Bachelor's Thesis
Computing Science & Mathematics

Radboud University Nijmegen

Towards Formalising the Isoperimetric Theorem

Author:
Marten Straatsma
s1041007

First supervisor/assessor:
dr. Freek Wiedijk
`freek@cs.ru.nl`

Second assessor:
dr. Wieb Bosma
`bosma@math.ru.nl`

August 28, 2022

Abstract

The Isoperimetric Theorem, though millennia old, has yet to be formally proven with the use of computer programs to verify correctness. In this research, we lay out the first steps towards formalising this seemingly intuitive theorem in two parts.

First, we work out a proof of the Isoperimetric Theorem by Nikolaos Dergiades in great detail. Second, we develop some definitions and formalise some arithmetic and basic properties in HOL Light to be used in future research.

Preface

Before you lies the thesis ‘Towards Formalising the Isoperimetric Theorem’, the basis of which is a proof of the Isoperimetric Theorem and an interactive theorem prover. It has been written to fulfill the graduation requirements of the Double Bachelor Mathematics and Computer Science Programme at the Radboud University in Nijmegen. I was engaged in researching and writing this thesis from March to August 2022.

This project was the most interesting and challenging undertaking I had ever set out on. Coincidentally, it is also the project with the most hours of labour gone to waste, as I spent the majority of my research working under a false assumption.

I would like to thank my supervisor Freek Wiedijk for his guidance and being patient with me for the past six months. Additionally, I would like to thank my great friends Jip and Muireann for their proofreading and incredibly helpful suggestions. Most of all, I would like to thank my parents, who supported me both morally and physically in times where I struggled to keep my head above water.

I hope you enjoy.

Marten Straatsma

Epse, August 25, 2022

Contents

Chapter 1

Introduction

1.1 The Isoperimetric Problem

1.1.1 Appearance in tales

Perhaps surprisingly, the Isoperimetric Problem is not strictly introduced into literature through mathematical works. Instead, we find that its concept had already been applied in poetry as early as the first century B.C.E. The most famous example is perhaps in the Latin epic poem *Aeneid* by Publius Vergillius Maro, later known as Virgil [19]. In Book I, Queen Dido flees her murderous borther Pygmalion to the shores of North Africa where she founds the city of Carthage. Virgil writes:

> They sailed to this place where today you'll see
> Stone walls going higher and the citadel
> Of Carthage, the new town. They bought the land,
> Called Drumskin from the bargain made, a tract
> They could enclose with one bull's hide.

[26, 500-504]
According to the legend, Dido cut the hide into a long rope and used it to enclose a semicircle along the coast, indirectly implying that this shape has the greatest area. It is because of this tale that the Isoperimetric Problem is sometimes also referred to as *Dido's problem.*

1.1.2 Appearance in ancient literature

Greek mathematician Zenodorus was the first to tackle the Isoperimetric Problem as it is generally understood, sometime after Archimedes in the second century B.C.E. However, his *On Isoperimetric Figures* has been lost to time, and only commentaries on it exist [23, p. 47]. One such commenter was Pappus of Alexandria. In the fourth century C.E., he introduced the fifth book of his *Mathematical Collection* with a preface 'On the Sagacity of Bees'. He wrote:

> Bees, then, know just this fact which is useful to them, that the hexagon is greater than the square and the triangle and will hold more honey for the same expenditure of material in constructing each. But we, claiming a greater share in wisdom than the bees, will investigate a somewhat wider problem, namely that, of all equilateral and equiangular plane figures having an equal perimeter, that which has the greater number of angles is always greater, and the greatest of them all is the circle having its perimeter equal to them.

[25, p. 593]
This investigation commences with the works of Zenodorus, who proved that 'of all rectilineal figures having an equal number of sides and equal perimeter, the greatest is that which is equilateral and equiangular.' [25, p. 395], i.e. a regular polygon. Pappus suggests that the rest of Zenodorus' proof is provided in a later chapter. He, however, fails to deliver on this promise [17, p. 212]. Pappus does provide his own contribution by proving that 'of all circular segments having the same circumference the semicircle is the greatest'. [17, p. 390-391]

1.1.3 Appearance in modern literature

As much as Zenodorus was the most prominent figure to work on the Isoperimetric Problem in the ancient world, Swiss mathematician Jakob Steiner tackled the problem in the modern world. Where ancient mathematicians restricted their proofs to polygons, Steiner approached the problem for any closed curve [23, p. 55]. His proof relied on the equivalence of two statements:

- Of all closed curves in a plane with equal perimeters, the circle bounds the largest area.
- Of all closed curves in a plane with equal areas, the circle has the smallest perimeter.

which is easily shown. This top statement is what is currently most commonly accepted as the Isoperimetric Theorem on the plane, and in this thesis is denoted as

$$\mathcal{L} = \mathcal{L}_{circle}$$
$$\mathcal{A} \leq \mathcal{A}_{circle}$$

where $\mathcal{A}$ and $\mathcal{L}$ are the area and perimeter of any closed curve, and $\mathcal{A}_{circle}$ and $\mathcal{L}_{circle}$ are the area and perimeter of a circle.

Steiner was convinced his proof was the solution to the Isoperimetric Problem. However, as later mathematicians (such as the German Peter Dirichlet) noted, Steiner's proof assumed that a solution to the Isoperimetric Problem exists; that there exists a curve that bounds an area greater than any other curve does, which is not a given [23, p. 59].

Several other geometric problems do not have solutions. A great example of this is the Kakeya problem, which in its simplest form asks: 'What is the least area in the plane required to continuously rotate a needle of unit length and zero thickness around completely?' [24]. We define a Kakeya (needle) set as a set on the plane which contains a unit length segment in each direction [24]. Many's intuition would argue the Kakeya set with smallest area is a circle with half a unit length's radius. However, by moving the needle back and forth we can create these so-called Perron trees (see figure 1.1), and construct Kakeya sets with arbitrarily small area [24]. Asking which Kakeya set has least area is equivalent to asking which strictly positive real number is the smallest, which has no answer. Hence, there is no answer to the Kakeya problem.

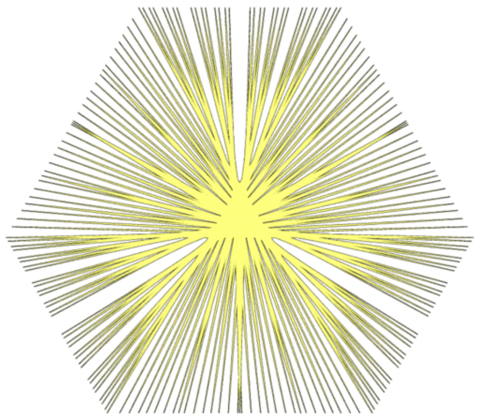

Figure 1.1: A Kakeya needle set constructed using Perron trees [10]

Rather than approaching the Isoperimetric Problem geometrically, some mathematicians attempted to use calculus. Using calculus, the Isoperimetric Problem can be stated as follows: Find an arc with parametric equations $x = x(t)$, $y = y(t)$ for $t \in [t_1, t_2]$ such that $x(t_1) = x(t_2)$ and $y(t_1) = y(t_2)$ (where no further intersections occur) constrained by

$$\mathcal{L} = \int_{t_1}^{t_2} \sqrt{x'^2(t) + y'^2(t)} dt$$

such that

$$\mathcal{A} = \frac{1}{2} \int_{t_1}^{t_2} \big(x(t)y'(t) - x'(t)y(t)\big)\, dt$$

is a maximum [27].

