}|f(x_0+h_1, y_0+h_2) - f(x_0, y_0)| &= |f(x_0+h_1, y_0+h_2) - f(x_0+h_1, y_0) + f(x_0+h_1, y_0) - f(x_0, y_0)| \\ &\leq |f(x_0+h_1, y_0+h_2) - f(x_0+h_1, y_0)| + |f(x_0+h_1, y_0) - f(x_0, y_0)|.\end{aligned}$$

Since f is continuous in the first coordinate, we are able to find some δ_3 for which $\|h_1\| < \delta_3$ implies $|f(x_0 + h_1, y_0) - f(x_0, y_0)| < \frac{\epsilon}{2}$. This gives us a bound on the second term.

For the first term, we will resort once again to bounded variation. Define $g(t) = f(x_0 + h_1, t)$. By requiring $\|(h_1, h_2)\| < \min(\delta_1, \delta_2)$, we get differentiability of g on the interval $[y_0, y_0 + h_2]$. The derivative of g is then equal to $g'(t) = \frac{\partial f}{\partial y}(x_0 + h_1, t)$. We also obtain that $|g'(t)| < D$ on $[y_0, y_0 + h_2]$. If we require that $|h_2| < \frac{\epsilon}{2D}$, then

$$\begin{aligned} |f(x_0 + h_1, y_0 + h_2) - f(x_0 + h_1, y_0)| &= |g(y_0 + h_2) - g(y_0)| \\ &\le D|h_2| \\ &< D \cdot \frac{\epsilon}{2D} = \frac{\epsilon}{2}. \end{aligned}$$

We define $\delta = \min(\delta_1, \delta_2, \delta_3, \frac{\epsilon}{2D})$. For $\|(h_1, h_2)\| < \delta$, we have:

$$\begin{aligned} |f(x_0 + h_1, y_0 + h_2) - f(x_0, y_0)| &\le |f(x_0 + h_1, y_0 + h_2) - f(x_0 + h_1, y_0)| + |f(x_0 + h_1, y_0) - f(x_0, y_0)| \\ &< \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon \end{aligned}$$

as required. □

We will give an example of how this theorem gives us continuity for some nasty functions. Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$, defined by

$$f(x, y) = \begin{cases} xy^2 \sin\left((xy)^{-1}\right) & \text{for } x \neq 0 \text{ and } y \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

Then f is continuous in the first variable at $(0, 0)$, and everywhere partially differentiable in the second variable, with derivative:

$$\frac{\partial f}{\partial y}(x, y) = \begin{cases} 2xy \sin\left((xy)^{-1}\right) - \cos\left((xy)^{-1}\right) & \text{for } x \neq 0 \text{ and } y \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\frac{\partial f}{\partial y}$ exists everywhere but is discontinuous at $(0, 0)$. Furthermore, $\frac{\partial f}{\partial x}(0, 0)$ does not exist. However, f does satisfy the conditions of the theorem, since $\frac{\partial f}{\partial y}$ is bounded around $(0, 0)$. This means that f is continuous at $(0, 0)$.

3.2.4. Lists of propositions

We need a meaningful way to talk about all or some propositions of a list being true. This will be useful when defining convex combinations in Section 3.2.5. We will need it there to be able to state that all coefficients are positive in a convex combination. It will also see heavy use in Section 3.3, when talking about gridpoints in cubes.

Let (P_n) be a list of propositions. We want to build terms like

$$\bigwedge_{i=1}^{S((P_n))} P_i \quad \text{and} \quad \bigvee_{i=1}^{S((P_n))} P_i,$$

the logical conjunction or disjunction of all elements of (P_n), where $S((P_n))$ denotes the size of list (P_n). The only note to make here is that we need a neutral element for the empty list. This will be `True` for conjunction and `False` for disjunction. They are neutral, since $P \wedge \texttt{True} \implies P$, and $P \vee \texttt{False} \implies P$.

```
Fixpoint all_valid (l : list Prop) :=
  match l with
  | nil ⇒ True
  | cons a l ⇒ a ∧ all_valid l
  end.
```

```
Fixpoint some_valid (l : list Prop) :=
  match l with
  | nil ⇒ False
  | cons a l ⇒ a ∨ some_valid l
  end.
```

all_valid and some_valid will be the conjunction and disjunction of a list of propositions, respectively. We will now start proving some properties of all_valid and some_valid. If the list is not empty, then conjunction implies disjunction.

```
Proposition all_implies_some (l : list Prop) :
  l ≠ nil → all_valid l → some_valid l.
```

The proof follows by using the first element of l to prove the disjunction.

Sometimes we wish to split this list of propositions into two parts. In this case, we can prove the following two results, where ↔ is notation for 'if and only if'.

```
Proposition all_cat (l1 l2 : list Prop) :
  all_valid (l1 ++ l2) ↔ all_valid l1 ∧ all_valid l2.
```

```
Proposition some_cat (l1 l2 : list Prop) :
  some_valid (l1 ++ l2) ↔ some_valid l1 ∨ some_valid l2.
```

Both are not hard to prove in Coq; they follow easily from list induction and the definition of list concatenation.

We will often construct a list of propositions from another list of some type A, and a *predicate* on this type. A predicate is a function of type A → Prop. For a predicate P and a list l of type A, the term map P l is a list of propositions. We would like to know when all_valid (map P l) is provable, which means that the predicate P is true for all elements of the list l. After some manipulation we could end up in the case where P is provable for every element of type A. This means in particular that P is valid for all elements of l, i.e. that all_valid (map P l) is provable. This is what the following proposition tells us.

```
Proposition forall_all_map {A : Type} (l : list A) (P : A → Prop) :
  (∀ a, P a) → all_valid (map P l).
```

We can interpret functions of type A → Prop both as sets and as predicates. These interpretations are compatible, but depending on the context one might be preferrable over the other: "convex predicates" sounds somewhat strange. We show this compatibility of interpretations with an example. We will define P := (fun x : R ⇒ 0 ≤ x) : R → Prop as the set/predicate, and let l : list R be a list of real numbers. Then we could interpret all_valid (map P l) as

$$\forall r \in \texttt{l} : 0 \leq r$$

or as

$$\forall r \in \texttt{l} : r \in \{x \in \mathbb{R} : 0 \leq x\}.$$

We now have enough results to define what it means for a list to *contain* an element.

```
Definition contains {A : Type} (a : A) (l : list A) :=
  some_valid (map (fun b ⇒ b = a) l).
```

From this definition, a list l contains a if any of the elements of the list l are equal to a. This should mean every list contains all of its elements, which is indeed provable.

```
Proposition all_valid_contains {A : Type} (l : list A) :
  all_valid (map (fun a ⇒ contains a l) l).
```

Using contains, we can prove all_valid (map P l) by a more intuitive way of reasoning; that for every element in the list l, the predicate P is provable. We can prove this, and an analogous result for some_valid.

```
Theorem contains_P_then_all {A : Type} (l : list A) (P : A → Prop) :
  (∀ a, contains a l → P a) ↔ all_valid (map P l).
Theorem contains_sol_then_some_iff {A : Type} (l : list A) (P : A → Prop) :
  (∃ a, contains a l ∧ P a) ↔ some_valid (map P l).
```

The formal proof of all three theorems relies on list induction.

3.2.5. Convexity

We need results on convex sets, since our verification strategy uses convexity. Specifically, convexity is used when verifying nonpositivity of the solution in a cube $\mathcal{C}$ which intersects the Minkowski difference $\mathcal{K}^\circ - \mathcal{K}^\circ$. In this case, we need to show that for all $x \in \mathcal{C} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ)$, we have $f(x) \leq 0$. For every $x \in \mathcal{C}$, we can find vertices in $\mathcal{C}_N$, such that x is contained in the subcube generated by these vertices. Remember that $\mathcal{C}_N$ are the gridpoints inside the cube. If x is outside $\mathcal{K}^\circ - \mathcal{K}^\circ$, one of the vertices of the subcube must also be outside $\mathcal{K}^\circ - \mathcal{K}^\circ$. This is because both cubes and $\mathcal{K}^\circ - \mathcal{K}^\circ$ are convex sets.

We will treat the specifics of the strategy later, and first lay down the necessary groundwork on convexity for cubes and other shapes. We develop two (compatible) things: convex sets, and convex combinations of a given list of vectors.

3.2.5.1. Convex sets

The notion of sets is usually translated to a function of type `U → Prop` in Coq. Here `U` stands for the type of the elements of the set, but one can also interpret `U` as the set of all elements of this type. A function `P : U → Prop` maps an element of `U` to a propositon, which might be provable or not. We interpret an element `u` as being in the set `P` if `P u` is a provable proposition. In this way `p : P u` is a proof that `u` is in the set `P`.

Convexity is then a property that sets, so functions of types `U → Prop`, can possess. We will assume furthermore that the space `U` is a module space over the real numbers, since then we can talk about convex combinations of elements of `U`. This motivates the following definition of convexity:

```
Definition convex (P : U → Prop) := ∀ (x y : U) (t: R),
      P x → P y → 0 ≤ t ≤ 1 → P (plus (scal (1 - t) x) (scal t y)).
```

This indicates that if we have two elements `x` and `y` in `P`, then any convex combination of them is also in `P`. Here `scal` and `plus` are scalar multiplication and addition in the space `U`.

We will prove some easy properties of convex sets. The first one is that the intersection of two convex sets is again convex.

```
Proposition conj_convex (P Q : U → Prop) :
  convex P → convex Q → convex (fun u ⇒ P u ∧ Q u).
```

Folowing the interpretation of sets mentioned before, `u` is in (`fun u ⇒ P u ∧ Q u`) if `P u ∧ Q u` is provable. This means that `P u` is provable and `Q u` is provable, or that `u` is in `P` and in `Q`. This means that this function describes the intersection of two sets.

To prove this proposition, we follow the definition of convexity. Suppose `x` and `y` are both in `P ∩ Q`. A convex combination of them is in `P ∩ Q` if it is both in `P` and `Q`. We can prove it is in both sets by using the convexity of `P` and `Q`, and that `x` and `y` are in both sets.

The next property is that the entire space, when interpreted as a set, is convex. The entire space corresponds to the function sending every element to a provable proposition, like `True`.

```
Proposition convex_true : convex (fun _ : U ⇒ True).
```

This is easy to prove; once we have expanded the definition of convex, we just need to exhibit a proof of `True` — which does not even depend on the convex combination itself. A proof of true is `I` by definition.

The next proposition is that if two sets have the same members, showing one is convex proves the other is convex. This sounds trivial (it follows from the axiom of extensionality in ZFC) but is not provable for every property of sets in Coq. This is because propositions are not provably equal if they are provably equivalent, unless you add an axiom.

```
Proposition convex_equiv (P1 P2 : U → Prop) :
  (∀ u, P1 u ↔ P2 u) → convex P1 → convex P2.
```

For the proof we need to show that a convex combination of `x` and `y` is in `P2` if `x` and `y` are in `P2`. Because of the equivalence, it is enough to show that the convex combination is in `P1`. Since `P1` is convex, it is enough to show `x` and `y` are in `P1` — which they are, since `P1` and `P2` are equivalent and `x` and `y` are in `P2`.

Another useful property is that convexity is preserved under affine transformations. An affine transformation can be seen as a linear transformation followed by a translation, i.e. something of the form

$$\begin{aligned} A &: U \to V \\ u &\mapsto L(u) + v, \end{aligned}$$

where $L : U \to V$ is a linear transformation and v a vector in V. A preserves convexity, since for u_1 and u_2 in U we have

$$\begin{aligned} A(\lambda u_1 + (1-\lambda)u_2) &= L(\lambda u_1 + (1-\lambda)u_2) + v \\ &= \lambda L(u_1) + \lambda v \ + \ (1-\lambda)L(u_2) + (1-\lambda)v \\ &= \lambda A(u_1) + (1-\lambda)A(u_2). \end{aligned}$$

A formal definition and some properties of affine functions can be found in Appendix A.3. We will inspect their relation to convex sets. Suppose we have a convex set $P \subseteq U$ and an affine transformation $A : U \to V$. Then $A(P)$ is convex, since for $x = A(u_1)$ and $y = A(u_2)$, we have

$$tx + (1-t)y = tA(u_1) + (1-t)A(u_2) = A(tu_1 + (1-t)u_2).$$

Since x and y are in P and P is convex, $tu_1 + (1-t)u_2$ is in P and $A(tu_1 + (1-t)u_2)$ is in $A(P)$.

The translation of this theorem to Coq has the form:

```
Proposition affine_comp_convex' (P : U → Prop) (l : U → V) :
  is_affine l→ convex P → convex (fun v ⇒ ∃ u, P u ∧ l u = v).
```

Here V is another module space over $\mathbb{R}$. The statement $A(P)$ is intuitive from a set-theoretical view, but translates to a more difficult statement in Coq. We would usually write

$$A(P) = \{Ap : p \in P\}.$$

This is not sufficient for Coq; it does not provide the Prop which should be provable for a $v \in V$ to be in $A(P)$. We obtain a proper translation of $A(P)$ to Coq, when we write it in the form

$$A(P) = \{v \in V : \exists p \in P, A(p) = v\}.$$

A statement that is easier to write (and prove!) in Coq is

```
Proposition affine_comp_convex (P : V → Prop) (l: U → V) :
  is_affine l → convex P → convex (fun u ⇒ P (l u)).
```

This means that if we have an affine transformation $A : U \to V$ and a convex set $P \subseteq V$, the set $\{x \in U : A(x) \in P\}$ is convex as well. This last set is usually denoted like $A^{-1}(P)$. In this case, we have a statement $A^{-1}(P)$ that is less intuitive from a set-theoretical point of view, but more intuitive from Coq's point of view. This is no exception, and one can wonder which point of view is better at capturing what is actually going on. When looking at the definition of continuity on topological spaces, Coq's point of view may be preferable.

affine_comp_convex has various useful consequences. One of these is that if $P \subseteq U$ and $Q \subseteq V$ are convex, then $P \times Q = \{(p, q) : p \in P \wedge q \in Q\}$ is convex.

```
Proposition product_convex (P : U → Prop) (Q : V → Prop) :
  convex P → convex Q → convex (fun w : U × V ⇒ P w.1 ∧ Q w.2).
```

This follows directly from conj_convex and the fact that projections to the coordinates are affine transformations.

Another useful consequence is that half-spaces defined by linear inequalities are convex as well. We will assume U is a normed module over $\mathbb{R}$ now, since then we can borrow the linearity definition of Coquelicot.

```
Theorem lin_ineq_convex (l : U → R) (r : R) :
  is_linear l → convex (fun v ⇒ l v ≥ r).
```

After one application of affine_comp_convex, we are left to show that l is affine and that sets of the form $\{v : v \geq r\} \subseteq \mathbb{R}$ are convex. The first part follows easily, since all linear transformations are affine transformations. The second part is easy to derive from our definition of convexity.
Similar theorems, where $\geq$ is replaced by $\leq$ or where $l(v)$ and r have switched places are easy to derive from lin_ineq_convex and convex_equiv.

We have set out to build this infrastructure since we want to be able to prove that cubes in $\mathbb{R}^3$ are convex. Coquelicot interprets $\mathbb{R}^3$ as the product of $\mathbb{R}^2$ with $\mathbb{R}$, and endows it with a product norm and

a product 'ball'. Normed modules have a concept of balls, since normed spaces are uniform spaces with additional structure. Coquelicot defines balls for products of uniform spaces U and V as follows:

$$B_{U\times V}((u_0, v_0), \epsilon) = \{(u, v) : u \in B_U(u_0, \epsilon) \wedge v \in B_V(v_0, \epsilon)\}.$$

Balls on $\mathbb{R}$ are defined as $B_{\mathbb{R}}(x, \epsilon) = (x - \epsilon, x + \epsilon)$, like one would expect. This means that balls in $\mathbb{R}^3$ correspond to open cubes. It would be helpful if we were be able to prove that all balls (or their closure) are convex. Although this might not hold in general uniform spaces, balls in normed modules are somehow compatible with the ball induced by the norm. Recall normed module axioms 3 and 4:

- Axiom 3: For all $x, y \in U$ and ϵ, we have that $\|y -_U x\| < \epsilon$ implies $y \in B(x, \epsilon)$;
- Axiom 4: There exists some 'norm factor' $N_U \in \mathbb{R}$ satisfying the following property. For all $x, y \in U$ and $\epsilon > 0$, if $y \in B(x, \epsilon)$ then $\|y -_U x\| < N_U \cdot \epsilon$.

This compatibility might give an angle to prove balls in normed modules are sufficiently well-behaved to provide convexity of all balls. One can show that the ball induced by the norm is convex.

Theorem 10. *Let U be a normed module over $\mathbb{R}$. Let x be in U, and let $\epsilon > 0$. Then the set*

$$\{y \in U : \|y - x\| < \epsilon\}$$

is convex.

The translation to Coq becomes

```
Theorem ball_norm_convex (x : U) (eps : R) : convex (fun y ⇒ norm (minus y x) < eps).
```

Proof. Let y and z be in the set, and $0 \leq t \leq 1$. We need to show that $(1-t)y + tz$ is in the set. We distinguish the cases where $t = 0$, $0 < t < 1$ and $t = 1$. The first and the last case are easy, since the convex combination reduces to y and z respectively. When $0 < t < 1$ we use the following inequalities:

$$\begin{aligned} \|(1-t)y + tz - x\| &= \|\,((1-t)y - (1-t)x) + (tz - tx)\,\| \\ &\leq (1-t)\|y - x\| + t\|z - x\| \\ &< (1-t)\epsilon + t\epsilon = \epsilon \end{aligned}$$

Note that this last inequality is only strict if $0 < 1 - t$ and $0 < t$. □

It turns out that even though the normball is always convex and the ball and the normball are compatible, we can construct a counterexample; a normed module over $\mathbb{R}$ where not all balls are convex even though they are compatible with the normball. One such counterexample is if we give $\mathbb{R}$ the absolute value as norm, and the following definition for balls of radius ϵ around x:

$$B'(x, \epsilon) = \{y : |y - x| < \epsilon\} \cup \begin{cases} \emptyset & \text{if } \epsilon < 1; \\ \{y : |y - x| = 2\} & \text{if } \epsilon = 1; \\ \{y : |y - x| \leq 2\epsilon\} & \text{if } \epsilon > 1. \end{cases}$$

The first term makes sure that axiom 3 of normed modules holds. The additional elements at $\epsilon = 1$ cause balls $B'(x, 1)$ to be non-convex: all elements y with $1 \leq |y - x| < 2$ are missing. To satisfy the triangle inequality for balls (axiom 3 of uniform spaces), we include all those missing elements for balls with radius larger than 1. In the worst case, the distance $\|y - x\|$ is just over two times the ϵ for which $y \in B(x, \epsilon)$. This means we can find a norm factor (for example: 3) so that axiom 4 holds, which concludes the counterexample.

In conclusion, this approach does not yield a proof that cubes are convex. We need additional information on convex sets with more structure: convex combinations of a given set of vectors.

3.2.5.2. Convex combinations

To develop a notion of convex combinations, we first need a notion of linear combinations. Convex combinations are linear combinations with extra conditions on the coefficients of the vectors. The extra conditions are that all coefficients need to be positive, and that their sum is equal to 1. The requirement that all coefficients are positive can be captured with the `all_valid` construction discussed in Section 3.2.4. For a notion of sums we need some additional definitions.

The formal definition of linear combinations and sums in Coq will rely heavily on the concept of lists. Lists appear often in computer science, but less frequent in mathematics. Clever notation allows intuitive reasoning on sums and linear combinations. Unfortunately, Coq does not come with a sense of intuition, so we have to do some work to prove relatively easy things.

In this section, we will first mention the important theorems we needed on convex combinations, along with a proof. The results will not be given in their exact formal representation in Coq.

We will then cover one formal proof of a result on linear combinations. A formal mathematical proof will take more effort than expected or wanted, even though the proof in Coq is not very hard. We will therefore skip formal proofs of other results on linear combinations. The interested reader can find these results in Appendix A.

In the following part, U will always be a module space on $\mathbb{R}$.