It is this very approach that ultimately lead the brothers Jakob and Johann Bernoulli to developing the calculus of variations in one of their various altercations [8, p. 14].

1.2 Contributions

In July of 1999, Paul and Jack Abad presented their list of ‘The Hundred Greatest Theorems’ [1]. They based their selection on ‘the place the theorem holds in the literature, the quality of the proof, and the unexpectedness of the result’. Though, one should not take this list too seriously.

Freek Wiedijk has taken this list and kept track of most formalisations that exist for each of the theorems [28]. Of all theorems on this list, only two have yet to be formalised: Fermat’s Last Theorem, and the Isoperimetric Theorem. Hence formalising the Isoperimetric Theorem would bring us one step closer towards formalising the entire list. Note however that before the Isoperimetric Theorem can be crossed off the list, a more generalised version has to be formalised, since the Isoperimetric Inequality holds for any surface and volume in any dimension.

In our research we have worked out a proof of the Isoperimetric Theorem by Nikolaos Dergiades in great detail and formalised some arithmetic and basic definitions & lemmas involving polygons.

1.3 Related Work

In this thesis we look into two areas of research: proving the Isoperimetric Theorem and formalising measure theory & geometry.

The Isoperimetric Inequality has already been proven using methods from several mathematical fields. As we have seen in the previous section, it was geometrically ‘proven’ on the plane by Steiner in the nineteenth century [23]. It has since been properly proven on the plane using results from linear algebra and mathematical analysis by Erhard Schmidt [18]. Even more advanced: it has been proven in any dimension using convex geometry and in two dimensions using differential geometry by Penelope Gehring [9]. Proving the Isoperimetric Theorem for polygons on the plane has seemingly gone out of fashion in recent mathematics, though there is this one paper by Nikolaos Dergiades from 2002 which proves this version of the Isoperimetric Theorem using some ingeniously constructed parallelograms [7]. We will discuss this paper in greater detail in chapter 3.

The aforementioned list of 100 theorems includes many formalisations of results from measure theory and geometry. Theorems such as the Area of a Circle, Minkowski's Fundamental Theorem and Lebesgue Measure and Integration have all been formalised in HOL Light, Isabelle and occasionally Coq.

Formalisations in HOL Light are for the most part collected in HOL Light's GitHub. This repository includes libraries with results from measure theory [13] and geometry [15], but also includes formalisations of theorems such as the Jordan Curve Theorem [12], the theorems listed above [16] and results intended for the Flyspeck project [14], which formalised the Kepler conjecture.

Isabelle has a similarly convenient collection of libraries and scientific developments: the Archive of Formal Proofs. Results in this archive include libraries for complex and projective geometry [21][3], but also formalisations of the Hahn and Jordan Decomposition Theorems [6]. Formalisations of the 100 greatest theorems (including the aforementioned ones) are collected by Gerwin Klein [20].

Coq is arguably not as developed in the field of measure theory [22, p. 1], however formalisations do exist. Boldo, Clément, Faissole, Martin and Mayero formalised the Lebesgue Integration of Nonnegative functions in 2021 [2], and Brun, Dufourd and Magoud used Coq in 2012 to prove the correctness of convex hull algorithms with hypermaps [4].

1.4 Overview

The following chapter provides a brief overview of the proof assistant that has been used in our formalisations: the HOL Light theorem prover. Readers who are already familiar with this proof assistant may choose to begin reading at chapter 3, in which we explain the proof we chose to formalise in great detail. Chapter 4 contains the main contributions of this thesis in the form of the formalisations we were able to produce. Finally, we conclude our thesis in chapter 5.

Chapter 2

Preliminaries: HOL Light

Back in 1972, Robin Milner and his assistants designed one of the first proof checking systems at Stanford. He based this system on the 'Logic of Computable Functions', hence the system was called 'Stanford LCF'. Unfortunately, Stanford LCF had two major issues: memory usage and customisability. So when Milner moved to Edinburgh around 1979, he and his assistants C. Wadsworth and Mike Gordon designed a new and improved system called 'Edinburgh LCF'. This version cut down on memory usage by removing parts of the proof once it had moved on, akin to wiping out part of a blackboard to make room for the rest of a proof. It improved on customisability by embedding Edinburgh LCF in a specially designed programming language: ML (a predecessor of OCaml). This version was later improved upon again by Lawrence Paulson in 1987 at Cambridge, hence creating 'Cambridge LCF'.

As Milner's assistant Gordon noted, despite the name 'LCF', nothing in the Edinburgh LCF methodology is tied to the Logic of Computable Functions. He modified Cambridge LCF to support classical Higher Order Logic, thus creating HOL. HOL saw further development at Cambridge, until being consolidated and rationalised in a major release in late 1988, called 'HOL88'. At this point HOL started garnering a community, which lead to the development of multiple versions of HOL.

One such version was HOL Light, developed by John Harrison. This version of HOL was based on an isolated and simplified version of the essential logical core of HOL, written by Konrad Slind. Instead of ML, Harrison developed HOL Light in CAML Light, a lightweight descendant of ML that included some important extra features (such as arrays). Though ultimately the system was ported from CAML Light to Objective CAML.

[11, p. 216-222]

2.1 The Basics

2.1.1 Expressions

Before we can start formalising in HOL, we need to know what language HOL uses to construct mathematical statements. Fortunately, this language is very similar to the mathematical language we are already familiar with. Each symbol has its own encoding in HOL (see figure 2.1), every sequence of digits is a number and any string can be a variable (though some variable names are not parsed correctly). We can combine these inside backticks (‘) to construct mathematical expressions within HOL.

```
# `0 < 1`;;
# `(statement1 ==> statement2)
    ==> (~statement2 ==> ~statement1)`;;
```

Note that these are not simply strings. HOL stores these expressions as abstract syntax trees. Hence, it is not possible to construct expressions that do not make sense.

```
# `3 + (<)`;;
# `(1 < 3) + 8`;;
```

However, nothing refrains us from creating false statements, or
non-statements.

```
# `P <=> ~P`;;
# `1 + 2 + 3`;;
```

HOL gets around this by differentiating between terms and theorems. Terms can be any mathematical expression. Theorems are mathematical statements that have been formally proven.

2.1.2 Types

In HOL's terms, every variable has a type. Most of the time, HOL can determine which type every variable is of from context. Though in cases where it does not, we fortunately can specify the correct type ourselves.