Definition 3. *Let $V = \{v_1, \dots, v_n\}$ be a finite set of vectors in U. We call an element u of U a* convex combination *of V, if there exists nonnegative coefficients $\lambda_1, \dots, \lambda_n$ such that the following two conditions hold:*

- $\sum_{i=1}^{n} \lambda_i v_i = u$
- $\sum_{i=1}^{n} \lambda_i = 1$

We will now prove it makes sense to call such a combination *convex*.

Proposition 11. *Let $V = \{v_1, \dots, v_t\}$ be a finite set of vectors in U. Then*

$$\{u \in U : u \text{ is a convex combination of } V\}$$

is a convex set.

Proof. Let x and y be convex combinations of V. This means there exists $\mu_1, \dots, \mu_n$ and $\nu_1, \dots, \nu_n$ such that we have $\sum_{i=1}^{n} \mu_i = \sum_{i=1}^{n} \nu_i = 1$ and

$$\sum_{i=1}^{n} \mu_i v_i = x \qquad\qquad \sum_{i=1}^{n} \nu_i v_i = y$$

Suppose $0 \leq t \leq 1$. We wish to show that $(1-t)x + ty$ is a convex combination of v_i. Define $\lambda_i = (1-t)\mu_i + t\nu_i$ for $i = 1, \dots, n$. Firstly, we have

$$\sum_{i=1}^{n} \lambda_i = (1-t)\left(\sum_{i=1}^{n} \mu_i\right) + t\left(\sum_{i=1}^{n} \nu_i\right) = 1 - t + t = 1.$$

Furthermore, we have

$$\sum_{i=1}^{n} \lambda_i v_i = (1-t)\left(\sum_{i=1}^{n} \mu_i v_i\right) + t\left(\sum_{i=1}^{n} \nu_i v_i\right) = (1-t)x + ty.$$

So $(1-t)x + ty$ is a convex combination of v_i. □

Next, we investigate how we can prove that a convex combination lies in some convex set.

Theorem 12. *Let $V = \{v_1, \dots, v_n\}$ be a finite set of vectors in U, and $P \subseteq U$ a convex set. If each $v_i \in V$ is contained in P, then every convex combination of V is contained in P.*

Proof. We prove this with induction on the size of the set V. Suppose V is empty. Then there are no convex combinations, so the statement holds trivially.

Now suppose the statement holds for all sets V with n elements. Let $V' = \{v'_1, \dots, v'_{n+1}\}$ be a set with $n+1$ elements. Let u be a convex combination of V'. This means we can find λ_i so that $\sum_{i=1}^{n+1} \lambda_i = 1$ and $\sum_{i=1}^{n+1} \lambda_i v'_i = u$. We separate the first element of V', by writing

$$u = \sum_{i=1}^{n+1} \lambda_i v'_i = \lambda_1 v'_1 + \sum_{i=2}^{n+1} \lambda_i v'_i.$$

If $\lambda_1 = 0$, then u is a convex combination of the set $V' \setminus \{v'_1\}$, which has n elements. By the induction hypothesis we obtain $u \in P$. If $\lambda_1 = 1$, then $u = v_1 \in P$ and we are done. So suppose $0 < \lambda_1 < 1$. Then we are allowed to rewrite u to

$$u = \lambda_1 v'_1 + (1 - \lambda_1) \sum_{i=2}^{n+1} \frac{\lambda_i}{1 - \lambda_1} v'_i.$$

This implies u is in P if both v'_1 and the $\sum$ expression is in P, since P is convex. By assumption, v'_1 is in P. Now note that

$$\sum_{i=2}^{n+1} \frac{\lambda_i}{1 - \lambda_1} = \frac{1}{1 - \lambda_1} \cdot \left(\left(\sum_{i=1}^{n+1} \lambda_i \right) - \lambda_1 \right) = \frac{1 - \lambda_1}{1 - \lambda_1} = 1.$$

This means that $\sum_{i=2}^{n+1} \frac{\lambda_i}{1-\lambda_1} v'_i$ is a convex combination of $V \setminus \{v'_1\}$, and is thus in P by the induction hypothesis. This concludes the proof. □

By contraposition we obtain the following lemma:

Lemma 13. *Let $V = \{v_1, \ldots, v_n\}$ be a finite set of vectors in U, and $P \subseteq U$ a convex set. Suppose u is a convex combination of V and $u \notin P$. Suppose furthermore that we can decide whether $w \in P$ or $w \notin P$ for every $w \in U$. Then there exists a $v_j \in V$ for which $v_j \notin P$.*

Proof. It is enough to prove $\neg \forall v_i \in V : v_i \in P$. This is because for finite quantification, $\neg \forall i : A_i$ implies $\exists i : \neg A_i$ if we have $A_i \vee \neg A_i$ for all i.

So suppose for contradiction that $\forall v_i \in V : v_i \in P$. Then by Theorem 12, all convex combinations of V are also in P. However, u is a convex combination of V with $u \notin P$, so we have reached a contradiction. □

Lemma 13 is important and will be used directly in the verification procedure of the polynomials. The only thing that remains to be shown is that a cube in $\mathbb{R}^3$ is indeed a convex combination of its vertices. We will define a cube to be a product of intervals, since this will be convenient for interval arithmetic. This representation is less convenient to show that cubes are convex combinations of their vertices. Although geometrically obvious, we need a formal proof. We show some results which are needed for such a formal proof.

Theorem 14. *Let $V = \{v_1, \ldots, v_n\}$ be a finite set of vectors in U, and $l : \mathbb{R} \to U$ an affine function. Suppose $l(r_1)$ and $l(r_2)$ are convex combinations of V and $r_1 \leq r_2$. Then for any $u \in \mathbb{R}$ with $r_1 \leq u \leq r_2$, $l(u)$ is a convex combination of V.*

Proof. It suffices to show that $l(u)$ is a convex combination of $l(r_1)$ and $l(r_2)$. Then the result follows by convexity of convex combinations of V.

If $r_1 = r_2$ then $l(u) = l(r_1)$ and we are done. So suppose $r_1 < r_2$. Define $\lambda = \frac{u - r_1}{r_2 - r_1}$. Then $0 \leq \lambda \leq 1$, and

$$(1 - \lambda) l(r_1) + \lambda l(r_2) = l((1 - \lambda) r_1 + \lambda r_2) = l \left(\frac{r_2 - u}{r_2 - r_1} r_1 + \frac{u - r_1}{r_2 - r_1} r_2 \right) = l \left(\frac{r_2 - r_1}{r_2 - r_1} u \right) = l(u),$$

which concludes the proof. □

Lemma 15. *Let $V = \{v_1, \ldots, v_n\}$ be a finite set of vectors in $U \times \mathbb{R}$. Suppose (u, r_1) and (u, r_2) are convex combinations of V and $r_1 \leq r_2$. Then every (u, w) with $r_1 \leq w \leq r_2$ is a convex combination of V.*

Proof. This follows directly from Theorem 14, since $r \mapsto (u, r)$ is an affine transformation. □

Lemma 15 will be convenient, since we will define cubes as products of intervals. Showing an element is in the cube then reduces to proving that two faces of the cube are convex combinations of the vertices. For cubes in $\mathbb{R}^3$, vectors on a face of the cube are convex combinations of just 4 of the 8 vertices. To capture this, we have the following proposition.

Proposition 16. *Let $V = \{v_1, \dots, v_n\}$ and $W = \{w_1, \dots, w_m\}$ be finite sets of vectors in U. Suppose $W \subseteq V$. If some $u \in U$ is a convex combination of W, then it is a convex combination of V.*

We omit the proof, since the result is trivial. Note that in Coq this needs to be shown explicitly.
A face of a cube is a subset of lower dimension. We can leverage this with the following theorem.

Theorem 17. *Let U_1 and U_2 be module spaces over $\mathbb{R}$. Suppose $V = \{v_1, \dots, v_n\}$ is a finite set of vectors in U_1, w a convex combination of V and $l : U_1 \to U_2$ an affine transformation. Then $l(w)$ is a convex combination of $l(V)$.*

Proof. Let λ_i be such that $\sum_{i=1}^n \lambda_i = 1$ and $\sum_{i=1}^n \lambda_i v_i = w$. Using induction it easily follows that for affine functions l, we have

$$l\left(\sum_{i=1}^n \lambda_i v_i\right) = \sum_{i=1}^n \lambda_i l(v_i).$$

This directly shows that $l(w)$ is a convex combination of $l(V)$. □

Lemma 18. *Suppose $V = \{v_1, \dots, v_n\}$ is a finite set of vectors in $U \times \mathbb{R}$. Suppose there is some $c \in \mathbb{R}$ so that for all $v_j = (a, b) \in V$, we have $b = c$, i.e. all vectors in V have the same second coordinate. Let V' be the set of all first coordinates of V. Suppose $u \in U$ is a convex combination of V'. Then $(u, c) \in U \times \mathbb{R}$ is a convex combination of V.*

Proof. Define $l : U \to U \times \mathbb{R}$ by $w \mapsto (w, c)$. Then l is an affine function. By assumption, we then have $l(V') = V$. The result then follows directly from Theorem 17. □

All results mentioned above have been formalized in Coq. These all rely heavily on the concept of lists and their formal statements can therefore feel somewhat disconnected to the theorems mentioned above. To illustrate this, we will show how linear combinations are defined and prove a property on them.

Linear combinations make sense on a module space, where we have scalar multiplication and addition. In the next part, K is a ring and U is a module space over K.

```
Fixpoint linear_combination (coeffs: list K) (vectors: list U): U :=
  match vectors with
  | nil ⇒ zero
  | cons v vectors ⇒
    match coeffs with
    | nil ⇒ zero
    | cons c coeffs ⇒
      plus (scal c v) (linear_combination coeffs vectors)
    end
  end.
```

Note that we do not require `coeffs` and `vectors` to have the same length. This allows more flexibility in using `linear_combination`. If `coeffs` and `vectors` have different length, the recursion stops as soon as one of them is used up. The result of `linear_combination` is therefore equal when applied to the same arguments extended with `zero : K` or `zero : U` respectively.

Let (c_n) be a list of elements of K and (v_n) be a list of elements of U, not necessary of equal size. Let $S(l)$ denote the length/size of a list l. Then the mathematical notation of `linear_combination` (c_n) (v_n) would be:

$$\sum_{i=1}^{\min(S((c_n)),S((v_n))} c_i v_i$$

By construction, we also have the following two identities.

```
Proposition empty_vectors (coeffs : list K) :
  linear_combination coeffs nil = zero.
```

```
Proposition empty_coefficients (vectors : list U) :
  linear_combination nil vectors = zero.
```

We can prove this in Coq by performing case analysis on the arguments.

We occasionally need to perform scalar multiplication on a linear combination, i.e. $k \cdot \sum_{i \in I} c_i v_i$. This is of course equal to $\sum_{i \in I} (kc_i) v_i$ — equal to the linear combination obtained by premultiplying all c_i with k. To prove this, case analysis will not be enough. Instead, we need to perform induction on the arguments, which are lists. Although it is easy to convince oneself of their truth, carrying out the complete formal argument can sometimes be confusing. Care needs to be taken so that the generated induction hypotheses are strong enough to prove the goal.

Theorem 19 (combination_scal)**.** *Let K be a ring, V a module space over K, k an element of K. Suppose that (c_n) is a list of elements of K and that (v_n) is a list of elements of V. Let $S(l)$ denote the length/size of a list l. Then*

$$k \cdot \sum_{i=1}^{\min(S((c_i)),S((v_i))} c_i v_i = \sum_{i=1}^{\min(S((c_i)),S((v_i))} (kc_i) v_i.$$

```
Theorem combination_scal (coeffs : list K) (k : K) (vectors : list U) :
  scal k (linear_combination coeffs vectors)
    = linear_combination (map (fun r ⇒ mult k r) coeffs) vectors.
```

Proof. Let $k \in K$ and (v_n) be a list of elements of V. For brevity we will use the notation

$$\sum_{i=1}^{\text{minsize}} c_i v_i := \sum_{i=1}^{\min(S((c_n)),S((v_n))} c_i v_i$$

We will prove the following expression with induction on the list (v_n):

$$\forall (c_n) : k \cdot \sum_{i=1}^{\text{minsize}} c_i v_i = \sum_{i=1}^{\text{minsize}} (kc_i) v_i.$$

The base case is when (v_n) is an empty list. In this case the linear combinations both reduce to 0_V for all lists (c_n). The remaining expression is $k \cdot 0_V = 0_V$, which is a result that Coquelicot has proven.

Let $\text{cons}(v, (v_n))$ denote the list obtained by adding v to the head of the (v_n). For the induction step we then need to show that the following implication holds.

$$\begin{aligned} \forall (v_n) \forall (v \in V) : &\left(\forall (c_n) : k \cdot \sum_{i=1}^{\text{minsize}} c_i v_i = \sum_{i=1}^{\text{minsize}} (kc_i) v_i \right) \implies \\ &\left(\forall (c_n) : k \cdot \sum_{i=1}^{\text{minsize}} c_i (\text{cons}(v, (v_i)))_i = \sum_{i=1}^{\text{minsize}} (kc_i)(\text{cons}(v, (v_i))) \right) \end{aligned}$$

So, let (v_n) be a list in V, v be an element of V and (c_n) be a list in K. If (c_n) is an empty list, the expression reduces once again to $k \cdot 0_V = 0_V$, which is easy to prove. So suppose $(c_n) = \text{cons}(c_0, (c'_n))$. The sum on the left hand side can now be rewritten to yield

$$\begin{aligned} k \cdot \sum_{i=1}^{\text{minsize}} c_i (\text{cons}(v, (v_i)))_i &= k \sum_{i=1}^{\text{minsize}} (\text{cons}(c, (c'_n)))_i (\text{cons}(v, (v_i))) \\ &= k \left(cv + \sum_{i=1}^{\text{minsize}} c'_i v_i \right) \\ &= (kc)v + k \cdot \sum_{i=1}^{\text{minsize}} c'_i v_i \\ &= (kc)v + \sum_{i=1}^{\text{minsize}} (kc'_i) v_i \\ &= \sum_{i=1}^{\text{minsize}} (\text{cons}(kc, (kc'_n)))_i (\text{cons}(v, (v_n)))_i, \end{aligned}$$

which is what we wanted to prove. We used the induction hypothesis to go from the third line to the fourth line. The quantification on (c_n) was necessary to be able to apply it for the (c'_n) list. □

As we can see, formal proofs and theorems involving lists quickly become painful to write in the standard mathematical format, even though writing them in Coq is not a big hassle. Intermediate results that were necessary for the results on convex combinations can be found in Appendix A. We will not mention them here. We will show the Coq translation of the definition and theorems on convex combinations we mentioned.

```
Definition convex_combination_of (u : U) (vectors : list U) :=
  ∃ (coeffs : list R),
    all_valid (map (fun r ⇒ 0 ≤ r) coeffs) ∧
    sum (take (size vectors) coeffs) = 1 ∧
    linear_combination coeffs vectors = u.
```

The all_valid part ensures that all coefficients are nonnegative. We need to take (size vectors) of coeffs, since coeffs might be a longer list than vectors and these extra terms do not contribute to the linear_combination. This means we do not actually need to require that all elements of coeffs are nonnegative, since the last coefficients do not make a contribution. We require it anyway since it is more succinct and equally valid.

```
Theorem convex_combinations_convex (vectors : list U) :
  convex (fun u ⇒ convex_combination_of u vectors).
```

convex_combinations_convex is the translation of Proposition 11.

```
Theorem vertices_in_convex_combinations_in
    (P: U → Prop) (vectors : list U) (u : U) :
  convex P → all_valid (map P vectors) → convex_combination_of u vectors → P u.
```

vertices_in_convex_combinations_in is the translation of Theorem 12. Note the use of all_valid (map P vectors), which means that every element of the list vectors is in the set defined by P.

```
Theorem outside_convex_implies_vertex_outside
    (u : U) (vectors : list U) (P: U → Prop) :
  convex P → (∀ v : U, decidable (P v)) →
  convex_combination_of u vectors → ¬ P u → some_valid (map (fun v ⇒ ¬ P v) vectors).
```

outside_convex_implies_vertex_outside is the translation of Lemma 13. We need to explicitly require P to be decidable for the argument to work without the law of excluded middle. The some_valid expression encodes $\exists v \in$ vectors: $v \notin$ P.

```
Theorem convex_collapse_affine (vectors : list U) (u r1 r2 : R ) (l : R → U) :
  is_affine l → r1 ≤ u ≤ r2 →
  convex_combination_of (l r1) vectors → convex_combination_of (l r2) vectors →
    convex_combination_of (l u) vectors.
```

convex_collapse_affine is Theorem 14.

```
Theorem convex_collapse_last (u : U × R) (vectors : list (U × R)) (r1 r2 : R ) :
  r1 ≤ u.2 ≤ r2 → convex_combination_of (u.1, r1) vectors →
  convex_combination_of (u.1, r2) vectors →
    convex_combination_of u vectors.
```

convex_collapse_last is Lemma 15, a direct consequence of convex_collapse_affine for the affine function fun r ⇒ (u1, r).

```
Theorem convex_vector_subset (v1 v2 : list U) (u : U) :
  all_valid (map (fun v ⇒ contains v v2) v1) → convex_combination_of u v1 →
    convex_combination_of u v2.
```

convex_vector_subset is Proposition 16. Note that the notion of subsets is translated to lists with the contains property defined in Section 3.2.4.

```
Theorem combination_map (u : U) (vectors : list U) (l : U → V) :
  is_affine l → convex_combination_of u vectors →
    convex_combination_of (l u) (map l vectors).
```

combination_map is Theorem 17.

```
Theorem convex_last_irrelevant (u : U × R) (vectors : list (U × R)) :
  all_valid (map (fun v ⇒ v.2 = u.2) vectors) →
  convex_combination_of u.1 (map (fun v ⇒ v.1) vectors) →
    convex_combination_of u vectors.
```

`convex_last_irrelevant` is Lemma 18, a direct consequence of `combination_map` for the affine function `fun v ⇒ (v, u2)`.

3.3. Specialized verification theorems

Let us recall the specialized results we need to perform the verification:

- We need to prove that ∇f is the derivative of f. The extra problem here is that f can be represented in multiple polynomial bases: either in the regular x,y and z corresponding to the axis in $\mathbb{R}^3$, or in s_1, s_2 and s_3: the basis of B_3-invariant polynomials.
- We need to be able to verify that f is nonpositive in a cube which *does not* intersect $\mathcal{K}^\circ - \mathcal{K}^\circ$.
- We need to be able to verify that f is nonpositive in a cube which *does* intersect $\mathcal{K}^\circ - \mathcal{K}^\circ$.
- We need to verify that the union of all cubes cover the required region.

3.3.1. Proving ∇f is the derivative of f

For our verification, all our functions f will be B_3-invariant. They are specified with coefficients in the base of the basic invariants s_1, s_2 and s_3, where

$$s_1 = x^2 + y^2 + z^2, \qquad s_2 = x^4 + y^4 + z^4, \quad \text{and} \quad s_3 = x^6 + y^6 + z^6.$$

When $f \in \mathbb{R}[s_1, s_2, s_3]$ and we want to calculate ∇f, we could preserve some of this form. Writing f in the form

$$f = p \circ ((x, y, z) \mapsto (s_1(x, y, z), s_2(x, y, z), s_3(x, y, z)))$$

suggests we might be able to use the chain rule when calculating derivatives. Indeed,

$$\nabla f = \nabla p(s_1, s_2, s_3) \cdot \begin{pmatrix} \frac{\partial s_1}{\partial x} & \frac{\partial s_1}{\partial y} & \frac{\partial s_1}{\partial z} \\ \frac{\partial s_2}{\partial x} & \frac{\partial s_2}{\partial y} & \frac{\partial s_2}{\partial z} \\ \frac{\partial s_3}{\partial x} & \frac{\partial s_3}{\partial y} & \frac{\partial s_3}{\partial z} \end{pmatrix}.$$

Although this approach works fine, it is not sufficient for our purposes. It turns out that bounds obtained on ∇f by performing interval arithmetic on this expression are not very good. If we simplify the resulting expression for ∇f by collecting powers of x, y and z, the obtained bounds get sharper.