$\top$	$\bot$	$\wedge$	$\vee$	$\neg$	$\Longrightarrow$	$\Longleftrightarrow$	$\forall$	$\exists$	$\exists!$
`T`	`F`	`/\`	`\/`	`~`	`==>`	`<=>`	`!`	`?`	`?!`

$<$	$\leq$	$+$	$-$	$\cdot$	$/$
`<`	`<=`	`+`	`-` (binary) or `--` (unary)	`*`	`/`

Figure 2.1: Translation of mathematical symbols (top) into their HOL Light equivalents (bottom)

Standard HOL has three types of numbers: natural numbers (`num`), integers (`int`) and real numbers (`real`). `&` is used to project a natural number into the integers or reals, and `--` is used to negate a number.

```
# `0:num`;;
# `-- &11:int`;;
# `(sqrt (&2)):real`;;
```

Support for complex numbers does exist, but only through a separate library. Additionally, HOL has types that contain another type, such as lists (`(A)list`) and vectors (`A^N`).

```
# `[1,1,2,3,5]:(num)list =
    CONS 1 (CONS 1 (CONS 2 (CONS 3 (CONS 5, NIL))))`;;
# `x:A^3 = vector [x$1:A; x$2:A; x$3:A]`;;
```

More interestingly though, is that operators have types too.

```
# `=:A->A->bool`;;
# `<=>:bool->bool->bool`;;
# `/:real->real->real`;;
# `&:num->real`;;
```

This is because HOL is a functional language. This means we can define expressions as we would with any functional language.

```
# `((+) 5):num->num`;;
```

More practically, this means that any part of an expression has a type.

```
# `(==>:bool->bool->bool) ((<:num->num->bool) m:num n:num)
    ((<=:num->num->bool) m:num n:num)`;;
# `(m < n):bool ==> (m <= n):bool`;;
# `(m < n ==> m <= n):bool`;;
```

Finally, as we already have seen in the examples above, HOL has the boolean type (`bool`). This type is particularly interesting, because terms of this type are statements, and we can try to prove statements.

2.1.3 Proving Theorems

Proving a theorem in HOL happens either through conversions, or on the goalstack. Conversions (listed amongst others in the Reference Manual) turn a term into a theorem using elementary logic, such as the symmetry of equality.

```
# SYM_CONV `x = y`;;
val it : thm = |- x = y <=> y = x
```

Additionally, we can combine conversions using `THENC` to create even more powerful conversions.

```
# BETA_CONV `(\x. x + 1) 3`;;
val it : thm = |- (\x. x + 1) 3 = 3 + 1

# (BETA_CONV THENC NUM_ADD_CONV) `(\x. x + 1) 3`;;
val it : thm = |- (\x. x + 1) 3 = 4
```

Alternatively, we can prove a theorem using the goalstack. A goal consists of a list of assumptions and a conclusion we want to draw from these assumptions. We can alter goals through the application of functions called tactics. Tactics (listed amongst others in the Reference Manual) vary wildly in their effects, though all of them take a goal and produce a list of goals. Occasionally the tactic will solve the goal, in which case the produced list is empty. Much like conversions, we can combine tactics using `THEN` to create even more powerful tactics.

Proving a theorem happens by creating a goalstack with the theorem as the goal and adding tactics to this goalstack. HOL keeps track of the resulting list of goals after each application of a tactic in what is called the goalstate. Once the goalstate is empty, the sequence of tactics on the goalstack is sufficient to prove the goal and HOL automatically uses the sequence to prove the theorem.

Which tactics are used is completely up to the user, though most formalisations follow these three steps:

- Simplify the goal using `STRIP_TAC`. This tactic removes universal quantifiers and turns antecedents into assumptions. Occasionally, it will also split goals into multiple smaller goals (such as when the desired conclusion contains a conjunction).
- Simplify the goal using `REWRITE_TAC`. This tactic takes an additional input in the form of a list of theorems and attempts to rewrite the goal with those theorems. `SIMP_TAC` works very similarly, though is slightly more powerful.
- Solve the goal using `MESON_TAC` or `ARITH_TAC`. Unlike other tactics, these tactics will either solve the goal or throw a failure. `MESON_TAC`, much like `REWRITE_TAC`, takes a list of theorems and attempts to solve the goal using those theorems and pure first order logic. `ARITH_TAC` attempts to solve the goal using basic arithmetic. HOL comes with multiple versions of this tactic, the most powerful of which we first have to convert into a tactic: `CONV_TAC REAL_FIELD`.

Once proven, we can store the theorem in a variable.

```
# let THEOREM_NAME = top_thm();;
```

However, repeating this process of creating goalstacks for every theoreom we want to store in a variable is not efficient. To this end, we use the function `prove`.

```
# let THEOREM_NAME = prove (tm, tac);;
```

Where `tm` is the term that is proven and `tac` is the tactic that proves it. We can store these blocks of proofs in a text file and at a later point either load a single theorem into HOL by copying and pasting a single block, or load all blocks in a file with the `needs` function.

```
# needs "Multivariate/vectors.ml";;
```

2.1.4 Definitions

Creating a new definition is generally done in either one of two ways, depending on whether the definition is recursive. Non-recursive definitions are generally defined using `new_definition`.

```
# let dot = new_definition
    `(x:real^N) dot (y:real^N) =
     sum(1..dimindex(:N)) (\i. x$i * y$i)`;;
```

Recursive definitions are generally defined using `define`.

```
# let multifactorial = define
    `multifactorial m n =
        if m = 0 then 1
        else if n <= m
             then n
             else n * multifactorial m (n - m)`;;
```

2.2 Examples

In this section we ultimately formally proof that for any two vectors, the cross product is zero if and only if the vectors are parallel or if either vector is zero. Fortunately for us, many definitions and theorems involving vectors have already been formalised and stored in `Multivariate/vectors.ml`.

```
# needs "Multivariate/vectors.ml";;
```

Unfortunately, this file does not contain the definition of the cross product, hence we have to define it ourselves.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_2y_3 - x_3y_2 \\ x_3y_1 - x_1y_3 \\ x_1y_2 - x_2y_1 \end{pmatrix}$$

```
# parse_as_infix ("cross", (20, "right"));;
# let cross = new_definition
    `(x:real^3) cross (y:real^3) =
        vector [x$2 * y$3 - x$3 * y$2;
                x$3 * y$1 - x$1 * y$3;
                x$1 * y$2 - x$2 * y$1] :real^3`;;
```

With this definition, we can try to prove something simple. First, we create a goalstack with our statement.

$$\forall c \in \mathbb{R}, x, y \in \mathbb{R}^3 : (cx) \times y = c(x \times y)$$

```
# g `!c x y. (c % x) cross y = c % (x cross y)`;;
val it : goalstack = 1 subgoal (1 total)

`!c x y. c % x cross y = c % (x cross y)`
```

Second, we get rid of the universal quantifiers.

```
# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

`c % x cross y = c % (x cross y)`
```