This can be explained as follows. Note that s_1, s_2 and s_3 are not independent functions; they all depend on x, y and z. For example, we know that $s_1 < s_2 < s_3$ for large values of x, y and z. In its unsimplified form, interval arithmetic calculations will give a result as if s_1, s_2 and s_3 are independent. The information of the dependency of the values of s_1, s_2 and s_3 is not taken into account, and as such the obtained bounds are worse. This information is used when we collect powers of x, y and z, and this makes the bound of the simplified expression sharper.

In the original verification program, the simplified expression was used to generate bounds. This means the cubes expect this sharp bound to verify that $f(x) \leq 0$. Therefore, we need Coq to understand the two different representations of the polynomial f. In Coq, we will define the function f in terms of s_1, s_2, and s_3. By proceeding in this way, we can prove the function is indeed B_3-invariant, and that performing the verification on the $0 \leq x \leq y \leq z$ region is indeed enough.

To make Coq understand the direct representation, we will define the coefficients of the direct representation as symbolic expressions on the original coefficients. We will explain what we mean by this with an example. Firstly, check that

$$\begin{aligned} s_1^2 = \left(x^2 + y^2 + z^2\right)^2 &= \left(x^2 + y^2\right)^2 + 2\left(x^2 + y^2\right) z^2 + z^4 \\ &= x^4 + 2x^2y^2 + y^4 + 2x^2z^2 + 2y^2z^2 + z^4. \end{aligned}$$

Now suppose we have $f = c_1 s_1^2 + c_2 s_2$. Then

$$\begin{aligned} f(x,y) &= c_1\left(x^4+y^4+z^4+2x^2y^2+2x^2z^2+2y^2z^2\right)+c_2\left(x^4+y^4+z^4\right) \\ &= (c_1+c_2)\left(x^4+y^4+z^4\right)+2c_1\left(x^2y^2+x^2z^2+y^2z^2\right). \end{aligned}$$

We could now define $c_1' = c_1 + c_2$ and $c_2' = 2c_1$; new coefficients of the direct representation defined in terms of the original coefficients. This is nice since the two equations are still provably equal. Furthermore, we can use the bounds on the original coefficients to prove bounds on the new coefficients. After the representations have been proven equal and bounds have been proven for the new coefficients, we can make the definition of the new coefficients abstract. This prevents Coq from ever looking at the definition of the new coefficients again. We prevent Coq from doing this because all necessary information has been captured, and recalculation of bounds is computationally expensive.

Manually proving that the two expressions are equal is not something we would want to do by hand. Thankfully Coq comes with the built-in `ring` tactic. This tactic tries to prove any equality on a ring, such as $\mathbb{R}$, and will succeed in doing this for our polynomial. The only remaining problem is generating these symbolic expressions for the new coefficients in terms of the old ones.

We created a program using the Sage Mathematics Software System [14] to accomplish this. Some programs of the original verification procedure also used Sage to generate C++ files needed for performing the verification. Sage comes with various methods for manipulating polynomials on arbitrary rings, so is well-suited to the task. Note that Sage is an external program, and anything written in Sage cannot be trusted for the verification. We get around this by letting our Sage program generate a Coq source file with the required results. If this Coq source file can be verified, we can be confident the results are true without depending on the Sage program.

Now that we have verified the two representations are equal, we need to verify that ∇f is the derivative of f — now using the direct representation. We will use Theorem 8 and Theorem 9 here. Remember that derivatives on $\mathbb{R}^3 \to \mathbb{R}$ are defined as Fréchet derivatives.

We will use an example to demonstrate the approach. Suppose g is a function of $\mathbb{R}^2$ to $\mathbb{R}$. We wish to prove that ∇g is the derivative of g at (x_0, y_0), i.e. that

$$(h_1, h_2) \mapsto \frac{\partial g}{\partial x}(x_0, y_0)h_1 + \frac{\partial g}{\partial y}(x_0, y_0)h_2$$

is the Fréchet derivative of g at (x_0, y_0). By Theorem 8 (where $U = \mathbb{R}$) it suffices to show the following three conditions:

- $h_1 \mapsto \frac{\partial g}{\partial x}(x_0, y_0)h_1$ is the partial Fréchet derivative in the first coordinate of g at (x_0, y_0);
- $\frac{\partial g}{\partial y}$ is the partial derivative of g in the second coordinate in a region around (x_0, y_0);
- $\frac{\partial g}{\partial y}$ is continuous at (x_0, y_0).

The first condition is the same as saying that $\frac{\partial g}{\partial x}$ is the partial derivative of g in the first coordinate. This can be show using automation provided by Coquelicot. The same approach works for verifying the second condition: we can prove it for every $(x, y) \in \mathbb{R}^2$ which can be seen as a (large) region around (x_0, y_0).

For the third condition, we will use Theorem 9. This means we need to show:

- $\frac{\partial g}{\partial y}$ is continuous in the first coordinate at x_0;
- $\frac{\partial^2 g}{\partial y^2}$ exists in a region around (x_0, y_0);
- There exists some $M \in \mathbb{R}$ so that $\left|\frac{\partial^2 g}{\partial y^2}\right| < M$ in a region around (x_0, y_0).

We can prove the first condition by showing that $\frac{\partial^2 g}{\partial y \partial x}$ is the derivative at x_0, which implies continuity. The second condition can again be shown with automation from Coquelicot. We can prove the third

condition by using interval arithmetic to provide a bound. This does require an additional assumption on (x_0, y_0): namely, that we have some intervals which contain x_0 and y_0. This means we cannot prove ∇g is the derivative of g globally. This is fine, since we only need this fact in a bounded region.

The difference between proving this for a function $\mathbb{R}^2 \to \mathbb{R}$ and a function $\mathbb{R}^3 \to \mathbb{R}$ lies in the way the first conditions of both lists are proven. The first condition for $\mathbb{R}^3 \to \mathbb{R}$ is to show another function is the partial Fréchet derivative of a function of $\mathbb{R}^2 \to \mathbb{R}$. To prove this we need another application of Theorem 8. The same holds for the first condition for proving continuity; to prove a function of $\mathbb{R}^3 \to \mathbb{R}$ is continuous, we need that another function of $\mathbb{R}^2 \to \mathbb{R}$ is continuous, which requires an extra application of Theorem 9.

The approach we have taken here is applicable for general functions f. Another approach would have been to exploit that f is a polynomial. It is possible to inductively define a polynomial type in Coq. One could then prove that all polynomials are infinitely differentiable, that partial derivatives are polynomials again, and thus that the derivative of a multivariate polynomial is the gradient.

3.3.2. Showing f is nonpositive in a cube outside the Minkowski difference

There are two cases to consider when we want to verify f is nonpositive in a given cube. Either this cube intersects the Minkowski difference, or it does not. When the cube is completely outside the Minkowski difference, we can disregard this information altogether. We will verify that f is nonpositive in this cube, and the Minkowski difference $\mathcal{K}^\circ - \mathcal{K}^\circ$ will play no further role in this section.

Let us recap the approach described in Section 1.2. We start with a cube $\mathcal{C} = [x_l, x_u] \times [y_l, y_u] \times [z_l, z_u]$ and some $x \in \mathcal{C}$. We wish to prove $f(x) \leq 0$ by evaluating f at a number of gridpoints $\mathcal{C}_N$ and using a bound $\|\nabla f(\mathcal{C})\| \leq D$. For every $x_n \in \mathcal{C}_N$, we have

$$\begin{aligned} f(x) &= f(x) - f(x_n) + f(x_n) \\ &\leq f(x_n) + |f(x) - f(x_n)| \\ &\leq f(x_n) + D\|x - x_n\|. \end{aligned}$$

The third line follows from the second line follows by Theorem 2. If any gridpoint x_n has the property that $f(x_n) + D\|x - x_n\| \leq 0$, we get $f(x) \leq 0$. More formally:

$$\forall x \in \mathcal{C} : \left(\exists x_n \in \mathcal{C}_N : f(x_n) + D\|x - x_n\| \leq 0\right) \implies f(x) \leq 0. \tag{3.1}$$

From the $\|x - x_n\|$ term, it is clear that x_n which are close to x should provide a better estimate of $f(x)$. In chapter 1 we mentioned that the closest gridpoint x_n has the property

$$\|x - x_n\| \leq \frac{\sqrt{3}\delta}{2N}.$$

Here δ is the sidelength of the cube $\mathcal{C}$, i.e. $\delta = x_u - x_l$. This is true since we can find a smaller cube in $\mathcal{C}$ whose vertices are in $\mathcal{C}_N$. This subcube has sidelength $\frac{\delta}{N}$. The closest vertex of this subcube must then have distance less than $\frac{\sqrt{3}\delta}{2N}$, by the Pythagorean theorem. More formally, the statement is

$$\forall x \in \mathcal{C} : \exists x_n \in \mathcal{C}_N : \|x - x_n\| \leq \frac{\sqrt{3}\delta}{2N}. \tag{3.2}$$

We can now combine this with our previous inequality, where we take x_n to be the gridpoint with minimum distance:

$$\begin{aligned} f(x) &\leq f(x_n) + D\|x - x_n\| \\ &\leq f(x_n) + D\frac{\sqrt{3}\delta}{2N}. \end{aligned}$$

This last statement no longer depends on x but only on x_n. If

$$\forall x_n \in \mathcal{C}_N : f(x_n) + D\frac{\sqrt{3}\delta}{2N} \leq 0$$

is true, then it proves that $\forall x \in \mathcal{C} : f(x) \leq 0$. Since $\mathcal{C}_N$ is a finite set, this can be shown by checking whether the statement holds for each $x_n \in \mathcal{C}_N$ individually.

More formally, we have the implication

$$\left(\forall x_n \in \mathcal{C}_N : f(x_n) + D\frac{\sqrt{3}\delta}{2N} \leq 0\right) \implies \forall x \in \mathcal{C} : f(x) \leq 0. \tag{3.3}$$

We will now prove the statements 3.1, 3.2, and 3.3 in Coq. We will first show how the cubes and gridpoints are defined. We will then prove statement 3.2. We will proceed with a proof of a slightly more general version of 3.1. Once these results have been shown, we can use them to prove 3.3.

3.3.2.1. Defining cubes and gridpoints

```
Definition cube_closed (v : R × R × R) (side : R) (x : R × R × R) :=
  v.1.1 ≤ x.1.1 ≤ v.1.1 + side ∧
  v.1.2 ≤ x.1.2 ≤ v.1.2 + side ∧
  v.2 ≤ x.2 ≤ v.2 + side.
```

The statement `cube_closed v side x` means that `x` is contained in the cube with sidelength `side` and lower-left corner `v`. Note that `side` is not required to be nonnegative. However, it must be nonnegative if the cube has at least one element, which is what the following proposition states.

```
Proposition sideNonNeg (xl x : R × R × R) (side : R) :
  cube_closed xl side x → 0 ≤ side.
```

Suppose we divide each side of `cube_closed` v δ in N intervals. We can then define the $\mathcal{C}_N$ as

$$\mathcal{C}_N = \left\{\left(v_x + \frac{\delta i}{N}, v_y + \frac{\delta j}{N}, v_z + \frac{\delta z}{N}\right) : i, j, z \in 0, \dots, N\right\}.$$

To define this in Coq, we will first build $\{(i, j, z) : i, j, z \in 0, \dots, N\}$. Our initial building block is `range n`. This builds a list of length `n` containing the natural numbers $0, \dots, n-1$. Next, we need `list_product`. The function `list_product` takes two lists as arguments and builds a new list containing all element pairings of the two list. This means it is the list analog of the Cartesian product.

```
Definition range_triple (n : nat) := let therange := range n in
  list_product (list_product therange therange) therange.
```

The `list_products` are ordered in this way since we want elements of `range_triple n` to be of type `nat` × `nat` × `nat`, which is notation for (`nat` × `nat`) × `nat`.

We now need to map each (i, j, k) to the corresponding value in $\mathbb{R}^3$. Each natural number is mapped in a similar way to a coordinate; based on the lower coordinate, the side length, and the total number of intervals.

```
Definition interval_offset (lower side : R) (pieces : positive) (index : nat) :=
  lower + side / (IZR (Zpos pieces)) × (IZR (Z.of_nat index)).
```

We require `pieces` to be `positive` so that the division will always make sense. The `positive` type encodes the numbers $\mathbb{N} \setminus \{0\}$, but these numbers are defined in a binary way. This is different from the unary way `nat` is defined.

We cannot divide real numbers directly by positive numbers, we must first map them back to $\mathbb{R}$. `Zpos p` maps `positive` numbers to $\mathbb{Z}$, `IZR` is an Injection from $\mathbb{Z}$ to $\mathbb{R}$, and `Z.of_nat` maps $\mathbb{N}$ to $\mathbb{Z}$. There is also `INR` and `IPR`, but using `IZR` is more convenient for displaying numbers in Coq.

We can now use `interval_offset` to define a function which maps triples of natural numbers to triples of real numbers.

```
Definition cube_offset (vl : R × R × R) (side : R) (n : positive)
                       (nv : nat × nat × nat) :=
  (interval_offset vl.1.1 side n nv.1.1,
   interval_offset vl.1.2 side n nv.1.2,
   interval_offset vl.2 side n nv.2).
```

We now use the `map` function to apply `cube_offset` to all elements of `range_triple`. Note that we want the natural numbers range from 0 to N inclusive, so we need `range_triple` $N+1$. We want N to be positive, but `range_triple` needs a natural number as argument. To accomplish this we first map N back to the natural numbers $\mathbb{N}$ with `Pos.to_nat`. Putting it all together, we get the following definition.

```
Definition cube_grid (vl : R × R × R) (side : R) (N : positive) :=
  map (cube_offset vl side N) (range_triple (S (Pos.to_nat N))).
```

Also note the partial application of `cube_offset` here; the `nv` argument was not provided, so `cube_offset vl side n` has type `nat` × `nat` × `nat` → $R \times R \times R$.

3.3.2.2. Existence of a close gridpoint

To prove statement 3.2, we need to first investigate the properties of `range` and `range_triple`.

We want `range (n : nat)` to be the list of length n containing all i with $0 \le i < n$. The following definition will turn out to have this property. Note that the obtained list is in reverse order, since this is more natural with this recursive definition.

```
Fixpoint range (n : nat) :=
  match n with
  | 0 ⇒ @nil nat
  | S n ⇒ n :: (range n)
  end.
Theorem range_contains (n m : nat) : (m < n)%nat ↔ contains m (range n).
```

The proof of `range_contains` requires induction on `n`. It proves that the current definition of `range` suffices for our purposes. We use `contains` once again to encode $m \in$ `range n`.

This helps, once we know how `list_product` behaves with respect to contained elements. As a Cartesian product, it should satisfy that $(a, b) \in A \times B$ iff $a \in A$ and $b \in B$. This is indeed the case, by the following result.

```
Theorem list_product_contains {A B : Type} (a : A) (b : B)
                              (la : list A) (lb : list B):
  (contains a la ∧ contains b lb) ↔ contains (a,b) (list_product la lb).
```

The formal proof relies on list induction and is not trivial, but we will refrain from discussing the details. Proofs involving lists are best understood when walking through the proof script manually, while a translation of this formal proof to English would obscure more than it would illuminate.

By applying this twice we obtain the following result for `range_triple`.

```
Proposition in_range_triple' (n : nat) (a11 a12 a2 : nat) :
  contains (a11, a12, a2) (range_triple n) ↔
  (a11 < n)%nat ∧ (a12 < n)%nat ∧ (a2 < n)%nat.
```

This provides us with information on the triples of natural numbers contained in this list. This will be useful after we prove a one dimensional version of statement 3.2. The one dimensional version would be, for $0 < N \in \mathbb{N}$,

$$\forall x \in [a, a+\delta] : \exists n \in \mathbb{N} : n \le N \wedge \left| x - \left(a + \frac{\delta n}{N} \right) \right| < \frac{\delta}{2N}.$$

This translates to Coq as follows.

```
Lemma closest_fraction2 (lower x side : R) (N : positive) : lower ≤ x ≤ lower + side
  → ∃ n, Rabs (x - lower - side / (IZR (Zpos N)) * IZR (Z.of_nat n))
                      ≤ side / (2 * IZR (Zpos N))
                ∧ (n ≤ Pos.to_nat N)%nat.
```

Note again the use of `IZR`, `Zpos`, and `Pos.to_nat` to map the numbers to the appropriate types. We prove `closest_fraction2` by first proving

$$\forall x \in [a, a+\delta] : \exists n \in \mathbb{N} : n < N \wedge \left(a + \frac{\delta n}{N} \le x \le a + \frac{\delta(n+1)}{N} \right). \tag{3.4}$$

If we can find such an $n \in \mathbb{N}$, either n or $n+1$ is closer than $\frac{\delta}{2N}$ to x. This proves `closest_fraction2`.

We can prove statement 3.4 for the interval $[0, 0+\delta]$, and translate the result to $[a, a+\delta]$ later. We will prove the untranslated version of 3.4 here, which is translated to Coq as follows.

```
Lemma close_fraction_closed (x upper : R) (N : positive) : 0 ≤ x ≤ upper →
  ∃ n : nat, upper / (IZR (Zpos N)) * (IZR (Z.of_nat n)) ≤ x ≤
             upper / (IZR (Zpos N)) * (IZR (Z.of_nat (S n)))
          ∧ (n < Pos.to_nat N)%nat.
```

We need another intermediate result to prove this; we need that

$$\forall x \in [0, \delta), \exists n \in \mathbb{N} : n < N \wedge \left(\frac{\delta n}{N} \leq x < \frac{\delta(n+1)}{N} \right). \tag{3.5}$$

Note the condition and the conclusion changes from closed intervals to half-closed intervals. If we have this result, we can prove `close_fraction_closed` by distinguishing cases. Suppose $x \in [0, \delta]$. If $x \neq \delta$, then $x \in [0, \delta)$ and we can use 3.5. The conclusion is sharper than we need (x is in the half-closed interval, which is strictly smaller than the closed interval), but suffices. If $x = \delta$, then $n = N - 1$ satisfies the inequality. We can distinguish between x and δ by the axioms of $\mathbb{R}$ in Coq.

```
Lemma close_fraction (x upper : R) (N : positive) : 0 ≤ x < upper →
  ∃ n : nat, upper / (IZR (Zpos N)) * (IZR (Z.of_nat n)) ≤ x <
             upper / (IZR (Zpos N)) * (IZR (Z.of_nat (S n)))
          ∧ (n < Pos.to_nat N)%nat.
```

Proof. We prove `close_fraction` directly. This proof will use the archimedean property of the real numbers. Specifically, the real numbers in Coq come with a function `up : R → Z`, which sends a real number r to the smallest integer strictly larger than r. This makes it almost but not exactly equal to the round-up/ceiling operation $\lceil \cdot \rceil$:

$$\mathsf{up}(r) = \begin{cases} r + 1 & \text{if } r = \lceil r \rceil; \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

The axioms of $\mathbb{R}$ then specify that $\mathsf{up}(r) > r$ and that $\mathsf{up}(r) - r \leq 1$. We claim that

$$\mathsf{up}\left(\frac{xN}{\delta}\right) - 1$$

satisfies the conditions. Since $0 \leq x$, we have $0 \leq \frac{xN}{\delta} < \mathsf{up}(\frac{xN}{\delta})$. Now `up` maps to `Z`, but by the previous inequality we get $0 \leq \mathsf{up}(\frac{xN}{\delta}) - 1$. This makes $\mathsf{up}(\frac{xN}{\delta}) - 1$ a natural number, as required.

We have to prove that

$$\frac{\delta}{N} \cdot \left(\mathsf{up}\left(\frac{xN}{\delta}\right) - 1 \right) \leq x < \frac{\delta}{N} \cdot \mathsf{up}\left(\frac{xN}{\delta}\right).$$

To do so, first multiply both inequalities with $\frac{N}{\delta}$. We now have to prove

$$\mathsf{up}\left(\frac{xN}{\delta}\right) - 1 \leq \frac{xN}{\delta} < \mathsf{up}\left(\frac{xN}{\delta}\right)$$

Both equalities now follow directly from the axioms on `up`.

The only remaining thing to prove is $\mathsf{up}(\frac{xN}{\delta}) - 1 < N$, or equivalently $\mathsf{up}(\frac{xN}{\delta}) - N < 1$. Since $x < \delta$, $\frac{Nx}{\delta} < N$, so $-N < -\frac{Nx}{\delta}$. Then

$$\underbrace{\mathsf{up}\left(\frac{xN}{\delta}\right) - N < \mathsf{up}\left(\frac{xN}{\delta}\right) - \frac{Nx}{\delta}}_{\leq 1}$$

where we used the archimedean property of $\mathsf{up}(r) - r \leq 1$. ☐

We are now ready to prove statement 3.2:

```
Lemma closest_gridpoint (vl : R × R × R) (side : R) (N : positive) (v : R × R × R):
  cube_closed vl side v →
    some_valid (map (fun vn ⇒
                        norm (minus vn v) ≤ sqrt 3 × side / (2 × IZR (Zpos N)))
                   (cube_grid vl side N)).
```

Proof. Recall that `cube_closed` is defined as the product of three intervals. This means that for $w, v \in \mathbb{R}^3$ and $\delta \in \mathbb{R}$, `cube_closed` $w\ \delta\ v$ means

$$v_x \in [w_x, w_x + \delta] \quad \wedge \quad v_y \in [w_y, w_y + \delta] \quad \wedge \quad v_z \in [w_z, w_z + \delta].$$

Each of these statements are exactly the input that `closest_fraction2` requires. We obtain a triple of natural numbers (n_x, n_y, n_z), which are all less than or equal to `N`. Additionally we get

$$\left|v_x - \left(w_x + \frac{\delta n_x}{N}\right)\right| \le \frac{\delta}{2N} \ \wedge\ \left|v_y - \left(w_y + \frac{\delta n_y}{N}\right)\right| \le \frac{\delta}{2N} \ \wedge\ \left|v_z - \left(w_z + \frac{\delta n_z}{N}\right)\right| \le \frac{\delta}{2N}.$$

By definition, `cube_grid` $w\ \delta$ `N` is equal to `map` (`cube_offset` $w\ \delta$ `N`) (`range_triple` (`N` + 1)). It is now enough to show that (n_x, n_y, n_z) is in `range_triple` (`N` + 1), and that

$$\left\|\left(w_x + \frac{\delta n_x}{N}, w_y + \frac{\delta n_y}{N}, w_z + \frac{\delta n_z}{N}\right) - v\right\| \le \frac{\sqrt{3}\delta}{2N}.$$

This is enough, since if (n_x, n_y, n_z) is contained in `range_triple` (`N` + 1), then the associated vector `cube_offset` $w\ \delta$ `N` (n_x, n_y, n_y) is contained in `cube_grid` $w\ \delta$ `N`. If the inequality is satisfied, this means that there exists an element in `cube_grid` $w\ \delta$ `N` satisfying the distance predicate. This is equivalent to `some_valid` (`map _ cube_grid` $w\ \delta$ `N`).

Since n_x, n_y and n_z are all less than or equal to N, they are all strictly less than $N + 1$. Then `in_range_triple'` gives us immediately that (n_x, n_y, n_z) is contained in `range_triple` (`N` + 1).

The inequality follows from

$$\begin{aligned}
&\left\|\left(w_x + \frac{\delta n_x}{N}, w_y + \frac{\delta n_y}{N}, w_z + \frac{\delta n_z}{N}\right) - v\right\| \\
&\quad = \sqrt{\left|v_x - \left(w_x + \frac{\delta n_x}{N}\right)\right|^2 + \left|v_y - \left(w_y + \frac{\delta n_y}{N}\right)\right|^2 + \left|v_y - \left(w_y + \frac{\delta n_y}{N}\right)\right|^2} \\
&\quad \le \sqrt{3 \cdot \left(\frac{\delta}{2N}\right)^2} \\
&\quad = \frac{\sqrt{3}\delta}{2N},
\end{aligned}$$

which concludes the proof. □

3.3.2.3. Nonpositivity from a nonpositive neighbour

We will now prove a more general version of statement 3.1.

Theorem 20. *Let P be a convex subset of $\mathbb{R}^n$ and $x \in P$. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function which is differentiable on P. If there exists $D \in \mathbb{R}$ with $\|\nabla f(y)\| \le D$ for all $y \in P$, and there is $z \in P$ with*

$$D\|x - z\| + f(z) \le 0$$

then $f(x) \le 0$.

The translation to Coq requires us to replace $\mathbb{R}^n$ with a normed module U. Furthermore, we replace the derivative with a Fréchet derivative and the bound on the gradient with a bound on the operator norm of the Fréchet derivative.

```
Lemma close_grad_bound_convex (f : U → R) (fd : U → U → R) (x : U)
                               (D : R) (P : U → Prop) :
  P x → convex P →
  (∀ y, P y → filterdiff f (locally y) (fd y) ∧ ∀ v, norm (fd y v) ≤ D × norm v) →
  (∃ z, P z ∧ D × norm (minus z x) + f z ≤ 0) →
    f x ≤ 0.
```

Proof. Let P, x, f and D be as in the assumptions. Then

$$\begin{aligned} f(x) &\leq f(x) - f(z) + f(z) \\ &\leq |f(x) - f(z))| + f(z) \\ &\leq D\|x - z\| + f(z) \\ &\leq 0. \end{aligned}$$

We need to motivate why $|f(x) - f(z)| \leq D\|x - z\|$. We can do this by using Theorem 2, but only if for all $0 \leq t \leq 1$ we have $\|\nabla f((1-t)x + tz)\| \leq D$. This is true since we have $x, z \in P$, P is convex and the bound holds in all of P. □

3.3.2.4. Nonpositivity from nonpositive gridpoints

We now want to combine our results so far to prove statement 3.3. Translated to Coq, this statement becomes:

```
Lemma cube_nonpos_gridcheck (f : R × R × R → R) (fd: R × R × R → R × R × R → R)
                             (xl x : R × R × R) (side : R) (D : R) (N : positive) :
  (∀ y, cube_closed xl side y →
          filterdiff f (locally y) (fd y) ∧ (∀ v, norm (fd y v) ≤ D × norm v)) →
  cube_closed xl side x →
  all_valid (map (fun y ⇒ D × (sqrt 3) × side / (2 × IZR (Zpos N)) + f y ≤ 0)
                 (cube_grid xl side N)) →
    f x ≤ 0.
```

Note that `f`, `fd`, `D` and the first assumption are exactly as required for `close_grad_bound_convex`. We will use ∇f instead of the Fréchet derivative in the proof.

Proof. Let $f : \mathbb{R}^3 \to \mathbb{R}$ be differentiable on a cube with lower left corner w and side δ. Suppose $\|\nabla f(y)\| \leq D \in \mathbb{R}$ for all $f \in$ `cube_closed` $w\ \delta$. Let $N \in \mathbb{N}$ be positive and suppose furthermore that for $x_n \in$ `cube_grid` $w\ \delta\ N$, we have $D\frac{\sqrt{3}}{2N} + f(x_n) \leq 0$. We wish to show that $f(x) \leq 0$ for every x contained in `cube_closed` $w\ \delta$. We now apply `close_grad_bound_convex`. To do so, we need to show that `cube_closed` $w\ \delta$ is convex, and that there exists some $z \in$ `cube_closed` $w\ \delta$ with $D\|x - z\| + f(z) \leq 0$. The convexity of `cube_closed` can easily be shown,since it is an intersection of half-spaces defined by linear inequalities. We have proved in Section 3.2.5.1 that half-planes defined by linear inequalities are convex, and that convexity is preserved under taking intersections.

For the other part, we will show that there is an element of `cube_grid` $w\ \delta\ N$ satisfies the requirements; i.e. that `some_valid` (`map` _ (`cube_grid` $w\ \delta\ N$)). This means that there should exist some $x_n \in$ `cube_grid` $w\ \delta\ N$ with $x_n \in$ `cube_closed` $w\ \delta$ and $D\|x - x_n\| + f(x_n) \leq 0$. We should have that $\forall x_n \in$ `cube_grid` $w\ \delta\ N : x_n \in$ `cube_closed` $w\ \delta$, but we do need to prove this. This can be derived from `range_contains` and is not too hard.

What is left to show is that $\exists x_n \in$ `cube_grid` $w\ \delta\ N : D\|x - x_n\| + f(x_n) \leq 0$. We will instead prove that

$$\exists x_n \in \texttt{cube_grid}\ w\ \delta\ N : D\frac{\sqrt{3}\delta}{2N} + f(x_n) \leq 0 \wedge \|x - x_n\| \leq \frac{\sqrt{3}\delta}{2N}$$

since this directly implies the required inequality. Indeed, suppose x_n satisfies the conditions, then

$$D\|x - x_n\| + f(x_n) \leq D\frac{\sqrt{3}\delta}{2N} + f(x_n) \leq 0.$$

Now by assumption, all $x_n \in$ `cube_grid` $w\ \delta\ N$ satisfy the first inequality. This means we just need to show

$$\exists x_n \in \texttt{cube_grid}\ w\ \delta\ N : \|x - x_n\| \leq \frac{\sqrt{3}\delta}{2N}$$

which is exactly `closest_gridpoint`, so we are done. □

3.3.3. Showing f is nonpositive in a cube intersecting the Minkowski difference

It takes a bit more effort to prove f is nonpositive when the cube intersects the Minkowski difference. In the previous case it was enough to find a gridpoint that was close enough to an x in the cube. In this case we need to find a smaller cube whose vertices are gridpoints, and which contains x. Cubes are specified by a lower left corner and a side length. This makes some gridpoints not elligible to be the lower left corner of a subcube. For example, the subcube whose lower-left corner is the gridpoint in the upper-right corner of the original cube, is wholly outside the original cube. This requires some care with inequalities.

Let us write Q for the Minkowski difference $\mathcal{K}^\circ - \mathcal{K}^\circ$. Once a subcube containing x has been found, we need one of our results on convexity to show that if $x \notin Q$, then one of the vertices of the cube is not in Q. This means we also have to prove that elements in a cube are convex combinations of the vertices.

We then need to prove that if x is in a cube with sidelength δ, it has distance at most $\sqrt{3}\delta$ to all vertices. We can then combine this with Theorem 20 to prove the final statement

$$\left(\forall x_n \in \mathcal{C}_n : x_n \in Q \vee D\frac{\sqrt{3}\delta}{N} + f(x_n) \leq 0\right) \implies (\forall x \in \mathcal{C} \cap Q : f(x) \leq 0)\,. \tag{3.6}$$

We start by proving elements of cubes can be written as convex combinations of the vertices of the cubes.

```
Definition cube_vertices (xl : R × R × R) (side : R) :=
  [:: xl; (xl.1.1, xl.1.2, xl.2 + side);
   (xl.1.1, xl.1.2 + side, xl.2); (xl.1.1, xl.1.2 + side, xl.2 + side);
   (xl.1.1 + side, xl.1.2, xl.2); (xl.1.1 + side, xl.1.2, xl.2 + side);
   (xl.1.1 + side, xl.1.2 + side, xl.2); (xl.1.1 + side, xl.1.2 + side, xl.2 + side)].
```

We will now prove that x is in a cube with lower-left corner w and δ if and only if it is a convex combination of `cube_vertices` w δ. Note that this only works if δ is nonnegative.

```
Proposition cube_closed_is_polyhedron (xl x : R × R × R) (side : R) :
  cube_closed xl side x ↔
  (0 ≤ side) ∧ convex_combination_of x (cube_vertices xl side).
```

We can prove the $\implies$ implication using the infrastructure we built for convex combinations. We first project an x in the cube to a convex combination of elements of two opposing faces. Each face element is a convex combinations of all the vertices, if they are convex combinations of a subset of the original vertices: the corners of their face. It is then enough to project this face to a square in $\mathbb{R}^2$, and prove the projection of the face element to $\mathbb{R}^2$ is a convex combination of the projection of the corners of the face to $\mathbb{R}^2$. We can repeat this procedure to go from $\mathbb{R}^2$ to $\mathbb{R}$, which means we just have to show that intervals are convex combinations of their edges.

For the $\impliedby$ implication it is enough to show that each vertex is in the cube. That each convex combination of the vertices is in the cube then follows from convexity of the cube.

We will now prove the required bound on the distance of a cube element to every cube vertex.

```
Proposition vertex_distance (xl x : R × R × R) (side : R) : cube_closed xl side x →
  all_valid (map (fun v ⇒ norm (minus x v) ≤ sqrt 3 × side) (cube_vertices xl side)).
```

This is an easy consequence of the fact that $|a-b| \leq \delta$ and $|a-(b+\delta)| \leq \delta$ if $b \leq a \leq b+\delta$. This can be used for every coordinate of an $x \in$ `cube_closed` w δ. Then we have for vertices v,

$$\|x - v\| \leq \sqrt{3 \cdot \delta^2} = \sqrt{3}\delta.$$

The next result is on the existence of a subcube containing an x in the original cube.

```
Proposition in_contained_sub_cube (xl x : R × R × R) (side : R) (n : positive) :
  cube_closed xl side x →
  ∃ xn : R × R × R, all_valid (map (fun el ⇒ contains el (cube_grid xl side n))
                                (cube_vertices xn (side / IZR (Zpos n))))
                    ∧ cube_closed xn (side / IZR (Zpos n)) x.
```

To prove this we rely again on (a translated version of) close_fraction_closed which states that for $a, b, \delta \in \mathbb{R}$ and $N \in \mathbb{N}$ a positive number, we have

$$b \in [a, a+\delta] \implies \exists n \in \mathbb{N} : n < N \wedge b \in \left[a + \frac{\delta n}{N}, a + \frac{\delta(n+1)}{N}\right].$$

It is important that this n is strictly smaller than N, since it guarantees that $a + \frac{\delta(n+1)}{N} \leq a + \delta$. This means that the smaller interval is really contained in the bigger interval.

Proof. Suppose $w, v \in \mathbb{R}^3$, $\delta \in \mathbb{R}$ and $N \in \mathbb{N}$ a positive number. If $v \in$ cube_closed $w\ \delta$, then we can use close_fraction_closed to obtain n_x, n_y, $n_z \in \mathbb{N}$, with

$$v_i \in \left[w_i + \frac{\delta n_i}{N}, w_i + \frac{\delta(n_i+1)}{N}\right] \quad \text{for } i \in \{x, y, z\}.$$

The left hand sides of each interval specify a vector in $\mathbb{R}^3$ which satisfies our requirement. By the statement above, the subcube has this vector as lower-left, has sidelength $\frac{\delta}{N}$ and contains v. This vector also coincides with cube_offset $w\ \delta\ N\ (n_x, n_y, n_z)$. Such a value is contained in the list of gridpoints cube_grid $w\ \delta\ N$ if each of n_x, n_y, and n_z is less than $N+1$. Now n_x, n_y and n_z are less than N by construction. This gives just enough slack for the other vertices of the subcube to still be contained in cube_grid $w\ \delta\ N$. □

We can now prove statement 3.6. Its translation to Coq is as follows.

```
Lemma cube_nonpos_gridcheck_edge (f : R × R × R → R)
                    (fd: R × R × R → R × R × R → R) (xl x : R × R × R)
                    (side : R) (D : R) (n : positive) (Q : R × R × R → Prop) :
  convex Q → (∀ v : R × R × R, decidable (Q v)) →
  (∀ y, cube_closed xl side y →
       filterdiff f (locally y) (fd y) ∧ (∀ v, norm (fd y v) ≤ D × norm v)) →
  cube_closed xl side x → (¬ Q x) →
  all_valid (map (fun y ⇒ Q y ∨ D × (sqrt 3) × side / IZR (Zpos n) + f y ≤ 0)
                 (cube_grid xl side n)) →
    f x ≤ 0.
```

We need the decidability assumption to be able to prove that one of the vertices of a cube containing x is outside Q when x is outside Q.

Proof. Let $f : \mathbb{R}^3 \to \mathbb{R}$ be differentiable on a cube with lower left corner w and side δ. Suppose $\|\nabla f(y)\| \leq D \in \mathbb{R}$ for all $y \in$ cube_closed $w\ \delta$. Let $N \in \mathbb{N}$ be positive, Q be a convex set and suppose that for every $x_n \in$ cube_grid $w\ \delta\ N$ we have $x_n \in Q$ or $D\frac{\sqrt{3}\delta}{N} + f(x_n) \leq 0$. We wish to show that for $x \in$ cube_closed $w\ \delta$ with $x \in Q$, we have $f(x) \leq 0$. We use close_grad_bound_convex once again, which means that it is enough to show that

$$\exists z \in \texttt{cube_closed}\ w\ \delta : D\|x - z\| + f(z) \leq 0.$$

We now use in_contained_sub_cube to find a z_1 for which $x \in$ cube_closed $z_1\ \frac{\delta}{N}$ and for which the vertices of that cube are all contained in cube_grid $w\ \delta\ N$. Since $x \notin Q$, there is a vertex z_2 of the cube with $z_2 \notin Q$ by outside_convex_implies_vertex_outside. We claim that this z_2 satisfies the requirements.

By the construction of the subcube, $z_2 \in$ cube_grid $w\ \delta\ N$ and so $z_2 \in$ cube_closed $w\ \delta$. By assumption, we must have

$$(z_2 \in Q) \vee \left(D\frac{\sqrt{3}\delta}{N} + f(z_2) \leq 0\right).$$

We have $z_2 \notin Q$, which means that we must have $D\frac{\sqrt{3}\delta}{N} + f(x_n) \leq 0$. By vertex_distance we have $\|x - z_2\| \leq \frac{\sqrt{3}\delta}{N}$, so that

$$D\|x - z_2\| + f(z_2) \leq D\frac{\sqrt{3}\delta}{N} + f(z_2) \leq 0,$$

which concludes the proof. □

3.3.4. Cubes cover the required region

Now that we can prove f is nonnegative on cubes, we need to show that the cubes we check are enough. The C++ program of the original verification provides us with a list of cubes to check, along with a gridsize for each cube for which the verification should be successful. We do not trust this list, so we will have to check whether the cubes in the list do indeed cover the region which requires verification. The region for which we wish to prove nonpositivity of f is $\{v \in \mathbb{R}^3 : s(v) < 0\} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ)$. The function s was defined as $s(x) = \|x\|^2 - 1$ for all Platonic and Archimedean solids. The verification has been performed for the truncated tetrahedron, so we will assume furthermore that the region we wish to cover is $\{x : \|x\| < 1\} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ)$.

We verify that the provided cubelist covers the region in several steps.

- We show the $0 \leq x \leq y \leq z$ region (the 'wedge') is indeed enough for B_3 invariant functions;
- We define $\mathcal{K}^\circ - \mathcal{K}^\circ$ as a system of linear inequalities, and try to simplify them in the wedge;
- We use the results mentioned above and the list of cubes to conclude.

3.3.4.1. Verification in the wedge is enough

We want to show that verification in the wedge, the region for which $0 \leq x \leq y \leq z$, is enough. To do this, it is convenient to characterize B_3-invariant functions and sets. Remember that B^3 was defined as the symmetry group of the regular cube $[-1, 1]^3$. This means that $A \in \mathsf{B}_3$ if $A \in O(3)$ and $A([-1, 1]^3) = [-1, 1]^3$. A set P is B_3-invariant if for all $A \in \mathsf{B}_3$, $AP = P$. A function f with domain $\mathbb{R}^3$ is B^3-invariant if for $A \in \mathsf{B}_3$, and all $x \in \mathbb{R}^3$ we have $f(Ax) = f(x)$. We will pick a convenient set of generators of B_3 to define invariance. Requiring invariance on just these generators is enough to guarantee B_3-invariance and simplifies proving something is B_3-invariant.