Then we write out our definition of the cross product.

```
# e (REWRITE_TAC [cross]);;
val it : goalstack = 1 subgoal (1 total)

`vector [(c % x)$2 * y$3 - (c % x)$3 * y$2;
         (c % x)$3 * y$1 - (c % x)$1 * y$3;
         (c % x)$1 * y$2 - (c % x)$2 * y$1] =
 c % vector [x$2 * y$3 - x$3 * y$2;
             x$3 * y$1 - x$1 * y$3;
             x$1 * y$2 - x$2 * y$1]`
```

Two vectors are equal if their components are equal. Since in HOL it is easier to work with individual components, we do another rewrite.

```
# e (REWRITE_TAC [CART_EQ; LAMBDA_BETA;
                  DIMINDEX_3; FORALL_3]);;
val it : goalstack = 1 subgoal (1 total)

`vector [(c % x)$2 * y$3 - (c % x)$3 * y$2;
         (c % x)$3 * y$1 - (c % x)$1 * y$3;
         (c % x)$1 * y$2 - (c % x)$2 * y$1]$1 =
 (c % vector [x$2 * y$3 - x$3 * y$2;
              x$3 * y$1 - x$1 * y$3;
              x$1 * y$2 - x$2 * y$1])$1
 /\
 vector [(c % x)$2 * y$3 - (c % x)$3 * y$2;
         (c % x)$3 * y$1 - (c % x)$1 * y$3;
         (c % x)$1 * y$2 - (c % x)$2 * y$1]$2 =
 (c % vector [x$2 * y$3 - x$3 * y$2;
              x$3 * y$1 - x$1 * y$3;
              x$1 * y$2 - x$2 * y$1])$2
 /\
 vector [(c % x)$2 * y$3 - (c % x)$3 * y$2;
         (c % x)$3 * y$1 - (c % x)$1 * y$3;
         (c % x)$1 * y$2 - (c % x)$2 * y$1]$3 =
 (c % vector [x$2 * y$3 - x$3 * y$2;
              x$3 * y$1 - x$1 * y$3;
              x$1 * y$2 - x$2 * y$1])$3`
```

We can make this a bit more readable by using a few equivalences.

$$\forall x \in \mathbb{R}^3 : \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}_i = x_i$$

$$\forall c \in \mathbb{R}, x \in \mathbb{R}^3 : (cx)_i = cx_i$$

```
# e (REWRITE_TAC [VECTOR_MUL_COMPONENT; VECTOR_3]);;
val it : goalstack = 1 subgoal (1 total)

`(c * x$2) * y$3 - (c * x$3) * y$2 =
 c * (x$2 * y$3 - x$3 * y$2)
 /\
 (c * x$3) * y$1 - (c * x$1) * y$3 =
 c * (x$3 * y$1 - x$1 * y$3)
 /\
 (c * x$1) * y$2 - (c * x$2) * y$1 =
 c * (x$1 * y$2 - x$2 * y$1)`
```

We are left with some arithmetic.

```
# e REAL_ARITH_TAC;;
val it : goalstack = No subgoals
```

That solves the goal.

This sequence of tactics is very similar for many of the simple statements we might want to prove that involve the cross product. In order to spare the effort in the future, we can gather these tactics in a completely new tactic.

```
# let VEC3_TAC =
    SIMP_TAC [CART_EQ; LAMBDA_BETA; FORALL_3; SUM_3;
              DIMINDEX_3; VECTOR_3;
              VECTOR_ADD_COMPONENT; VECTOR_SUB_COMPONENT;
              VECTOR_NEG_COMPONENT; VECTOR_MUL_COMPONENT;
              vector_add; vec; dot; cross; orthogonal; basis;
              ARITH]
    THEN CONV_TAC REAL_RING;;

# let VEC3_RULE tm = prove (tm, VEC3_TAC);;
```

Now we can proof simple statements in a single line.

$$\forall x \in \mathbb{R}^3 : x \times x = \vec{0}$$

$$\forall x \in \mathbb{R}^3 : x \times \vec{0} = \vec{0}$$

```
# let CROSS_REFL = VEC3_RULE `!x. x cross x = vec 0`;;
# let CROSS_RZERO = VEC3_RULE `!x. x cross vec 0 = vec 0`;;
# let CROSS_MUL_LDISTRIB =
    VEC3_RULE `!c x y. (c % x) cross y = c % (x cross y)`;;
```

Now for the more interesting statement we mentioned earlier: For any two vectors, the cross product is zero if and only if the vectors are parallel or if either vector is zero.

$$\forall x, y \in \mathbb{R}^3 : (\exists c \in \mathbb{R} : x = cy) \vee y = \vec{0} \iff x \times y = \vec{0}$$

```
# g `!x y. (?c. x = c % y) \/ y = vec 0 <=>
             x cross y = vec 0`;;
val it : goalstack = 1 subgoal (1 total)

`!x y. (?c. x = c % y) \/ y = vec 0 <=> x cross y = vec 0`
```

Again we first eliminate all of the universal quantifiers.

```
# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

`(?c. x = c % y) \/ y = vec 0 <=> x cross y = vec 0`
```

Next we separate the equality into two implications.

```
# e EQ_TAC;;
val it : goalstack = 2 subgoals (2 total)

`x cross y = vec 0 ==> (?c. x = c % y) \/ y = vec 0`

`(?c. x = c % y) \/ y = vec 0 ==> x cross y = vec 0`
```

This creates a goalstate with two goals. We will continue working on the top goal (which is printed below all other goals). We can simplify this goal even further.

```
# e STRIP_TAC;;
val it : goalstack = 2 subgoals (3 total)

  0 [`y = vec 0`]

`x cross y = vec 0`

  0 [`x = c % y`]

`x cross y = vec 0`
```

This creates a goalstate with three goals of which only the two that were just created are printed. Additionally, we see that these two goals have an assumption printed between square brackets. We can use this assumption to rewrite the goal once more.

```
# e (POP_ASSUM SUBST1_TAC);;
val it : goalstack = 1 subgoal (3 total)

`c % y cross y = vec 0`
```

This goal we can either solve using the theorems `CROSS_MUL_LDISTRIB` and `CROSS_REFL` we just proved, or by using the tactic we just created.

```
# e (REWRITE_TAC [CROSS_MUL_LDISTRIB; CROSS_REFL;
                  VECTOR_MUL_RZERO]);;
val it : goalstack = 1 subgoal (2 total)

  0 [`y = vec 0`]

`x cross y = vec 0`
```

HOL immediatedly prints the next highest goal, which is solved similarly to the previous goal.

```
# e (POP_ASSUM SUBST1_TAC THEN VEC3_TAC);;
val it : goalstack = 1 subgoal (1 total)

`x cross y = vec 0 ==> (?c. x = c % y) \/ y = vec 0`
```

For the remaining goal, it is easier to prove the contrapositive.

```
# e (MATCH_MP_TAC (TAUT `(~B ==> ~A) ==> A ==> B`));;
val it : goalstack = 1 subgoal (1 total)

`~((?c. x = c % y) \/ y = vec 0) ==> ~(x cross y = vec 0)`
```

Like last time, we write out the definition of a cross product, what it means for vectors to be equal, and simplify slightly.

```
# e (REWRITE_TAC [cross; CART_EQ; LAMBDA_BETA; FORALL_3;
                  DIMINDEX_3; VECTOR_3;
                  VEC_COMPONENT; VECTOR_MUL_COMPONENT]);;
val it : goalstack = 1 subgoal (1 total)

`~((?c. x$1 = c * y$1 /\ x$2 = c * y$2 /\ x$3 = c * y$3)
   \/
   y$1 = &0 /\ y$2 = &0 /\ y$3 = &0)
 ==> ~(x$2 * y$3 - x$3 * y$2 = &0
       /\
       x$3 * y$1 - x$1 * y$3 = &0
       /\
       x$1 * y$2 - x$2 * y$1 = &0)`
```

Negating the existential quantifier and using De Morgan's laws simplifies this further.

```
# e (REWRITE_TAC [NOT_EXISTS_THM; DE_MORGAN_THM]);;
val it : goalstack = 1 subgoal (1 total)

`(!c. ~(x$1 = c * y$1) \/ ~(x$2 = c * y$2) \/ ~(x$3 = c * y$3))
 /\
 (~(y$1 = &0) \/ ~(y$2 = &0) \/ ~(y$3 = &0))
 ==> ~(x$2 * y$3 - x$3 * y$2 = &0)
     \/
     ~(x$3 * y$1 - x$1 * y$3 = &0)
     \/
     ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

Now the antecedent is finally of a useable form. Hence we can add it to the list of assumptions, splitting the goal into multiple in the process.

```
# e STRIP_TAC;;
val it : goalstack = 3 subgoals (3 total)

  0 [`!c. ~(x$1 = c * y$1)
          \/
          ~(x$2 = c * y$2)
          \/
          ~(x$3 = c * y$3)`]
  1 [`~(y$3 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

```
  0 [`!c. ~(x$1 = c * y$1)
          \/
          ~(x$2 = c * y$2)
          \/
          ~(x$3 = c * y$3)`]
  1 [`~(y$2 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`

  0 [`!c. ~(x$1 = c * y$1)
          \/
          ~(x$2 = c * y$2)
          \/
          ~(x$3 = c * y$3)`]
  1 [`~(y$1 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

We want to get rid of the universal quantifier in our second assumption (assumption 0) by specifying $c = \frac{x_1}{y_1}$.

```
# e (POP_ASSUM (fun asm1 -> POP_ASSUM (fun asm0 ->
        ASSUME_TAC (SPEC `(x:real^3)$1 / (y:real^3)$1` asm0)
        THEN ASSUME_TAC asm1)));;
val it : goalstack = 1 subgoal (3 total)

  0 [`~(x$1 = x$1 / y$1 * y$1)
      \/
      ~(x$2 = x$1 / y$1 * y$2)
      \/
      ~(x$3 = x$1 / y$1 * y$3)`]
  1 [`~(y$1 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

After which we are left with some arithmetic.

```
# e (REPEAT (POP_ASSUM MP_TAC) THEN CONV_TAC REAL_FIELD);;
val it : goalstack = 1 subgoal (2 total)

  0 [`!c. ~(x$1 = c * y$1)
          \/
          ~(x$2 = c * y$2)
          \/
          ~(x$3 = c * y$3)`]
  1 [`~(y$2 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

The next two goals are solved similarly to the previous goal. Though we can use some functional programming to construct a tactic that works for all three goals. Instead of specifying `i` ourselves, we deduce it from the first assumption (assumption 1).

```
# e (POP_ASSUM (fun asm1 -> POP_ASSUM (fun asm0 ->
        let i = (rand o rand o rator o rand o concl) asm1 in
        ASSUME_TAC (SPEC (vsubst [i,`i:num`; i, `j:num`] `(x:
    real^3)$i / (y:real^3)$j`) asm0)
        THEN ASSUME_TAC asm1)));;
val it : goalstack = 1 subgoal (2 total)

  0 [`~(x$1 = x$2 / y$2 * y$1)
      \/
      ~(x$2 = x$2 / y$2 * y$2)
      \/
      ~(x$3 = x$2 / y$2 * y$3)`]
  1 [`~(y$2 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

And again we are left with some arithmetic.

```
# e (REPEAT (POP_ASSUM MP_TAC) THEN CONV_TAC REAL_FIELD);;
val it : goalstack = 1 subgoal (1 total)

  0 [`!c. ~(x$1 = c * y$1)
          \/
          ~(x$2 = c * y$2)
          \/
          ~(x$3 = c * y$3)`]
  1 [`~(y$3 = &0)`]

`~(x$2 * y$3 - x$3 * y$2 = &0)
 \/
 ~(x$3 * y$1 - x$1 * y$3 = &0)
 \/
 ~(x$1 * y$2 - x$2 * y$1 = &0)`
```

And the last goal is more of the same.

```
# e (POP_ASSUM (fun asm1 -> POP_ASSUM (fun asm0 ->
        let i = (rand o rand o rator o rand o concl) asm1 in
        MP_TAC (SPEC (vsubst [i,`i:num`; i, `j:num`]
            `(x:real^3)$i / (y:real^3)$j`) asm0)
        THEN MP_TAC asm1))
    THEN CONV_TAC REAL_FIELD);;
val it : goalstack = No subgoals
```

Chapter 3

The Proof

3.1 The Theorem

The proof we chose to formalise was originally developed by Nikolaos Dergiades in 2002 [7]. In his paper, Dergiades proves the following statement:

Theorem (Isoperimetric Inequality)**.** *In every polygon with perimeter $\mathcal{L}$ and area $\mathcal{A}$ we have $4\pi\mathcal{A} \leq \mathcal{L}^2$.*

This inequality is equivalent to the original Isoperimetric Inequality by the following logic:

$$\mathcal{L} = \mathcal{L}_{circle} = 2\pi r \implies r = \frac{\mathcal{L}}{2\pi}$$
$$\mathcal{A} \leq \mathcal{A}_{circle} = \pi r^2 = \pi \left(\frac{\mathcal{L}}{2\pi}\right)^2 = \frac{\mathcal{L}^2}{4\pi}$$

Note that this theorem only applies to polygons and not all closed curves. Dergiades addresses this in his closing statement and states that we can extend this theorem to all closed curves by means of limits on the number of segments in the polygon. He does not however, provide any reasoning with this statement, and we do not go into the validity of this statement in this thesis.

3.2 Definitions

In the proof, Dergiades does not bother to define any of the definitions he uses. However, since some of these definitions are not standard and we have to be precise in our formalisation, it has merit to provide them explicitly.

Definition (Polygonal Chain)**.** A *Polygonal Chain* P is a curve consisting of line segments connecting consecutive points $P_i \in \mathbb{R}^2$ specified in a sequence $(P_i)_{i=1}^{n}$.