We will choose 5 reflections as generators. Specifically, these will be

$$
\begin{aligned}
\sigma_1(x, y, z) &= (y, x, z),\\
\sigma_2(x, y, z) &= (x, z, y),\\
\rho_1(x, y, z) &= (-x, y, z),\\
\rho_2(x, y, z) &= (x, -y, z) \text{ and}\\
\rho_3(x, y, z) &= (x, y, -z).
\end{aligned}
$$

Geometrically, one can identify σ_1 and σ_2 with reflection on a plane through two opposite edges of the cube. Likewise, each of ρ_1, ρ_2 and ρ_3 corresponds to reflection on a plane that lies exactly halfway between two opposite faces. Algebraically, σ_1 and σ_2 permute the input coordinates while ρ_1, ρ_2 and ρ_3 negate exactly one coordinate.

These reflections generate all of B_3, but we will not prove this. We will prove that the basic invariants s_1, s_2, and s_3 are B_3-invariant when defined in this way. This characterization is also enough to show that verification in the wedge is sufficient.

We will first cover B_3-invariant sets. We will define a set P to be B_3-invariant if for every $A \in \mathsf{B}_3$, we have $\{x \in \mathbb{R}^3 : Ax \in P\} = A^{-1}P \subseteq P$. Since all such A are bijective and B_3 is closed under taking inverses, this implies that $P \subseteq AP$, so that $P = AP$ for each $A \in \mathsf{B}^3$. This means we have precisely captured B_3 invariant sets in this definition. We will only require this for A equal to one of our generators, since then invariance follows for all elements of B_3.

```
Definition permutes1_R3 (P : R × R × R → Prop) := ∀ v, P (v.1.2, v.1.1, v.2) → P v.
Definition permutes2_R3 (P : R × R × R → Prop) := ∀ v, P (v.1.1, v.2, v.1.2) → P v.
Definition coords_permute_R3 (P : R × R × R → Prop) :=
  permutes1_R3 P ∧ permutes2_R3 P.

Definition mirror1_R3 (P : R × R × R → Prop) := ∀ v, P (-v.1.1, v.1.2, v.2) → P v.
Definition mirror2_R3 (P : R × R × R → Prop) := ∀ v, P (v.1.1, -v.1.2, v.2) → P v.
Definition mirror3_R3 (P : R × R × R → Prop) := ∀ v, P (v.1.1, v.1.2, -v.2) → P v.
Definition coords_mirror_R3 (P : R × R × R → Prop) :=
  mirror1_R3 P ∧ mirror2_R3 P ∧ mirror3_R3 P.

Definition B3_invariant (P : R × R × R → Prop) :=
  coords_permute_R3 P ∧ coords_mirror_R3 P.
```

Note the divide between σ-invariance (`coords_permute_R3`) and ρ-invariance (`coords_mirror_R3`). A proof of `coords_permute_R3` will allow us to sort the coordinates, while `coords_mirror_R3` will allow us to only look at nonnegative coordinates. This last statement is captured in the following proposition.

```
Proposition mirror_implies_abs_enough (P : R × R × R → Prop) (v : R × R × R) :
  coords_mirror_R3 P → P (Rabs v.1.1, Rabs v.1.2, Rabs v.2) → P v.
```

The proof is easy: for each coordinate of `v` we check whether it is larger or equal then zero. If it is, the absolute value is equal to the coordinate itself and we are done. If it is not, we apply the relevant `mirror_R3` property to conclude.

We can now start defining B_3-invariant functions. This makes sense for a function with domain $\mathbb{R}^3$ and any codomain I. We can define this invariance in two ways. We can state that $f(Ax) = f(x)$ for every $A \in \mathsf{B}_3$ and $x \in \mathbb{R}^3$. This captures the property, but would be disconnected from the `B3_invariant` we defined earlier. We can also define f to be B_3-invariant if for every $x_0 \in \mathbb{R}^3$, the set $\{x \in \mathbb{R} : f(x_0) = f(x)\}$ is `B3_invariant`. These two definitions will turn out to be equivalent.

```
Definition equality_prop {D : Type} (f : D → I) (v0 : D) := fun v ⇒ f v0 = f v.
Definition B3_invariant_fun' (f : R × R × R → I) :=
  ∀ v, B3_invariant (equality_prop f v).
```

`B3_invariant_fun'` is the latter way to define B_3-invariant functions. We will now develop the first way.

```
Definition fun_permutes1_R3 (f : R × R × R → I) := ∀ v, f v = f (v.1.2, v.1.1, v.2).
Definition fun_permutes2_R3 (f : R × R × R → I) := ∀ v, f v = f (v.1.1, v.2, v.1.2).
Definition fun_permutes_R3 (f : R × R × R → I) :=
  fun_permutes1_R3 f ∧ fun_permutes2_R3 f.
```

We will now prove a partial result on this definition of σ invariance.

```
Proposition fun_permutes_funprop_permutes (f : R × R × R → I) :
  fun_permutes_R3 f → ∀ v, coords_permute_R3 (equality_prop f v).
```

Proof. Let f be a function from $\mathbb{R}^3$ to some codomain I. Suppose that for all $x, y, z \in \mathbb{R}$, we have $f(x, y, z) = f(y, x, z) = f(x, z, y)$. We want to show that for all $x, y, z \in \mathbb{R}$, $\{v \in \mathbb{R}^3 : f(v) = f(x, y, z)\}$ is σ_1- and σ_2-invariant. So suppose $f(v_2, v_1, v_3) = f(x, y, z)$. By assumption,

$$f(v_1, v_2, v_3) = f(v_2, v_1, v_3) = f(x, y, z)$$

so the set is σ_1-invariant. The same approach works for σ_2-invariance. □

We can see from the proof that the two representations come down to nearly the same thing, although this might not be directly clear from the statement.

We continue with defining ρ-invariant functions.

```
Definition fun_mir1_R3 (f : R × R × R → I) := ∀ v, f v = f (-v.1.1, v.1.2, v.2).
Definition fun_mir2_R3 (f : R × R × R → I) := ∀ v, f v = f (v.1.1, -v.1.2, v.2).
Definition fun_mir3_R3 (f : R × R × R → I) := ∀ v, f v = f (v.1.1, v.1.2, -v.2).
Definition fun_mir_R3 (f : R × R × R → I) :=
  fun_mir1_R3 f ∧ fun_mir2_R3 f ∧ fun_mir3_R3 f.
```

```
Proposition fun_mir_funprop_mir (f : R × R × R → I) :
  fun_mir_R3 f → ∀ v, coords_mirror_R3 (equality_prop f v).
```

`fun_mir_funprop_mir` is the same result as `fun_permutes_funprop_permutes` but for ρ-invariance. The proof is very comparable, so we omit it.

We can now state the other way to define B_3 invariant functions, and prove they are equivalent.

```
Definition B3_invariant_fun (f : R × R × R → I) := fun_permutes_R3 f ∧ fun_mir_R3 f.
```

```
Lemma B3_fun_invariance_equiv (f : R × R × R → I) :
  B3_invariant_fun f ↔ B3_invariant_fun' f.
```

This lemma proves the equivalence. The proof of the $\Longrightarrow$ implication relies on the two results we proved earlier: `fun_mir_funprop_mir` and `fun_permutes_funprop_permutes`. The $\Longleftarrow$ implication basically follows from reversing the argument.

We now need a method to sort the coefficients of an element of $\mathbb{R}^3$. The real numbers in Coq come provided with functions returning the minimum and the maximum of two real numbers, `Rmin` and `Rmax`. We use these to build functions for the minimum, maximum and middle values of three real numbers.

```
Definition Rmin3 (a b c : R) := Rmin (Rmin a b) c.
Definition Rmax3 (a b c : R) := Rmax (Rmax a b) c.
Definition Rmid3 (a b c : R) := Rmax3 (Rmin a b) (Rmin a c) (Rmin b c).
```

The next proposition shows that Rmid3 behaves like we want it to.

```
Proposition mid_is_between (a b c : R) : Rmin3 a b c ≤ Rmid3 a b c ≤ Rmax3 a b c.
```

Before we can actually use these functions to sort coordinates, we need to show that these functions are σ-invariant. This is intuitively obvious: the sorted version of a list stays the same when we change the order of the elements in the list.

We have Rmax a b = Rmax b a and likewise for Rmin by a result in the standard library, which makes it easy to prove the following propositions.

```
Proposition mid3_permutes1 (a b c : R) : Rmid3 a b c = Rmid3 b a c.
Proposition mid3_permutes2 (a b c : R) : Rmid3 a b c = Rmid3 a c b.
Proposition min3_permutes1 (a b c : R) : Rmin3 a b c = Rmin3 b a c.
Proposition min3_permutes2 (a b c : R) : Rmin3 a b c = Rmin3 a c b.
Proposition max3_permutes1 (a b c : R) : Rmax3 a b c = Rmax3 b a c.
Proposition max3_permutes2 (a b c : R) : Rmax3 a b c = Rmax3 a c b.
```

If the coordinates are in the proper order, we would like to be able to perform a rewrite. This is what the following propositions enable us to do. These all follow easily from results from Coq, stating that Rmin a b ≤ a ≤ Rmax a b for all a and b in R.

```
Proposition mid3_is_proper (a b c : R) : a ≤ b ≤ c → Rmid3 a b c = b.
Proposition min3_is_proper (a b c : R) : a ≤ b → a ≤ c → Rmin3 a b c = a.
Proposition max3_is_proper (a b c : R) : a ≤ c → b ≤ c → Rmax3 a b c = c.
```

We can now define the coordinate sorting function, prove it is σ-invariant, and provide a rewrite rule when the input is in the correct order. All these results rely on the previously proven properties of Rmin3, Rmax3 and Rmid3.

```
Definition sort_coords (v : R × R × R) :=
  (Rmin3 v.1.1 v.1.2 v.2, Rmid3 v.1.1 v.1.2 v.2, Rmax3 v.1.1 v.1.2 v.2).
Proposition sort_coords_permutes1 : fun_permutes1_R3 sort_coords.
Proposition sort_coords_permutes2 : fun_permutes2_R3 sort_coords.
Theorem sort_coords_proper (v : R × R × R) :
  v.1.1 ≤ v.1.2 ≤ v.2 → sort_coords v = v.
```

We now have enough infrastructure to show that verification on the wedge is enough. We will first prove a partial result for σ-invariant sets.

```
Proposition prop_permutes_then_sorted_equiv (P : R × R × R → Prop) (v : R × R × R):
  coords_permute_R3 P → P (sort_coords v) → P v.
```

We prove prop_permutes_then_sorted_equiv by performing case analysis on all possible orders of the coordinates. We then permute the coordinates to the sorted order using the σ-invariance, and rewrite with sort_coords_proper to conclude.

With prop_permutes_then_sorted_equiv and mirror_implies_abs_enough, we can finally prove that the wedge is enough.

```
Theorem b3_invariant_wedge_enough (P : R × R × R → Prop) : B3_invariant P →
  (∀ v, 0 ≤ v.1.1 → v.1.1 ≤ v.1.2 ≤ v.2 → P v) →
  (∀ v, P v).
```

For the proof, we apply mirror_implies_abs_enough so that we now need to prove $(|x|, |y|, |z|) \in P$. We then apply prop_permutes_then_sorted_equiv. This means we now have to prove that the sorted version of $(|x|, |y|, |z|)$ is in P. By construction, all coordinates are nonnegative and sorted. This means the sorted version of $(|x|, |y|, |z|)$ is in P by assumption: it is in the wedge, and all elements in the wedge are in P. We apply mid_is_between to prove that these sorted coordinates are indeed in the wedge.

We are now in a position to prove that the basic invariants s_1, s_2 and s_3 are indeed B_3-invariant functions. We also show that $\|\cdot\|$ is B_3-invariant, since in our case the region defining function will be $s(x) = \|x\|^2 - 1$. First, we define the invariants in Coq.

```
Definition s1_B3 (v : R × R × R) := v.1.1 ^ 2 + v.1.2 ^ 2 + v.2 ^ 2.
Definition s2_B3 (v : R × R × R) := v.1.1 ^ 4 + v.1.2 ^ 4 + v.2 ^ 4.
Definition s3_B3 (v : R × R × R) := v.1.1 ^ 6 + v.1.2 ^ 6 + v.2 ^ 6.
```

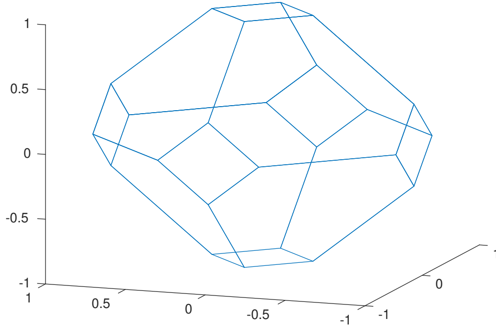


Figure 3.2: $\mathcal{K} - \mathcal{K}$ for $\mathcal{K}$ the truncated tetrahedron

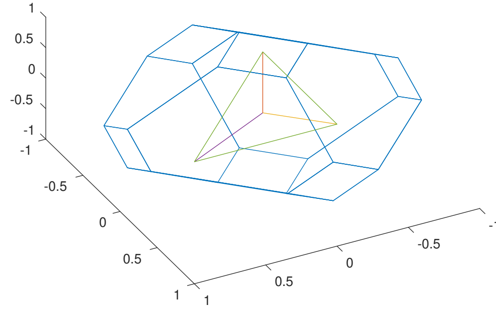


Figure 3.3: $\mathcal{K} - \mathcal{K}$ and the edges of the wedge

Now, we can show they are actually B_3-invariant. Note that `norm` automatically reduces to the norm on $\mathbb{R}^3$, since `B3_invariant_fun` requires a function with $\mathbb{R}^3$ as domain.

```
Proposition norm_R3_invariant : B3_invariant_fun norm.
Proposition s1_invariant : B3_invariant_fun s1_B3.
Proposition s2_invariant : B3_invariant_fun s2_B3.
Proposition s3_invariant : B3_invariant_fun s3_B3.
```

One of the benefits of the generator approach is that we only need to verify 5 equalities to prove a function is B_3-invariant. Coq comes provided with the convenient `ring` tactic, which can try to prove an equality on a ring, like $\mathbb{R}$. This makes proving the propositions above easy.

3.3.4.2. Analyzing the Minkowski difference on the wedge

We are interested in determining whether a given point in the wedge lies in the Minkowski difference of our shape $\mathcal{K}$. The Minkowski difference $\mathcal{K}^\circ - \mathcal{K}^\circ$ is B^3-invariant because of the tetrahedral symmetry of $\mathcal{K}$. The wedge is the fundamental domain of the symmetry group B_3, and thus relevant for B_3-invariant properties: if a B_3-invariant property holds in the wedge, it holds on all of $\mathbb{R}^3$. This suggests that some of the inequalities specifying $\mathcal{K}^\circ - \mathcal{K}^\circ$ are redundant for elements in the wedge.

We can confirm this by geometric inspection. We will investigate the case when $\mathcal{K}$ is the truncated tetrahedron. The shape $\mathcal{K} - \mathcal{K}$ can be found in Figure 3.2. In Figure 3.3, we can see $\mathcal{K} - \mathcal{K}$ along with the edge of the wedge. Notice that the wedge only intersects 2 of the 14 faces. This suggests that for elements in the wedge, we only need to check 2 inequalities to verify an element lies within $\mathcal{K} - \mathcal{K}$.

We can indeed prove this is true, and we have formally proven this for $\mathcal{K}$ equal to the truncated tetrahedron. The point we wish to make is that one can formulate precise, comparatively simple conditions determining whether an element lies within $\mathcal{K} - \mathcal{K}$, for every $\mathcal{K}$ with tetrahedral symmetry. It might be necessary to check more than two inequalities for more complicated shapes $\mathcal{K}$, but it will always be easier to determine whether $x \in \mathcal{K} - \mathcal{K}$ if we know that x is in the wedge. For illustrative purposes we will take $\mathcal{K}$ to be the truncated tetrahedron for the rest of this section.

Notice that one of the planes intersecting the wedge in Figure 3.3 is perpendicular to both the x and the y axis. To determine whether an (x, y, z) satisfies this inequality, we need only to check whether $z \leq c$ for some constant c. Stated in another way: $(x, y, z) \notin \mathcal{K} - \mathcal{K}$ if $z > c$. The inequality for the other plane can be written as

$$c_1 x + c_2 y + c_3 z \geq c_4$$

which is true whenever

$$c_3 z \geq c_4 - c_1 x - c_2 y.$$

We will have $c_3 < 0$, since (x, y, z) should not satisfy the inequality for large z. This means that (x, y, z) satisfies the inequality whenever

$$z \leq \frac{1}{c_3} \cdot (c_4 - c_1 x - c_2 y)\,.$$

We now define the function $l_2(x,y) = \frac{1}{c_3} \cdot (c_4 - c_1 x - c_2 y)$. Then, for (x,y,z) in the wedge, we have $(x,y,z) \notin \mathcal{K} - \mathcal{K}$ if $z > c$ or $z > l_2(x,y)$. Equivalently, we can check whether $z > \min(c, l_2(x,y))$.

We also want to know when $\|(x,y,z)\| \leq 1$. If (x,y,z) is in the wedge, $\|(x,y,z)\| \leq 1$ whenever

$$z \leq \sqrt{1 - x^2 - y^2}.$$

This gives us an explicit condition for given x, y and z with (x,y,z) in the wedge, to be in the verification region:

$$(x,y,z) \in \{\|x\| \leq 1\} \setminus (\mathcal{K} - \mathcal{K}) \quad \iff \quad \min(c, l_2(x,y)) < z \leq \sqrt{1 - x^2 - y^2}.$$

3.3.4.3. Cube lists are enough

Our goal is to prove that verification in all cubes of the provided cubelist is enough. Let us denote V for the verification region, defined as

$$V = \{\|x\| \leq 1\} \setminus (\mathcal{K} - \mathcal{K}).$$

Note that V differs from the actual region of verification

$$V' = \{\|x\| < 1\} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ).$$

This discrepancy will be explained at the end of this section. We will first prove that our function f is nonpositive on V, i.e. that

$$\forall (x,y,z) \in V : f(x,y,z) \leq 0,$$

and then show that nonpositivity on V implies nonpositivity on V' by a continuity argument. In Coq, this will be encoded into the equivalent statement

$$\forall (x,y,z) \in \mathbb{R}^3 : (x,y,z) \in V \implies f(x,y,z) \leq 0.$$

Let us denote $P(x,y,z)$ for the statement above. This means we wish to show $\forall (x,y,z) \in \mathbb{R}^3 : P(x,y,z)$. Using techniques developed in Section 3.3.4.1, we can show that P is a B_3-invariant predicate/set. This means that verification on the wedge is enough. Let us denote W for the wedge, defined by $W = \{(x,y,z) \in \mathbb{R}^3 : 0 \leq x \leq y \leq z\}$. Since verification on the wedge is enough, we have the implication

$$\left(\forall (x,y,z) \in \mathbb{R}^3 : (x,y,z) \in W \implies P(x,y,z)\right) \implies \left(\forall (x,y,z) \in \mathbb{R}^3 : P(x,y,z)\right).$$

We wish to cover W with cubes, prove that P holds on each cube, and conclude that P holds in W. We will not be able to prove this directly, so we will tackle this in several steps. Note first that for $x \geq \frac{1}{\sqrt{3}}$, every $(x,y,z) \in W$ has $\|(x,y,z)\| \geq 1$ and thus $(x,y,z) \notin V$. This is because $x \leq y \leq z$ for $(x,y,z) \in W$. We now have $P(x,y,z)$, since $(x,y,z) \notin V$. The condition of the implication does not hold, so the implication is valid. One can also see this if you read $A \implies B$ as the equivalent statement $\neg A \vee B$: in this case we have $\neg A$ since $(x,y,z) \notin V$. What is left to show is that

$$\forall (x,y,z) \in \mathbb{R}^3 : x \in \left[0, \frac{1}{\sqrt{3}}\right] \implies (x,y,z) \in W \implies P(x,y,z).$$

We define the shorthand $P_W(x,y,z)$ for "$(x,y,z) \in W \implies P(x,y,z)$", since in the remaining part of the section we will use this a lot. We will also use the more familiar shorthand $\forall x \in \left[0, \frac{1}{\sqrt{3}}\right] : A(x)$ instead of $\forall x \in \mathbb{R} : x \in \left[0, \frac{1}{\sqrt{3}}\right] \implies A(x)$, but remember that the latter form is how it is formalized in Coq.