Definition (Polygon)**.** A *polygon* P is a closed polygonal chain. For convenience, we add a point $P_0 = P_n$ to the corresponding sequence. This guarantees that the polygonal chain is closed and simplifies many of the formulae.

Definition (Simple Polygon)**.** A polygon P is *simple* if all of the following hold.

- For every $0 < i < n$ the segments $P_{i-1}P_i$ and P_iP_{i+1} only intersect in point P_i.
- The segments $P_{n-1}P_n$ and P_0P_1 only intersect in point $P_n = P_0$.
- For every $0 < i, j < n$ such that $i < j$ and not $i = 1 \land j = n - 1$, the segments $P_{i-1}P_i$ and P_jP_{j+1} do not intersect.

Note that if a polygon is simple, that we can differentiate between the inside (or interior) of the polygon and the outside (or exterior) of the polygon.

Definition (Convex Polygon)**.** A polygon is *convex* if it is simple and for every two points S and T in the interior of the polygon, the segment ST is completely contained in the interior of the polygon (i.e. does not intersect with the segment P_iP_{i+1} for any $0 \leq i < n$).

Definition (Polygon Length)**.** The *length* or *perimeter* of a polygon P is the length of the closed curve. It is given by:

$$\mathcal{L} = \sum_{i=0}^{n-1} \operatorname{dist}(P_i, P_{i+1})$$

Definition (Polygon Area)**.** The *area* of a simple polygon is the area of the interior of the curve. It is given by the shoelace formula, sometimes also known as surveyor's formula:

$$\mathcal{A} = \frac{1}{2} \left| \sum_{i=0}^{n-1} (P_i)_x (P_{i+1})_y - (P_i)_y (P_{i+1})_x \right|$$

Note that this definition only applies to simple polygons. For self-intersecting polygons, there exist multiple definitions.

1. Using the shoelace formula may result in some sections having negative area. This can result in polygons such as in figure 3.1 having area $\mathcal{A} = 0$.

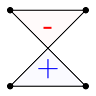

Figure 3.1: Shoelace formula would yield area $\mathcal{A} = 0$

2. Alternatively, one could separate the polygon into multiple simple polygons (as shown in figure 3.2) and sum their areas. However, this can result in some areas of the polygon being counted multiple times, as shown in figure 3.3.

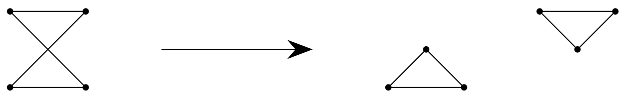

Figure 3.2: Separating into simple polygons

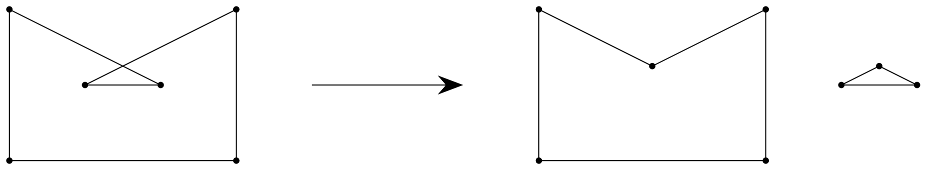

Figure 3.3: Separating into simple polygons with overlapping areas

3. Finally, we could apply the previous method to the outline of the polygon (as shown in figure 3.4). This prevents counting certain areas multiple times.

Fortunately, the validity of the theorem does not depend on our choice of method.

Proof. Suppose P is a (possibly self-intersecting) polygon with length $\mathcal{L}$ and areas A', A'' and A''' determined using the three aforementioned methods. By construction, we have $A', A''' \leq A''$. By the second method, P can be separated into simple polygons $\left(P^{(1)}, P^{(2)}, \dots\right)$ with areas $\left(\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \dots\right)$ and lengths $\left(\mathcal{L}^{(1)}, \mathcal{L}^{(2)}, \dots\right)$ respectively. Should the Isoperimetric Theorem hold for simple polygons, then we would have $4\pi\mathcal{A}^{(i)} \leq \left(\mathcal{L}^{(i)}\right)^2$. This would in turn imply:

$$4\pi\mathcal{A}', 4\pi\mathcal{A}''' \leq 4\pi\mathcal{A}'' = \sum_i 4\pi\mathcal{A}^{(i)} \leq \sum_i \left(\mathcal{L}^{(i)}\right)^2 \leq \left(\sum_i \mathcal{L}^{(i)}\right)^2 = \mathcal{L}^2$$

□

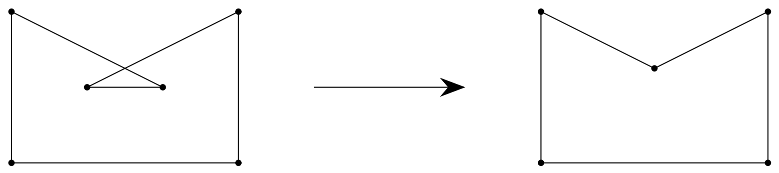
Figure 3.4: Outline of a polygon

3.3 Non-Convex Polygons

The proof by Dergiades starts off with 'It is sufficient to prove the inequality for a convex polygon'. Dergiades does not provide any reasoning with this claim, however it does have merit to go over this claim in more detail. The idea is to use some properties of the convex hull of a simple polygon.

Definition (Convex Hull of a Simple Polygon)**.** The polygon P' specified by $\left(P_{i_j}\right)_{j=0}^{k}$ is a *convex hull* of a simple polygon P specified by $(P_i)_{i=0}^{n}$ if:

- $\left(P_{i_j}\right)_{j=1}^{k}$ is a subsequence of $(P_i)_{i=1}^{n}$,
- $P_{i_0} = P_{i_k}$ and
- for every $0 \leq j < k$, the interior of P is contained on one side of the line through P_{i_j} and $P_{i_{j+1}}$.

An example of such a convex hull is displayed in figure 3.5.

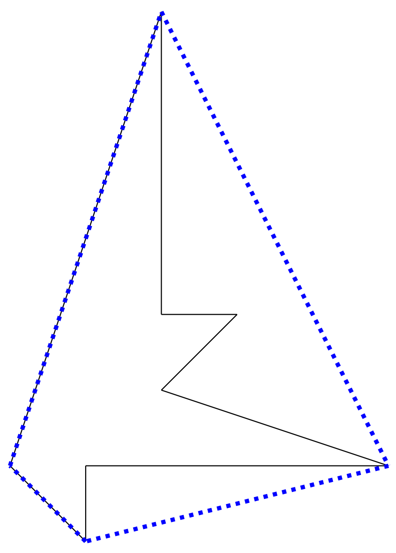
Figure 3.5: Convex Hull

There exist many algorithms that determine the convex hull of a polygon. One such algorithm is Jarvis' march (also known as package/gift wrapping)

[5, p. 1037]. This algorithm roughly proceeds as follows:

Let P be a simple polygon specified by $(P_i)_{i=0}^n$, we will determine a convex hull P' specified by $(P_{i_j})_{j=0}^k$. Without loss of generality, let P_1 be the leftmost point. This point has to be part of the convex hull, hence $i_1 = 1$. Every next point is determined through

$$i_{j+1} = \min\{i_j < i' \leq n \mid \text{the line through } P_{i_j} \text{ and } P_{i'} \text{ does not separate the interior of } P\}$$

Ultimately, at i_k such a segment no longer exists and we add $P_{i_0} = P_{i_k}$ to close the curve.