We need some more information on the format of our cube list to continue. The $[0,1]^3$ region was divided into N^3 cubes for some positive $N \in \mathbb{N}$. Each cube then has sidelength $\frac{1}{N}$, and can be specified by 3 integers. A given triple $(i,j,k) \in \mathbb{N}^3$ corresponds to the cube

$$C(i,j,k) = \left[\frac{i}{N}, \frac{i+1}{N}\right] \times \left[\frac{j}{N}, \frac{j+1}{N}\right] \times \left[\frac{k}{N}, \frac{k+1}{N}\right].$$

We thus cover the interval $\left[0, \frac{1}{\sqrt{3}}\right]$ for x with m_1 intervals of width $\frac{1}{N}$. This means we wish to prove

$$\left[0, \frac{1}{\sqrt{3}}\right] \subseteq \bigcup_{i=0}^{m_1} \left[\frac{i}{N}, \frac{i+1}{N}\right].$$

This should hold if $\frac{1}{\sqrt{3}} \leq \frac{m_1+1}{N}$. What we actually prove is that for $x \in \mathbb{R}$,

$$x \in \left[0, \frac{m_1+1}{N}\right] \implies \exists i \in \mathbb{N} : i \leq m_1 \wedge x \in \left[\frac{i}{N}, \frac{i+1}{N}\right].$$

This can be shown to hold with a slightly modified version of `close_fraction_closed`. Using this, we can show the implication

$$\left(\forall i \in \mathbb{N} : i \leq m_1 \implies \left(\forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y, z \in \mathbb{R} : P_W(x, y, z)\right)\right) \implies \forall (x, y, z) \in \mathbb{R}^3 : P_W(x, y, z).$$

The first term is encoded in Coq as a list of propositions being true, using the techniques from Section 3.2.4. We will now continue by showing how we prove the desired property holds for each $i \in \mathbb{N}$ with $i \leq m_1$.

We will use the verification region V once again to determine an interval for which to perform verification. For $(x, y, z) \in W$ we have $x \leq y \leq z$. Furthermore, for $(x, y, z) \in V$ we have $\|(x, y, z)\| \leq 1$. This means that $y \leq \sqrt{\frac{1}{2} \cdot (1 - x^2)}$, since

$$x^2 + y^2 + z^2 \leq 1 \implies x^2 + y^2 + y^2 \leq 1 \implies y^2 \leq \frac{1}{2} \cdot (1 - x^2).$$

We thus get an interval for y for which we need to perform verification: $y \in \left[x, \sqrt{\frac{1}{2} \cdot (1 - x^2)}\right]$. Note that we need to use interval arithmetic once again here: we do not know the exact value for x, but we do know an interval containing x. Using the same divide and conquer strategy, let m_2 be such that $\sqrt{\frac{1}{2} \cdot (1 - x^2)} \leq \frac{m_2+1}{N}$. We then have the following implication

$$\left(\forall j \in \mathbb{N} : i \leq j \leq m_2 \implies \left(\forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y \in \left[\frac{j}{N}, \frac{j+1}{N}\right], \forall z \in \mathbb{R} : P_W(x, y, z)\right)\right) \implies \forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y, z \in \mathbb{R} : P_W(x, y, z).$$

For the final step, we will determine an interval which contains z. This has been discussed in the previous Section 3.3.4.2. For $(x, y, z) \in W$ and $(x, y, z) \in V$, we know that

$$\min(c, l_2(x, y,)) < z \leq \sqrt{1 - x^2 - y^2}.$$

We can evaluate both expressions with the current intervals for x and y. Note that we also require that $y \leq z$. Once again, we can find integers m_3 and m_4 for which $y \leq \frac{m_3}{N} \leq \min(c, l_2(x, y))$ and $\sqrt{1 - x^2 - y^2} \leq \frac{m_4+1}{N}$. We then get the implication:

$$\left(\forall k \in \mathbb{N} : m_3 \leq k \leq m_4 \implies \left(\forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y \in \left[\frac{j}{N}, \frac{j+1}{N}\right], \forall z \in \left[\frac{k}{N}, \frac{k+1}{N}\right] : P_W(x, y, z)\right)\right) \implies \forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y \in \left[\frac{j}{N}, \frac{j+1}{N}\right], \forall z \in \mathbb{R} : P_W(x, y, z).$$

The range for (x, y, z) in the condition of this implication is now in a cube! As mentioned before, it is exactly the cube $C(i, j, k)$. Rewriting, we get:

$$(\forall k \in \mathbb{N} : m_3 \leq k \leq m_4 \implies (\forall (x, y, z) \in C(i, j, k) : P_W(x, y, z))) \implies \forall x \in \left[\frac{i}{N}, \frac{i+1}{N}\right], \forall y \in \left[\frac{j}{N}, \frac{j+1}{N}\right], \forall z \in \mathbb{R} : P_W(x, y, z).$$

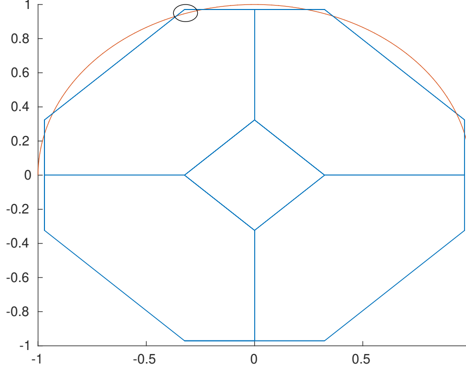


Figure 3.4: Part of $\mathcal{K} - \mathcal{K}$ outside the norm ball

This is what we were aiming for: once we have verified that P_W holds on cubes, we can show P_W holds on a larger region. We first show it holds on a 'column' of cubes, where there is no condition on the z coordinate. If P_W holds on enough columns, we can show it holds on a region with no conditions on the y and the z coordinate. From these we can go to the general case, that P_W holds everywhere.

The right values for m_1, m_2, m_3 and m_4 can be extracted from the provided cube list. Furthermore, we can prove the cubes are enough before starting the (computationally intensive) verification on the cubes.

There are some issues which still need to be addressed. For some values of $i, j \in \mathbb{N}$, there are no k for which (i, j, k) is in the cubelist. This happens when part of $\mathcal{K} - \mathcal{K}$ lies outside the ball with radius 1. Originally, $\mathcal{K} - \mathcal{K}$ lies wholly inside the ball with radius 1. However, the obtained solution f does not satisfy $f(x) \leq 0$ whenever $x \notin \mathcal{K} - \mathcal{K}$ since the solver cannot accurately detect the border of $\mathcal{K} - \mathcal{K}$. But we hope that for a factor $\alpha > 1$, $f(x) \leq 0$ whenever $x \notin \alpha\mathcal{K} - \alpha\mathcal{K}$.

This enlarged shape has some parts outside the ball with radius 1, as can be seen in Figure 3.4. To prove that P_W holds for these values of x and y, we need to show that there are no z values for which $(x, y, z) \in W \wedge (x, y, z) \in V$. It suffices to show that $\sqrt{1 - x^2 - y^2} \leq \min(c, l_2(x, y))$, since $\min(c, l_2(x, y)) < z \leq \sqrt{1 - x^2 - y^2}$ if and only if (x, y, z) is in W and in V.

Another issue which needs to be addressed is the size of the cubes. We have assumed in this section that all cubes in the list are the same size, but they are not. Cubes with larger sidelength have worse estimates for the norm of the gradient. When a better estimate is needed to prove nonpositivity, cubes are split into 8 smaller cubes, with half the sidelength. Splitting happens until the smallest cubes are sufficient. However, we need a proof that P_W holds on the original cube. Such a proof can be constructed from the proofs that P_W holds on all the subcubes using `close_fraction_closed` once more. Some of the subcubes might not lie in the verification region V or the wedge W. As such, these subcubes do not appear in the cube list, but we do need to show P_W holds on each of them. This can be done by contradiction on the correct assumption.

The first way for a cube to lie completely outside V, is if it lies inside $\mathcal{K} - \mathcal{K}$. We can prove the cube lies inside $\mathcal{K} - \mathcal{K}$ by proving all the vertices lie inside $\mathcal{K} - \mathcal{K}$, by convexity.

Another possibility for a cube to lie completely outside V, is if the norm of all elements in the cube are greater or equal than one. Suppose x is contained in a cube with lower-left corner v, and v lies in the wedge. We then have $\|v\| \leq \|x\|$, since v will be closer to the origin than x. By showing $\|v\| \geq 1$, we show that the cube is completely outside the norm ball, so P_W holds.

A final issue is that we have shown $f(x) \leq 0$ whenever $\|x\| \leq 1$ and $x \notin \mathcal{K} - \mathcal{K}$, while we needed to show this whenever $\|x\| < 1$ and $x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$. Although $\|x\| < 1$ implies $\|x\| \leq 1$, the implication is in the wrong direction for $x \notin \mathcal{K} - \mathcal{K} \implies x \notin \mathcal{K}^\circ - \mathcal{K}^\circ$.

We can overcome this problem since our function f is continuous. We know that f is nonpositive on some set, so by continuity it will be nonpositive on the closure of this set. We will first prove the following two theorems.

Theorem 21. *Let U be a uniform space and $f : U \to \mathbb{R}$ be a continuous function. Let A be a subset of U. Suppose $f(x) \leq 0$ for all $x \in A$. Then $f(x) \leq 0$ for $x \in \overline{A}$, where $\overline{A}$ denotes the closure of A.*

Proof. Let $x \in \overline{A}$, and suppose $f(x) > 0$. We aim to find a contradiction. Since f is continuous, we can find some $\delta > 0$ such that for all $y \in B_U(x, \delta)$ we have $f(y) \in B_{\mathbb{R}}(f(x), f(x))$. Now

$$B_{\mathbb{R}}(x, f(x)) = \{y \in \mathbb{R} : |f(x) - y| < f(x)\},$$

which means that $|f(x) - f(y)| < f(x)$ for $y \in B_U(x, \delta)$. If $f(x) - f(y) \geq 0$, it follows that $0 < f(y)$. For $f(x) - f(y) < 0$, we get $0 < f(x) < f(y)$, so in both cases we get that $0 < f(y)$ for $y \in B_U(x, \delta)$.

Since x lies in the closure of A, every open set containing x intersects A. This means $B_U(x, \delta) \cap A \neq \emptyset$. So, take $y \in B_U(x, \delta) \cap A$. Then $f(y) > 0$, since $y \in B_U(x, \delta)$, and $f(y) \leq 0$, since $y \in A$. We have reached a contradiction, so $f(x) \leq 0$. □

Theorem 22. *Let U be a uniform space, and A and B subsets of U. Suppose A is open. Then*

$$A \cap \overline{B} \subseteq \overline{A \cap B}.$$

Proof. Let $x \in A \cap \overline{B}$. We will prove that $x \in \overline{A \cap B}$. Let O be an open set with $x \in O$. It suffices to show that there is some $y \in A \cap B$ with $y \in O$. Since A and O are open, $A \cap O$ is open as well. Since x lies in the closure of B, every open set containing x intersects B. This means we have $(A \cap O) \cap B \neq \emptyset$. Thus, there exists some $y \in B$ with $y \in A \cap O$. Or, rearranging: we have $y \in A \cap B$ and $y \in O$. □

We will now apply these theorems to our case. Note that $(\mathcal{K}^\circ - \mathcal{K}^\circ)^c$ is the closure of $(\mathcal{K} - \mathcal{K})^c$. Let $x \in \mathbb{R}^3$ and suppose

$$x \in \{y \in \mathbb{R}^3 : \|y\| < 1\} \cap (\mathcal{K}^\circ - \mathcal{K}^\circ)^c = \{y \in \mathbb{R}^3 : \|y\| < 1\} \cap \overline{(\mathcal{K} - \mathcal{K})^c}.$$

This means $x \in \overline{\{y \in \mathbb{R}^3 : \|y\| < 1\} \cap (\mathcal{K} - \mathcal{K})^c}$ by Theorem 22, where we used that $\{y \in \mathbb{R}^3 : \|y\| < 1\}$ is an open set. Since we have verified that $f(x) \leq 0$ for $x \in \{y \in \mathbb{R}^3 : \|y\| < 1\} \cap (\mathcal{K} - \mathcal{K})^c$, we get $f(x) \leq 0$ as a result of Theorem 21.

3.4. Auxiliary verification tools

In this section we will describe additional tools and scripts that were used for the verification. Section 3.3 armed Coq with the required results for the verification, this section aims to explain how everything was made ready for the actual verification.

We have tried to minimize the manual work needed for verifying different shapes $\mathcal{K}$. For each shape $\mathcal{K}$, one has to provide a Coq file defining $\mathcal{K} - \mathcal{K}$. This file should also prove $\mathcal{K} - \mathcal{K}$ is B_3-invariant and a simplification of $\mathcal{K} - \mathcal{K}$ on the wedge, as described in Section 3.3.4.2. Additionally, one has to provide tactics for verification on the two types of cubes: those intersecting $\mathcal{K} - \mathcal{K}$, and those not intersecting. These tactics are easy to construct with the results from Section 3.3.2 and Section 3.3.3. Finally, tactics are needed for automatically proving the implications that cubelists are enough, mentioned in Section 3.3.4.3. Some manual work is needed here, since the exact inequalities will differ per shape $\mathcal{K}$.

Once these files have been built, the tools and scripts will automatically build the rest of the required Coq files for verification. The following five parts of this automation are the most important:

- Define the polynomial in Coq from a solution representation file;
- Build an alternative representation of the polynomial, and prove they are equal;
- Build an interval_container for the gradient, since letting Coq build it takes a long while;
- Generate files which perform the actual computation on the cubes;
- Distribute the load of the verification over the available computers.

Most of these tools were written in Python. Some additionally use the Sage Mathematics Software System [14], which is based on Python.

3.4.1. Polynomial definition from a solution file

Dostert et al. [5] provided solution files for various shapes $\mathcal{K}$. These files encode the coefficients of the polynomial $\widehat{f}$. The verification performed by Dostert et al. also needed to translate these solution files to C++ source files, which would evaluate f or ∇f. They used Sage to construct f from $\widehat{f}$, and we were able to reuse most of their Sage programs.

It is important to note that mapping $\widehat{f}$ back to f cannot be done exactly. The coefficients of $\widehat{f}$ are given as floating point numbers, which are fractions. Taking the Fourier inverse involves operations with π, an irrational number. This means that we do not have exact definitions of the coefficients of f as a floating-point number; we only know these coefficients lie in some interval. This means that the definition of our function $f\colon \mathbb{R}^3 \to \mathbb{R}$ in Coq will be abstract. This is fine, as long we are able to use interval arithmetic to prove bounds on f. We will define every coefficient as a `Parameter`, an abstract term. We add `Conjecture`s for each bound on each coefficient. In the Coq system, we are defining new 'axioms' for each coefficient.

Ideally, it would be best if the original coefficients of $\widehat{f}$ were entered directly into Coq. Then the Fourier inverse of $\widehat{f}$ could be calculated explicitly, and bounds could be calculated with interval arithmetic. This would require formalization of the Fourier transforms being used, and was outside the scope of this thesis.

Another hurdle was that one is not able to enter floating point values (i.e. $3.1415\ldots$) in Coq. Instead, given floating point values must first be converted to explicit rationals, whose denominator is a power of 2.

3.4.2. Alternative representation of polynomial

The polynomial f is defined as a polynomial over the basic invariants s_1, s_2 and s_3. As mentioned in Section 3.3.1, a direct gradient of this expression does not provide sharp enough bounds with interval arithmetic. We need to get a representation of f as a polynomial in the coordinates x, y and z, but the coefficients of this representation should be defined in terms of the original coefficients. This allows us to prove the two representations are equal.

We wrote a Sage program which gives us this alternative representation. Sage has support for symbolic computations and polynomials over arbitrary rings, so is well-suited to the task. The coefficients of f in the x, y and z coordinates can then be expressed as symbolic expressions of the coefficients in the s_1, s_2 and s_3 invariants. Coq can prove the two expressions are equal, prove bounds on these new coefficients and hide the definition of the new coefficients.

Since we have bounds on the original coefficients, we can use these to prove bounds on symbolic expressions of the coefficients. Sage also generates bounds on the coefficients, and these should be compatible. However, they differ very slightly. This was caused most likely by a slight difference in the (order of) calculation of the new bounds in Sage and in Coq. To overcome this, we verified the bounds in Coq with 300 bits of precision, which is more than the 256 bits precision used in Sage.

3.4.3. Manual expressions for interval_containers

Before being able to compute an expression f with interval arithmetic, we must convert our expression to some list of computations `f`. When constructing an interval_container, this happens automatically, although not instantly. The automation we built is sufficient for the input polynomial f, but starts to become troublesome when we compute the partial derivatives of f. It gets worse for the expression

$$\|\nabla f\| = \sqrt{\left|\frac{\partial f}{\partial x}\right|^2 + \left|\frac{\partial f}{\partial y}\right|^2 + \left|\frac{\partial f}{\partial z}\right|^2},$$

since it contains three partial derivatives. This expression is so large that conversion to a list of computations in Coq takes an unacceptable amount of time; we let it run for a couple of hours, and it did not terminate.

To tackle this, we delegate this computation of `f` from f to another program. We created a Python program which computes this list in a couple of seconds, and can generate a Coq file which uses this list of computations. The Python program needs the 'reification' of the expression, which is computed by Coq. This makes it a bit harder, since this means we need to be able to automatically feed Coq output into Python. Some scripts were created to launch the Coq command line interface, enter the appropriate commands and then feed this output automatically to Python.

The algorithm used in Python is the same as the one used in Coq, but Python appears to be much faster than Coq. Still, the computation is not instant. The list of computations only needs to be generated once, so this is acceptable.

3.4.4. Generating the verification files

Once all polynomial files have been generated and all tactics are ready, we just need files which will perform the actual verification. Another Python script was written to take care of this. Given a file specifying a cube list, it generates all the files necessary to start the verification.

Each cube in the provided cubelist came with the following information:

- A triple of natural numbers i, j and k;
- A gridsize $N > 0$, specifying the amount of points on the grid along one axis. The nonpositivity check should work for this gridsize;
- A boolean value, indicating whether or not this cube intersects the Minkowski difference $\mathcal{K} - \mathcal{K}$;
- The 'depth' of the cube, an integer n.

This information corresponds to the cube

$$\left[\frac{i}{300 \cdot 2^n}, \frac{i+1}{300 \cdot 2^n}\right] \times \left[\frac{j}{300 \cdot 2^n}, \frac{j+1}{300 \cdot 2^n}\right] \times \left[\frac{k}{300 \cdot 2^n}, \frac{k+1}{300 \cdot 2^n}\right].$$

Here 300 is the initial number of the cubes per side of our cubelist, not a fixed value.

The file generation is done in a way similar to the approach in Section 3.3.4.3. A folder is created for each i value, corresponding to the case where x is in the interval $[\frac{i}{300}, \frac{i+1}{300}]$. This folder contains a 'conclusion' file, which should prove that P_W holds for all values of y and z, and x in the interval mentioned before. It relies on other files in this folder, which prove this for x in this same interval and y in some specified interval. Each of those files relies again on results proving it holds for a list of cubes. Each of these cubes may rely again on results for subcubes, which all have their own file.

One might want to limit the computation time per file. This makes finding mistakes easier, and saves more intermediate results to disk. This means we lose less when the verification stops unexpectedly. We limit the computation time per file by dividing the verification for a list of cubes over multiple files. In our experience, verification at a gridpoint takes about 0.6 seconds of computing time. This allows us to estimate the total computation time of a file. We chose to limit this to 5 minutes per file, starting a new file every time computation for the current file would exceed 5 minutes. Files which need all of the results need to import each of these files.