Since this algorithm works for any simple polygon, we can conclude that there exists a convex hull for every simple polygon. It not clear however, that a convex hull is indeed convex. In the following lemma we prove that this is indeed the case.

Lemma. *A convex hull P' specified by $\left(P_{i_j}\right)_{j=0}^k$ of a simple polygon P specified by $(P_i)_{i=0}^n$ is convex.*

Proof. Suppose it is not convex. There exist two point S and T in the interior of P' such that the segment ST is not contained in the interior of P'. This implies that ST intersects at least two segments, say $P_{i_j}P_{i_{j+1}}$ and $P_{i_{j'}}P_{i_{j'+1}}$, with the former closest to S and the latter closest to T. By the definition of a convex hull, the interior of P has to be contained on the side of the line through P_{i_j} and $P_{i_{j+1}}$ that includes S, and on the side of the line through $P_{i_{j'}}$ and $P_{i_{j'+1}}$ that contains T. This limits the interior of P to an area that cannot have all four points P_{i_j}, $P_{i_{j+1}}$, $P_{i_{j'}}$ and $P_{i_{j'+1}}$ on its border (as shown in figure 3.6), which is impossible. Therefore a convex hull is convex. □

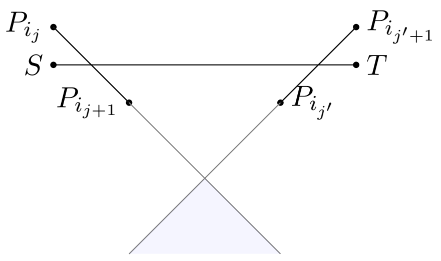

Figure 3.6: Impossible Interior

With the convex hull well-defined, we can prove the following two lemmas.

Lemma. *For every convex hull P' with perimeter $\mathcal{L}'$ of a simple polygon P with perimeter $\mathcal{L}$, we have $\mathcal{L}' \leq \mathcal{L}$.*

Proof. In P', the length of the curve connecting points P_{i_j} and $P_{i_{j+1}}$ is simply $\mathrm{dist}(P_{i_j}, P_{i_{j+1}})$. In P, this depends on whether P_0 lies on the curve.

$$\begin{cases} \sum\limits_{i=i_j}^{i_{j+1}-1} \mathrm{dist}(P_i, P_{i+1}) & j \neq 0 \\ \sum\limits_{i=i_k}^{n-1} \mathrm{dist}(P_i, P_{i+1}) + \sum\limits_{i=0}^{i_1-1} \mathrm{dist}(P_i, P_{i+1}) & j = 0 \end{cases}$$

In either case, by the generalised triangle inequality, the length in P' is always less than or equal to the length in P.

$$\mathrm{dist}(P_{i_j}, P_{i_{j+1}}) \leq \begin{cases} \sum\limits_{i=i_j}^{i_{j+1}-1} \mathrm{dist}(P_i, P_{i+1}) & j \neq 0 \\ \sum\limits_{i=i_k}^{n-1} \mathrm{dist}(P_i, P_{i+1}) + \sum\limits_{i=0}^{i_1-1} \mathrm{dist}(P_i, P_{i+1}) & j = 0 \end{cases}$$

This yields the following inequality:

$$\begin{aligned} \mathcal{L}' &= \sum_{j=0}^{k-1} \mathrm{dist}(P_{i_j}, P_{i_{j+1}}) \\ &= \mathrm{dist}(P_{i_0}, P_{i_1}) + \sum_{j=1}^{k-1} \mathrm{dist}(P_{i_j}, P_{i_{j+1}}) \\ &\leq \sum_{i=i_k}^{n-1} \mathrm{dist}(P_i, P_{i+1}) + \sum_{i=0}^{i_1-1} \mathrm{dist}(P_i, P_{i+1}) + \sum_{j=1}^{k-1} \sum_{i=i_j}^{i_{j+1}-1} \mathrm{dist}(P_i, P_{i+1}) \\ &= \sum_{i=0}^{n-1} \mathrm{dist}(P_i, P_{i+1}) \\ &= \mathcal{L} \end{aligned}$$

□

Lemma. *For every convex hull P' with area $\mathcal{A}'$ of a simple polygon P with area $\mathcal{A}$, we have $A \leq A'$.*

Proof. Since no segment of P' divides the interior of P, the interior of P has to be contain within the interior of P'. This implies that the area that makes up the interior of P also (partly) makes up the interior of P', meaning $\mathcal{A}'$ cannot be smaller than $\mathcal{A}$. □

We combine these lemmas to prove the claim.

Proof. We have already shown it is sufficient to prove the Isoperimetric Inequality for simple polygons in the previous section. Let $\mathcal{A}$ and $\mathcal{L}$ be the area and perimeter of a simple polygon, and let $\mathcal{A}'$ and $\mathcal{L}'$ be the area and perimeter of its convex hull. Suppose the Isoperimetric Inequality holds for convex polygons. Then this would imply the following inequality:

$$4\pi\mathcal{A} \leq 4\pi\mathcal{A}' \leq (\mathcal{L}')^2 \leq \mathcal{L}^2$$

□

3.4 The Proof

We return to the proof by Dergiades. We start off with a convex polygon P specified by $(P_i)_{i=0}^{n}$, with perimeter $\mathcal{L}$ and area $\mathcal{A}$. For the sake of clarity, every step of the proof will be applied to the example polygon shown in figure 3.7.