A note is in order on how we apply the results for each cube to prove that P_W holds for a list of cubes. We do not want to explicitly state the name of the proposition for each element on the list. Instead, we add each result to a 'Hint Database'. When proving P_W holds in a list of cubes, we can tell Coq to look for the required result in the provided hint database. If it finds a matching result, it automatically solves the goal.

3.4.5. Dividing the verification load

Verification is a computationally intensive task, but can be parallelized in our case. This is because verification in one cube is totally independent of the verification in another cube. There are several ways to divide tasks, and one way is to divide the i values over computers. This means that you divide the folders to verify over different computers: each computer verifies P_W for a different set of intervals for x.

In our case, 8 computers were available to do verification. A greedy load distribution algorithm tried to evenly divide the total computations over the computers. Total computations for each folder can be estimated by calculating the number of gridpoints for each cube. The greedy distribution was fair enough for our purposes, so there was no need to optimize further.

Each computer we used had multiple cores available for verification. The verification procedure consists of 'compiling' a large amount of files with Coq. Most of these files, even those in the same folder, are not interdependent, and can thus be compiled in parallel. Coq comes with a utility called CoqMakefile, which can automatically detect dependencies between Coq files, and compile them in the correct order. We used the build automation tool 'Make' to call CoqMakefile, which would then compile the required files. Make has support for parallel builds, for which one only needs to specify the number of processors Make is allowed to use. This allowed us to perform the verifications on 8 computers, each

using 3 processors at the same time. Once the computers were done, the compiled files can be imported and reused on other computers.

4

Conclusion

In this chapter, we will cover the results of the formal verification we set out to perform. After the results, we will discuss the validity of the results and improvements that could be made. We will also discuss our experiences with the proof assistant Coq. Finally, we make some suggestions for further work.

4.1. Results

Using the techniques developed in Chapter 3, we were able to verify that

$$f(x) \leq 0 \qquad \text{if } x \in \{x \in \mathbb{R}^3 : \|x\| < 1\} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ),$$

where $\mathcal{K}$ is the truncated tetrahedron, and f is the corresponding polynomial found by Dostert et al. [5]. The exact statements and definitions that were verified in Coq can be found in Appendix B. The coefficients of the polynomial f are known to be in some interval. For each coefficient, we added an axiom stating that the coefficient is contained in the corresponding interval. The verification took about a week of computation time, using 8 computers with each 3 cores running simultaneously. Recall that the third condition of Theorem 1 was

$$f(x) \leq 0 \qquad \text{whenever } \mathcal{K}^\circ \cap (x + \mathcal{K}^\circ) = \emptyset,$$

which is equivalent to

$$f(x) \leq 0 \qquad \text{if } x \notin \mathcal{K}^\circ - \mathcal{K}^\circ.$$

This means we still have to check whether $f(x) \leq 0$ in the region $\{x \in \mathbb{R}^3 : \|x\| \geq 1\} \setminus (\mathcal{K}^\circ - \mathcal{K}^\circ)$. However, this follows from the construction of the polynomial f; it can be written as

$$f(x) = (-s(x)q_1(x) - q_2(x)),$$

where $q_1(x)$ and $q_2(x)$ are sums of squares, and $s(x) = \|x\|^2 - 1$. Since q_1 and q_2 are nonnegative, we get that $f(x) \leq 0$ whenever $s(x) \geq 0$; whenever $\|x\| \geq 1$.

Reintroducing the exponential factor, this proves that $(-s(x)q_1(x) - q_2(x))e^{-\pi\|x\|^2}$ satisfies the conditions of Theorem 1, so provides an upper bound (in this case, of $0.7292\ldots$) on the translative packing density of truncated tetrahedra.

The techniques developed in Chapter 3 can easily be re-used to formally verify the upper bounds of Dostert et al. for other shapes. We chose to verify the truncated tetrahedron, since we estimated this verification to be the least computationally intensive. To verify the other shapes, one would need to define $\mathcal{K} - \mathcal{K}$ in Coq, and prove some required properties of the shape: closedness, boundedness, convexity, and a simplification on the wedge. Some minor tweaking will be needed for the verification tactics, substituting the right function names. After tweaking, one only requires computing time. In Figure 4.1, one can find an estimate of the verification time required for the remaining shapes.

	processor-hours	processor-days, when using 24 cores
truncated tetrahedron	4500	8
truncated cube	6700	12
truncated cuboctahedron	11600	20
rhombicuboctahedron	14000	24

Figure 4.1: Estimated processing time required for different shapes

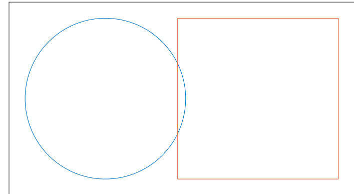


Figure 4.2: Squares vertices are outside the unit ball, but it contains elements inside the unit ball

During the development of the verification tools, we discovered one reasoning problem in the original verification program of Dostert et al. There is a procedure which discards cubes if the program decides that the cube does not intersect the verification region. When a cube is outside the unit ball, it is outside the verification region. The program would discard the cube when all of its vertices were outside the unit ball. However, this is not always valid! See Figure 4.2. This reasoning problem did not cause issues, since it turns out to be valid in the wedge. When the lower-left corner of a cube is inside the wedge, the norm of elements of that cube is greater or equal than the norm of the lower-left corner. When all vertices of a cube are outside the norm ball, the lower-left corner is outside the norm ball in particular. This means that the procedure in the original program gave valid results, even though the reasoning was not entirely sound.

The formal verification required some mathematics that was not yet available in Coq. Examples of this are results on convexity, mean-value theorems in general normed spaces, and results on lists of propositions. The formalizations built for this thesis could be packaged and reused for other projects. Proof assistants get easier to use when more math is formalized, so sharing packages helps other formal verification efforts.

The formal verification also called for a more efficient way of doing floating point interval arithmetic in Coq. We built the interval_container structure to accomodate this. Developing such a method is quite different from formalizing mathematics, where one does not need to worry about efficiency. The interval_container structure gives a significant speed boost when the same calculation needs to be performed a large amount of times. This could be also be helpful in other formal verification efforts.

4.2. Discussion

We will cover three topics in this section. We will first discuss the validity of the results. Then, we will mention some possible improvements for the techniques we developed. Finally, we will briefly cover our experience with the proof assistant Coq.

4.2.1. Validity of results

Verifying a result with a proof assistant should give total certainty the result is true. However, there are severaly reasons why one could disagree with the verification of the result. One could disagree with the axioms on which the result is based. One could also distrust the proof assistant. Finally, one could accept the verification that has been done in the proof assistant, but none of the results outside the proof assistant.

The validity of a formally verified result stands or falls based on whether one trusts the assumptions

of the result. Coq comes with a command to print the assumptions of a given result. This will list all the axioms used to prove the result.

The final result uses a lot of intermediate results, so that querying Coq to print all used axioms takes a long time. When we ask Coq what the assumptions are for a small subset of the results, it returns the following axioms:

- axioms of the real numbers;
- axioms specifying the bounds on the coefficients of f;
- axiom stating the law of excluded middle;
- for some versions: axioms stating that 'native' 64-bit integers behave like the corresponding inductive type.

The real numbers defined in the standard library of Coq are defined using axioms. This could be improved upon by redefining the real numbers. The axioms specifying the bounds on the coefficients of f can be proven explicitly, which would require formalization of the Fourier inverse and some interval arithmetic.

Another axiom we have used is the law of excluded middle. Although we did not use it in our proofs, it is used to prove the mean value theorem. We do not need the mean value theorem itself, but only a consequence: bounded variation. What we have called bounded variation is sometimes referred to as the mean value inequality. This inequality can be derived without the law of excluded middle, using just the axioms on the real numbers we have available. A constructive proof for the mean value inequality can be found in Tucker [16].

The last set of axioms state that 'native' 64-bit integers behave like their corresponding inductive type. Whether these axioms actually appear in the assumption seems to depend on the version of the packages one uses. The verification that was carried out completely used Coq version 8.8.0 and did not need to assume these axioms. Coq version 8.9.0 did assume these.

By 'native' integers, we mean integers in OCaml, outside of Coq. Coq was implemented using another programming language: OCaml. This is a functional programming language, and like most programming languages it has access to integer arithmetic on the processor of the computer. One can axiomatize that operations like addition and multiplication on the processor return the same result as when doing the multiplication in Coq. By assuming these axioms, we state that we 'trust' the processor on the machine we use. If we trust the processor, we can perform integer calculations on the processor, which should increase the speed of such calculations. The 64-bit integer type is used in the formalization of floating point arithmetic in Coq, which is used by the interval arithmetic package.

All generated verification files of the truncated tetrahedron together take up about 23 gigabytes of disk space. This becomes a problem when we want to use all the intermediate results to prove the final result. To do this, we need to load all results into memory. The results that are saved on disk are compressed, so that we need more than 23 gigabytes of memory to load the entire verification into memory: we estimate one would need about 52 gigabytes of memory. This would be worse for other shapes, where the verification that needs to be performed is larger.

To overcome this, we divided the total result into 8 smaller results, corresponding to the statement being valid in 8 sets of intervals for x. Each of these smaller results was small enough to be verified: all required intermediate results fit in the available memory. Then, we created files where these exact 8 results were assumed as an axiom, and proved the final result from these axioms. It is currently not possible to work around this memory problem, so that for computers with limited memory the verification can only be performed in two steps.

Coq comes with a tool called 'coqchk'. This tool was built to verify compiled Coq files. It aims to be a small program, so that humans could inspect the source code and agree that it is valid. By trusting the verification tool, we trust all the results it verifies. We could use this tool to verify our results after compilation. However, running coqchk is at least as computationally intensive as the verification itself. Especially when verifying results relying on interval arithmetic, the verification seems to take more than 200 times as long. This makes re-verification of the compiled files with coqchk practically impossible, which is unfortunate, as the main argument for trusting the results is trusting coqchk.

Finally, we only verified a bounded part of the third condition of Theorem 1. If the other conditions are not satisfied for our polynomial f, we do not get an upper bound on the translative packing density

of the truncated tetrahedron. To establish complete trust in this upper bound, we would need to formally verify the other conditions hold as well.

4.2.2. Possible improvements

As mentioned in the previous section, we currently rely on the (non-constructive) mean value theorem. This requires the classical axiom of the law of excluded middle. Bounded variation is the consequence of the mean value theorem we actually use. It is possible to prove bounded variation constructively, see Tucker [16]. We could therefore change the techniques developed in Chapter 3 to drop the requirement of the law of excluded middle.

There could alse be room for improvement for the interval_container structure. Experimentally, the time taken by one interval arithmetic calculation is about 0.6 seconds. However, this time seems to be independent of the complexity of the expression. Proving the norm of the gradient of f is lower than some number takes about the same amount of time as proving a linear inequality with 4 variables. This is odd, since the first expression requires about 1000 times as many floating point calculations. This suggests starting an interval arithmetic calculation takes some constant amount of time, which is a large part of the 0.6 seconds. Further analysis may provide a way to speed this up.

4.2.3. Experience with Coq

Our experience with Coq has been pleasant for the most part. Coq is easily powerful enough to prove all results we needed. The support for custom automation was also very helpful.

Coq is not without its flaws. We especially had some trouble debugging custom Ltacs. Coq can also be quite slow, without apparent reason.

Proving results in Coq takes more effort than just writing the proof down. This is a disadvantage, but it does stimulate rethinking the proof. We often found ourselves trying to prove some particular result, which follows from a more general result. It is often easier in Coq to first prove this general result, and then derive the particular consequence. In this way Coq pushes the user to prove more general, and often more beautiful results.

4.3. Further work

The techniques developed in Chapter 3 should be easily adaptable for other shapes $\mathcal{K}$. After adapting, one only needs computing time to verify the results of Dostert et al. [5] are valid for shapes other than the truncated tetrahedron.

It would also be nice to formally verify the other conditions of Theorem 1 are satisfied in Coq, as well as proving Theorem 1 itself in Coq. This would prove the results of Dostert et al. [5] beyond any shred of doubt. This would still require quite a bit of work:

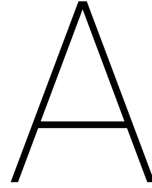
- formalizing the Fourier transform of B_3-invariant functions;
- formally verifying that we indeed have $f(x) = -s(x)q_1(x) - q_2(x)$;
- formally verifying that $\mathcal{K} - \mathcal{K}$ coincides with a set of linear inequalities.

The computation of the Fourier transform for B_3-invariant polynomials can be simplified by using the B_3-invariance. The Fourier transform itself also still needs to be formalized for the spaces defined by Coquelicot.

One could verify that $f(x) = -s(x)q_1(x) - q_2(x)$ with an approach similar to the one we used for proving that f as a polynomial of the basic invariants s_1, s_2, and s_3 can be written as a polynomial of x, y, and z. However, this will probably break down when we define f as the Fourier inverse of $\widehat{f}$. We want to define f in this way, since we know (the coefficients of) $\widehat{f}$ explicitly. To formally verify that $f(x) = -s(x)q_1(x) - q_2(x)$, we would need to use the approach described by Dostert et al. [5]. Here $f(x) = -s(x)q_1(x) - q_2(x)$ can be interpreted as a condition of a semidefinite programming problem. Then one checks that the maximum constraint violation of this problem is much smaller than the minimum eigenvalues of the matrices involved.

Finally, proving that $\mathcal{K} - \mathcal{K}$ is equal to the set defined by some linear inequalities also poses some problems. Minkowki addition and subtraction behave nicely when talking about $\mathcal{K}$ as a convex combination of vertices. What we then need to prove, is that values satisfying the given set of linear inequalities are precisely convex combinations of some set of vertices. One approach is to do this explicitly, with

an exhaustive case analysis. This would not be very scalable for different shapes, but it would work. A nicer approach is to use the Krein-Milman theorem, or a version of the Krein-Milman theorem in $\mathbb{R}^n$. The Krein-Milman theorem states that every compact closed convex set is the convex combination of its extreme points. One could then prove that the given vertices are the only extreme points of $\mathcal{K} - \mathcal{K}$ to conclude. The proof of Krein-Milman relies on the axiom of choice for general topological vector spaces. By limiting oneself to $\mathbb{R}^n$, we do not need the axiom of choice.



Auxilliary formal definitions and propositions

A.1. Linear combinations

To be able to define convex combinations, we need to define linear combinations.

```
Fixpoint linear_combination (coeffs: list K) (vectors: list U): U :=
  match vectors with
  | nil ⇒ zero
  | cons v vectors ⇒
    match coeffs with
    | nil ⇒ zero
    | cons c coeffs ⇒
      plus (scal c v) (linear_combination coeffs vectors)
    end
  end.
```

Note that we do not require `coeffs` and `vectors` to have the same length. This allows more flexibility in using `linear_combination`. If `coeffs` and `vectors` have different length, the recursion stops as soon as one of them is used up. The result of `linear_combination` is therefore equal when applied to the same arguments extended with `zero : K` or `zero : U` respectively.

Let (c_n) be a list of elements of K and (v_n) be a list of elements of U, not necessary of equal size. Let $S(l)$ denote the length/size of a list l. Then the mathematical notation of `linear_combination` (c_n) (v_n) would be:

$$\sum_{i=1}^{\min(S((c_n)),S((v_n)))} c_i v_i$$

```
Proposition empty_vectors (coeffs : list K) :
  linear_combination coeffs nil = zero.
```

```
Proposition empty_coefficients (vectors : list U) :
  linear_combination nil vectors = zero.
```

The proof of both `empty_coefficients` and `empty_vectors` consists just of performing `destruct` on the argument that is not `nil`, so that `linear_combination` can be evaluated.

```
Theorem combination_scal (coeffs : list K) (k : K) (vectors : list U) :
  scal k (linear_combination coeffs vectors)
    = linear_combination (map (fun r ⇒ mult k r) coeffs) vectors.
```

`combination_scal` is Theorem 19.

```
Theorem zero_combination (coeffs : list K) (vectors : list U) :
  all_valid (map (fun k ⇒ k = zero) coeffs)
    → linear_combination coeffs vectors = zero.
```

`zero_combination` states that for (c_i) a list in K and (v_i) a list in V, we have

$$\sum_{i=1}^{\text{minsize}} c_i v_i = 0_U \quad \text{if } \forall i : c_i = 0_K.$$

This follows from performing induction on `vectors` on the (c_n) quantified statement, and the previous equalities. We also need that scalar multiplication with 0_K yields 0_U.

```
Theorem trunc_coeffs_equal (coeffs : list K) (vectors : list U) :
  linear_combination (take (size vectors) coeffs) vectors
    = linear_combination coeffs vectors.
```

`trunc_coeffs_equal` is a technical theorem, but necessary for manipulating `linear_combinations` in Coq. Remember that we had no requirements on the lengths of `coeffs` and `vectors`. It is therefore perfectly valid that the length of `coeffs` is way larger than the length of `vectors`. The result of `linear_combination` would be the same if we were to input only the first elements of `coeffs`, and this is what the theorem states. Here `size list` computes the length of list `list`, while `take n list` returns the list with the first `n` elements of `list`. Note that if `n` $\geq$ `size list`, `take n list` = `list`.

```
Theorem combination_zip (c1 c2 : list K) (vectors : list U) :
  linear_combination (map (fun pr ⇒ plus pr.1 pr.2) (zip c1 c2)) vectors
    = plus (linear_combination (take (size c2) c1) vectors)
           (linear_combination (take (size c1) c2) vectors).
```

To explain `combination_zip`, we need to discuss `zip` and `map` first. The function `zip` sends two lists to one list of pairs. This means `zip` [:: 1; 2] [:: 3; 4; 5] will evaluate to [:: (1,3); (2,4)]; it breaks off as soon as one list runs out of elements. The function `map` takes as arguments a function of type `A` → `B`, and a list of type `list A`, to produce a list of type `list B`. It creates this list by applying the function to every element of the list, and concatenating the results.

Combining all this, `combination_zip` states that for two lists (a_n) and (b_n) of K and a list (v_n) of U, we have

$$\sum_{i=1}^{\text{minsize}} (a_i + b_i) v_i = \left(\sum_{i=1}^{\min(S((b_n)), S((v_n)))} a_i v_i \right) + \left(\sum_{i=1}^{\min(S((a_n)), S((v_n)))} b_i v_i \right).$$

Note that we do not use minsize on the right hand side, since we truncate the linear combination of $(a_n), (v_n)$ to the size of (b_n) and vice-versa. In `combination_zip` this is captured by using `take (size c2) c1`. The formula could also be stated in a way where we require `c1` to be a shorter list than `c2`, but this formulation requires less hypotheses.

The proof follows from induction on `vectors` on the (a_n) and (b_n) quantified statement. It also uses commutativity of addition in U.

A.2. Sums

Sums make sense on any group. They have nicer properties for abelian groups, and Coquelicot provides us with a structure for abelian groups. We will therefore define sums on abelian groups. In the rest of this section, G will be an abelian group.

```
Fixpoint sum (l : list G) :=
  match l with
  | nil ⇒ zero
  | cons a l ⇒ plus a (sum l)
  end.
```

Let `l1` and `l2` be lists in `G`. Summing each list individually and adding the result is the same as summing the concatenation of two lists.

```
Theorem cat_sum (l1 l2 : list G) : sum (l1 ++ l2) = plus (sum l1) (sum l2).
```

The proof follows from induction on `l1`.

The theorem above holds also for non-abelian groups. The following theorem does not.

```
Theorem zip_sum (l1 l2 : list G) :
  sum (map (fun pr ⇒ plus pr.1 pr.2) (zip l1 l2))
```

```
    = plus (sum (take (size l2) l1)) (sum (take (size l1) l2)).
```

Compare zip_sum with combination_zip. The proof follows from induction on l1, on the l2 quantified statement.

```
Theorem scal_sum (l : list V) (k : K) :
  scal k (sum l) = sum (map (fun r ⇒ scal k r) l).
```

scal_sum is the analogue of combination_scal for sums. Indeed, we can also write a linear combination as a sum. This is what lincomb_eq_sum states.

```
Theorem lincomb_eq_sum (l : list V) (c : list K) :
  sum (map (fun pr ⇒ scal pr.1 pr.2) (zip c l)) = linear_combination c l.
```

For convex combinations, we will use sums of real numbers. Real numbers come with an ordering. We can deduce properties of the sum of a list of numbers if we have information on the ordering of the elements of the list.

```
Theorem listpos_sumpos (l : list R) :
  all_valid (map (fun r ⇒ 0 ≤ r) l) → 0 ≤ sum l.
Theorem sumbound_listpos_listbound (l : list R) (b : R) :
  all_valid (map (fun r ⇒ 0 ≤ r) l) →
  sum l ≤ b →
    all_valid (map (fun r ⇒ r ≤ b) l).
Theorem sumzero_listpos_listzero (l : list R) :
  all_valid (map (fun r ⇒ 0 ≤ r) l) →
  sum l = 0 →
    all_valid (map (fun r ⇒ r = 0) l).
```

All these theorems are not very hard to prove. We need to perform list induction on the provided list, and manipulate the real numbers involved to get the required inequalities.

A.3. Affine functions

Affine functions map elements of one module space to another module space of the same ring. We define them by their behaviour when applied to sums of arguments and a scalar multiple of an argument. Here K will be a ring, and U and V module spaces over K.

```
Record is_affine (l : U → V) := {
  affine_plus : ∀ (x y : U), l (plus x y) = minus (plus (l x) (l y)) (l zero) ;
  affine_scal : ∀ (k : K) (x : U),
      l (scal k x) = plus (scal k (minus (l x) (l zero))) (l zero) }.
```

We can now show that affine functions are indeed closely related to linear functions.

```
Proposition lin_is_affine (l : U → V) : is_linear l → is_affine l.
Proposition lin_plus_is_affine (l : U → V) :
  is_linear (fun u ⇒ minus (l u) (l zero)) → is_affine l.
```

Finally, we show affince functions behave nicely when applied to affine combinations. If the ring contains an element that is half the unit, behaving nicely on affine combinations is enough to prove a function is affine.

```
Proposition affine_combination (l : U → V) (_ : is_affine l) (k : K) (u1 u2 : U) :
  l (plus (scal (minus one k) u1) (scal k u2)) =
     plus (scal (minus one k) (l u1)) (scal k (l u2)).
Proposition affine_comb_enough (l : U → V) :
  (∃ (half : K), plus half half = one) →
  (∀ (k : K) (u1 u2 : U), l (plus (scal (minus one k) u1) (scal k u2)) =
                          plus (scal (minus one k) (l u1)) (scal k (l u2))) →
    is_affine l.
```

B

Verified results for the truncated tetrahedron

The result that was verified for the truncated tetrahedron is the following.

```
Theorem solution_sat :
  ∀ x : R × R × R, norm x ≤ 1 → ¬ ttetra x → solution_pair x ≤ 0.
```

The continuity argument shown in Section 3.3.4.3 gives us the following consequence:

```
Definition ttetra_proper (x : R × R × R) := closure (fun v ⇒ ¬ ttetra v) x.

Proposition theorem_solution_sat3 :
  ∀ x : R × R × R, norm x < 1 → ttetra_proper x → solution_pair x ≤ 0.
```

This last statement corresponds exactly to the bounded part of the third condition of Theorem 1 where $\mathcal{K}$ is equal to the truncated tetrahedron, since $\overline{(\mathcal{K}-\mathcal{K})^c} = (\mathcal{K}^\circ - \mathcal{K}^\circ)^c$.

The Minkowski difference was defined as the intersection of the following 14 half-spaces, each defined by a linear inequality. Here `blowup` is the factor $\alpha > 1$, with which we must enlarge $\mathcal{K}-\mathcal{K}$. This is needed since `solution_poly` will not satisfy the nonpositivity condition near the edge of $\mathcal{K}-\mathcal{K}$. The `scaleDown` factor is used to scale $\sqrt{40}(\mathcal{K}-\mathcal{K})$ down, so that its vertices have at most norm 1. Note that the coefficients of the linear inequalities for $\sqrt{40}(\mathcal{K}-\mathcal{K})$ are integers.

```
Definition blowup := 1023 / 1000.
Definition scaleDown := / sqrt(40).

Definition ineq1 v := -1 × v.1.1 + 0 × v.1.2 + 0 × v.2 ≥ -6 × scaleDown × blowup.
Definition ineq2 v := 1 × v.1.1 + 0 × v.1.2 + 0 × v.2 ≥ -6 × scaleDown × blowup.
Definition ineq3 v := 0 × v.1.1 + -1 × v.1.2 + 0 × v.2 ≥ -6 × scaleDown × blowup.
Definition ineq4 v := 0 × v.1.1 + 1 × v.1.2 + 0 × v.2 ≥ -6 × scaleDown × blowup.
Definition ineq5 v := 0 × v.1.1 + 0 × v.1.2 + -1 × v.2 ≥ -6 × scaleDown × blowup.
Definition ineq6 v := 0 × v.1.1 + 0 × v.1.2 + 1 × v.2 ≥ -6 × scaleDown × blowup.

Definition ineq7 v := -1 × v.1.1 + -1 × v.1.2 + -1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq8 v := -1 × v.1.1 + -1 × v.1.2 + 1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq9 v := -1 × v.1.1 + 1 × v.1.2 + -1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq10 v := -1 × v.1.1 + 1 × v.1.2 + 1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq11 v := 1 × v.1.1 + -1 × v.1.2 + -1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq12 v := 1 × v.1.1 + -1 × v.1.2 + 1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq13 v := 1 × v.1.1 + 1 × v.1.2 + -1 × v.2 ≥ -8 × scaleDown × blowup.
Definition ineq14 v := 1 × v.1.1 + 1 × v.1.2 + 1 × v.2 ≥ -8 × scaleDown × blowup.
```

The `mk_decp` and `ineqx_dec` commands can be disregarded. They are used to be able to prove easily that `ttetra` is decidable, which is true since linear inequalities are decidable.

```
Definition ineqs_dec v := [::
  mk_decp (ineq1 v) (ineq1_dec v);
  mk_decp (ineq2 v) (ineq2_dec v);
  mk_decp (ineq3 v) (ineq3_dec v);
  mk_decp (ineq4 v) (ineq4_dec v);
  mk_decp (ineq5 v) (ineq5_dec v);
  mk_decp (ineq6 v) (ineq6_dec v);
  mk_decp (ineq7 v) (ineq7_dec v);
```

```
  mk_decp (ineq8 v) (ineq8_dec v);
  mk_decp (ineq9 v) (ineq9_dec v);
  mk_decp (ineq10 v) (ineq10_dec v);
  mk_decp (ineq11 v) (ineq11_dec v);
  mk_decp (ineq12 v) (ineq12_dec v);
  mk_decp (ineq13 v) (ineq13_dec v);
  mk_decp (ineq14 v) (ineq14_dec v)].
```

We can now use `ineqs_dec` to define `ttetra`

```
Definition ttetra v := all_valid (map getProp (ineqs_dec v)).
```

We will first define `solution_poly` in terms of s_1, s_2, and s_3. This is the following (large!) definition.

```
Definition solution_poly_gens x y z := c0_3_0 × (powerRZ y 3) + c6_0_2 × (powerRZ x
6) × (powerRZ z 2) + c11_1_0 × (powerRZ x 11) × (powerRZ y 1) + c0_0_0 + c3_3_0 ×
(powerRZ x 3) × (powerRZ y 3) + c4_2_0 × (powerRZ x 4) × (powerRZ y 2) + c1_2_2 ×
(powerRZ x 1) × (powerRZ y 2) × (powerRZ z 2) + c2_0_1 × (powerRZ x 2) × (powerRZ
z 1) + c7_2_0 × (powerRZ x 7) × (powerRZ y 2) + c8_1_0 × (powerRZ x 8) × (powerRZ
y 1) + c4_0_0 × (powerRZ x 4) + c1_0_2 × (powerRZ x 1) × (powerRZ z 2) + c4_0_3 ×
(powerRZ x 4) × (powerRZ z 3) + c1_3_1 × (powerRZ x 1) × (powerRZ y 3) × (powerRZ
z 1) + c8_0_0 × (powerRZ x 8) + c0_3_1 × (powerRZ y 3) × (powerRZ z 1) + c2_1_3 ×
(powerRZ x 2) × (powerRZ y 1) × (powerRZ z 3) + c6_3_0 × (powerRZ x 6) × (powerRZ y
3) + c3_1_2 × (powerRZ x 3) × (powerRZ y 1) × (powerRZ z 2) + c1_1_0 × (powerRZ x 1)
× (powerRZ y 1) + c2_0_3 × (powerRZ x 2) × (powerRZ z 3) + c12_0_0 × (powerRZ x 12)
+ c7_0_1 × (powerRZ x 7) × (powerRZ z 1) + c6_1_1 × (powerRZ x 6) × (powerRZ y 1) ×
(powerRZ z 1) + c1_5_0 × (powerRZ x 1) × (powerRZ y 5) + c0_2_1 × (powerRZ y 2) ×
(powerRZ z 1) + c5_4_0 × (powerRZ x 5) × (powerRZ y 4) + c7_1_1 × (powerRZ x 7) ×
(powerRZ y 1) × (powerRZ z 1) + c0_1_3 × (powerRZ y 1) × (powerRZ z 3) + c2_4_0 ×
(powerRZ x 2) × (powerRZ y 4) + c2_2_1 × (powerRZ x 2) × (powerRZ y 2) × (powerRZ
z 1) + c1_0_0 × (powerRZ x 1) + c0_2_2 × (powerRZ y 2) × (powerRZ z 2) + c7_1_0 ×
(powerRZ x 7) × (powerRZ y 1) + c0_2_0 × (powerRZ y 2) + c1_2_1 × (powerRZ x 1) ×
(powerRZ y 2) × (powerRZ z 1) + c3_2_0 × (powerRZ x 3) × (powerRZ y 2) + c4_1_0 ×
(powerRZ x 4) × (powerRZ y 1) + c2_3_1 × (powerRZ x 2) × (powerRZ y 3) × (powerRZ
z 1) + c5_1_0 × (powerRZ x 5) × (powerRZ y 1) + c1_3_2 × (powerRZ x 1) × (powerRZ
y 3) × (powerRZ z 2) + c0_0_3 × (powerRZ z 3) + c0_4_1 × (powerRZ y 4) × (powerRZ
z 1) + c5_3_0 × (powerRZ x 5) × (powerRZ y 3) + c1_0_4 × (powerRZ x 1) × (powerRZ
z 4) + c1_0_3 × (powerRZ x 1) × (powerRZ z 3) + c2_3_0 × (powerRZ x 2) × (powerRZ
y 3) + c7_0_2 × (powerRZ x 7) × (powerRZ z 2) + c13_0_0 × (powerRZ x 13) + c6_2_0 ×
(powerRZ x 6) × (powerRZ y 2) + c1_1_3 × (powerRZ x 1) × (powerRZ y 1) × (powerRZ
z 3) + c0_0_2 × (powerRZ z 2) + c3_0_1 × (powerRZ x 3) × (powerRZ z 1) + c9_2_0 ×
(powerRZ x 9) × (powerRZ y 2) + c10_1_0 × (powerRZ x 10) × (powerRZ y 1) + c4_2_1 ×
(powerRZ x 4) × (powerRZ y 2) × (powerRZ z 1) + c8_1_1 × (powerRZ x 8) × (powerRZ
y 1) × (powerRZ z 1) + c2_1_1 × (powerRZ x 2) × (powerRZ y 1) × (powerRZ z 1) +
c1_4_0 × (powerRZ x 1) × (powerRZ y 4) + c4_3_1 × (powerRZ x 4) × (powerRZ y 3) ×
(powerRZ z 1) + c9_0_0 × (powerRZ x 9) + c2_4_1 × (powerRZ x 2) × (powerRZ y 4) ×
(powerRZ z 1) + c3_2_2 × (powerRZ x 3) × (powerRZ y 2) × (powerRZ z 2) + c3_1_0 ×
(powerRZ x 3) × (powerRZ y 1) + c2_0_0 × (powerRZ x 2) + c0_2_3 × (powerRZ y 2) ×
(powerRZ z 3) + c2_0_2 × (powerRZ x 2) × (powerRZ z 2) + c10_0_1 × (powerRZ x 10) ×
(powerRZ z 1) + c0_1_0 × (powerRZ y 1) + c3_5_0 × (powerRZ x 3) × (powerRZ y 5) +
c0_0_4 × (powerRZ z 4) + c6_0_0 × (powerRZ x 6) + c0_3_2 × (powerRZ y 3) × (powerRZ
z 2) + c3_1_1 × (powerRZ x 3) × (powerRZ y 1) × (powerRZ z 1) + c0_5_0 × (powerRZ
y 5) + c4_0_1 × (powerRZ x 4) × (powerRZ z 1) + c0_1_1 × (powerRZ y 1) × (powerRZ
z 1) + c4_4_0 × (powerRZ x 4) × (powerRZ y 4) + c10_0_0 × (powerRZ x 10) + c6_2_1 ×
(powerRZ x 6) × (powerRZ y 2) × (powerRZ z 1) + c2_1_2 × (powerRZ x 2) × (powerRZ y
1) × (powerRZ z 2) + c9_1_0 × (powerRZ x 9) × (powerRZ y 1) + c1_3_0 × (powerRZ x 1)
× (powerRZ y 3) + c2_2_0 × (powerRZ x 2) × (powerRZ y 2) + c0_6_0 × (powerRZ y 6)
+ c5_2_0 × (powerRZ x 5) × (powerRZ y 2) + c6_1_0 × (powerRZ x 6) × (powerRZ y 1)
+ c3_2_1 × (powerRZ x 3) × (powerRZ y 2) × (powerRZ z 1) + c3_0_2 × (powerRZ x 3)
× (powerRZ z 2) + c0_0_1 × (powerRZ z 1) + c2_2_2 × (powerRZ x 2) × (powerRZ y 2)
× (powerRZ z 2) + c1_0_1 × (powerRZ x 1) × (powerRZ z 1) + c3_0_0 × (powerRZ x 3)
+ c4_3_0 × (powerRZ x 4) × (powerRZ y 3) + c7_3_0 × (powerRZ x 7) × (powerRZ y 3)
+ c8_2_0 × (powerRZ x 8) × (powerRZ y 2) + c5_1_2 × (powerRZ x 5) × (powerRZ y 1)
```

```
× (powerRZ z 2) + c6_0_1 × (powerRZ x 6) × (powerRZ z 1) + c5_0_0 × (powerRZ x 5)
+ c4_1_2 × (powerRZ x 4) × (powerRZ y 1) × (powerRZ z 2) + c4_0_2 × (powerRZ x 4)
× (powerRZ z 2) + c3_4_0 × (powerRZ x 3) × (powerRZ y 4) + c7_0_0 × (powerRZ x 7) +
c1_4_1 × (powerRZ x 1) × (powerRZ y 4) × (powerRZ z 1) + c0_4_0 × (powerRZ y 4) +
c1_1_2 × (powerRZ x 1) × (powerRZ y 1) × (powerRZ z 2) + c3_3_1 × (powerRZ x 3) ×
(powerRZ y 3) × (powerRZ z 1) + c9_0_1 × (powerRZ x 9) × (powerRZ z 1) + c1_1_1 ×
(powerRZ x 1) × (powerRZ y 1) × (powerRZ z 1) + c11_0_0 × (powerRZ x 11) + c5_2_1 ×
(powerRZ x 5) × (powerRZ y 2) × (powerRZ z 1) + c3_0_3 × (powerRZ x 3) × (powerRZ z
3) + c0_1_2 × (powerRZ y 1) × (powerRZ z 2) + c5_1_1 × (powerRZ x 5) × (powerRZ y
1) × (powerRZ z 1) + c1_2_0 × (powerRZ x 1) × (powerRZ y 2) + c2_1_0 × (powerRZ x
2) × (powerRZ y 1) + c8_0_1 × (powerRZ x 8) × (powerRZ z 1) + c0_5_1 × (powerRZ y
5) × (powerRZ z 1) + c1_6_0 × (powerRZ x 1) × (powerRZ y 6) + c2_5_0 × (powerRZ x
2) × (powerRZ y 5) + c5_0_1 × (powerRZ x 5) × (powerRZ z 1) + c4_1_1 × (powerRZ x
4) × (powerRZ y 1) × (powerRZ z 1) + c5_0_2 × (powerRZ x 5) × (powerRZ z 2).
```

We can now define `solution_poly` in terms of the usual coordinates.

```
Definition solution_poly x y z :=
  solution_poly_gens (powerRZ x 2 + powerRZ y 2 + powerRZ z 2)
                     (powerRZ x 4 + powerRZ y 4 + powerRZ z 4)
                     (powerRZ x 6 + powerRZ y 6 + powerRZ z 6).
```

Finally, we define this as a function of $\mathbb{R}^3$ to $\mathbb{R}$, instead of as a function of type $\mathbb{R} \to (\mathbb{R} \to (\mathbb{R} \to \mathbb{R}))$.

```
Definition solution_pair t := solution_poly t.1.1 t.1.2 t.2.
```

We have not shown the definitions of the coefficients of the polynomials. These are abstract, and have no exact definitions. Instead, they are assumed to be in some interval. We have omitted these intervals since they do not contribute meaningful information.

Bibliography

[1] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, jun 2014.

[2] Henry Cohn and Noam Elkies. New upper bounds on sphere packings i. *Annals of Mathematics*, 157(2):689–714, mar 2003.

[3] Henry Cohn and Abhinav Kumar. Universally optimal distribution of points on spheres. *Journal of the American Mathematical Society*, 20(01):99–149, jan 2007.

[4] Henry Cohn, Abhinav Kumar, Stephen D. Miller, Danylo Radchenko, and Maryna Viazovska. The sphere packing problem in dimension 24. *Annals of Mathematics*, 185(3):1017–1033, may 2017.

[5] Maria Dostert, Cristóbal Guzmán, Fernando Mário de Oliveira Filho, and Frank Vallentin. New upper bounds for the density of translative packings of three-dimensional convex bodies with tetrahedral symmetry. *Discrete & Computational Geometry*, 58(2):449–481, mar 2015.

[6] Carl Friedrich Gauss. Recension der „untersuchungen über die eigenschaften der positiven ternären quadratischen formen von ludwig august seeber, dr. der philosophie, ordentl. professor an der universität in freiburg. 1831. 248 s. 4.". *Journal für die Reine und Angewandte Mathematik*, 20: 312–320, 1840.

[7] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A Formal Proof of the Kepler Conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[8] Antonius J. C. Hurkens. A simplification of girard's paradox. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, TLCA '95, pages 266–278, Berlin, Heidelberg, 1995. Springer-Verlag.

[9] Yang Jiao, Frank Stillinger, and Sal Torquato. Optimal packings of superballs. *Physical Review E*, 79(4), apr 2009.

[10] Johannes Kepler. *Six-Cornered Snowflake*. Paul Dry Books, Inc, 1611.

[11] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in coq. *Journal of Automated Reasoning*, 57(3):187–217, oct 2015.

[12] Russell O'Connor. Certified exact transcendental real number computation in coq. *Theorem Proving in Higher Order Logics*, January 2008.

[13] C Tankink. *Documentation and formal mathematics: web technology meets theorem proving*. PhD thesis, Radboud University Nijmegen, 2013.

[14] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.6.0)*, 2019. `https://www.sagemath.org`.

[15] S. Torquato and Y. Jiao. Dense packings of the platonic and archimedean solids. *Nature*, 460 (7257):876–879, aug 2009.

[16] Thomas W. Tucker. Rethinking rigor in calculus: The role of the mean value theorem. *The American Mathematical Monthly*, 104(3):231, mar 1997.

[17] Maryna Viazovska. The sphere packing problem in dimension 8. *Annals of Mathematics*, 185(3): 991–1015, may 2017.

[18] Freek Wiedijk. Comparing mathematical provers. *Mathematical Knowledge Management*, pages 188–202, January 2003.