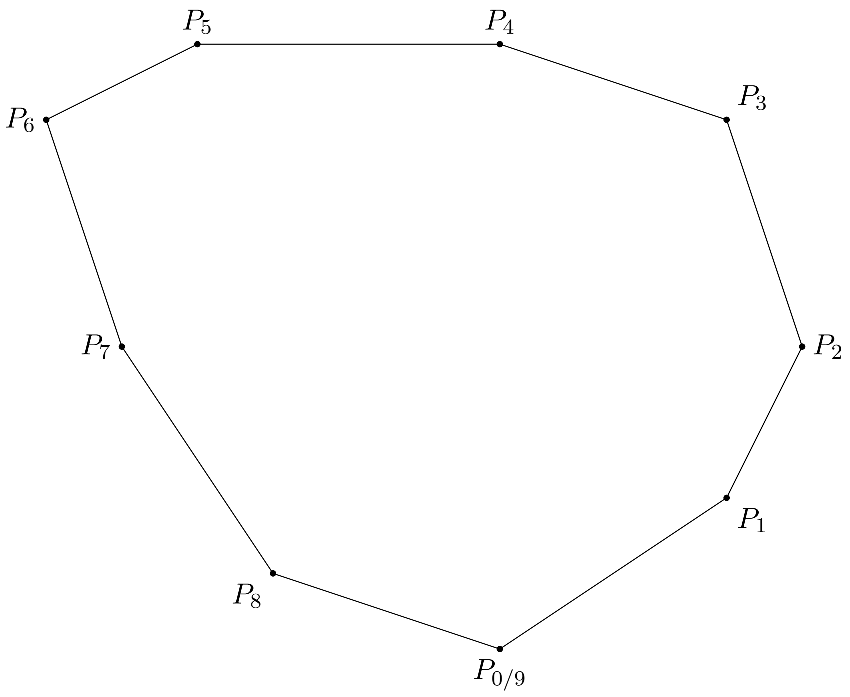

Figure 3.7: Convex Starting Polygon

Proof. First, we determine an m such that

$$\sum_{i=0}^{m-1} \text{dist}(P_i, P_{i+1}) \leq \frac{\mathcal{L}}{2}$$

$$\sum_{i=m+1}^{n-1} \text{dist}(P_i, P_{i+1}) \leq \frac{\mathcal{L}}{2}$$

On segment $P_m P_{m+1}$ we determine a point P_O such that

$$\sum_{i=0}^{m-1} \text{dist}(P_i, P_{i+1}) + \text{dist}(P_m, P_O) = \frac{\mathcal{L}}{2}$$

$$\text{dist}(P_O, P_{m+1}) + \sum_{i=m+1}^{n-1} \text{dist}(P_i, P_{i+1}) = \frac{\mathcal{L}}{2}$$

Then segment P_0P_O divides the polygon into two polygons. Let O be the mid-point of P_0P_O and let the polygon specified by $(P_0, \ldots, P_m, P_O)$ have area $\mathcal{A}_1$, as shown in figure 3.8. Without loss of generality, we assume that

$$\frac{\mathcal{A}}{2} \leq \mathcal{A}_1 \tag{3.1}$$

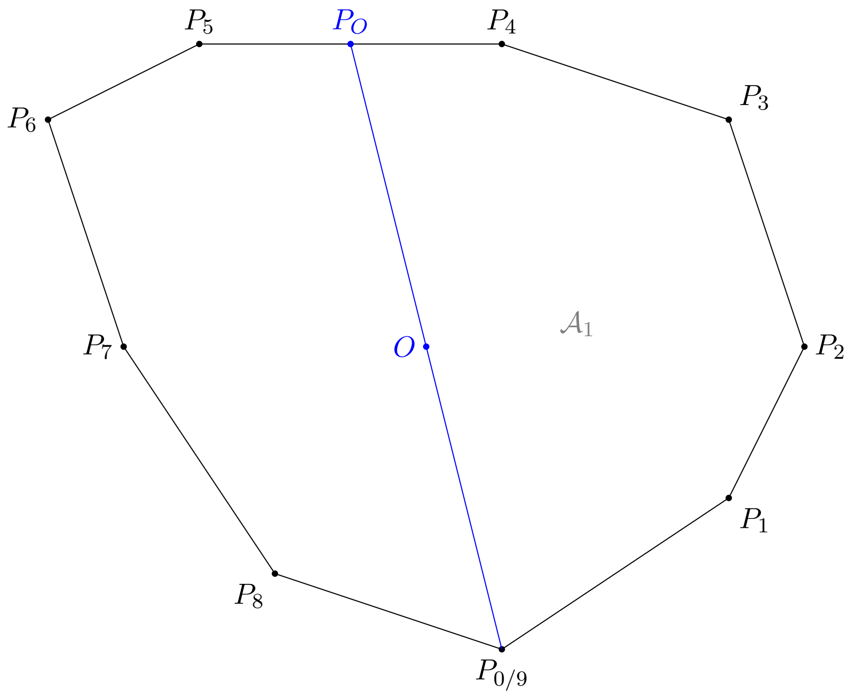

Figure 3.8: Divided Polygon

Let $i_{max} = \arg\max\limits_{0 \le i \le m} \operatorname{dist}(O, P_i)$ (note that $\operatorname{dist}(O, P_0) = \operatorname{dist}(O, P_O)$ by construction) and let $\mathcal{R} = \operatorname{dist}(O, P_{i_{max}})$. Draw circle $(O, \mathcal{R})$ and from the points P_0 and P_O draw perpendiculars to $OP_{i_{max}}$ to meet the circle at $\hat{P}_0$ and $\hat{P}_O$ respectively, as shown in figure 3.9. Because of symmetry, the polygonal chain $(\hat{P}_O, P_O, P_0, \hat{P}_0)$ divides the circle into two parts with equal area:

$$\mathcal{S} = \frac{1}{2}\pi\mathcal{R}^2 \tag{3.2}$$

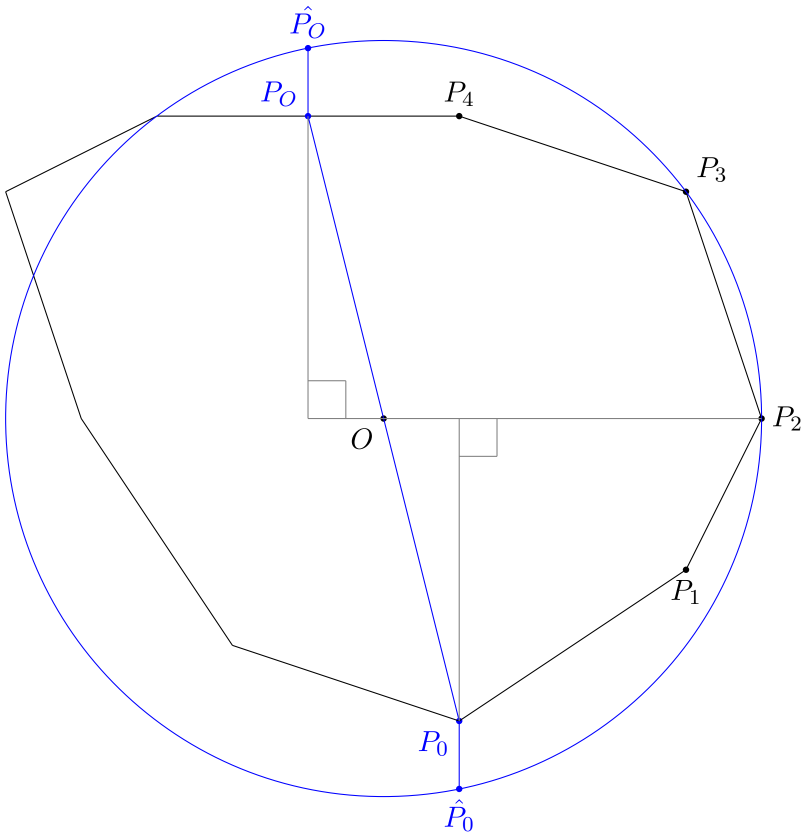

Figure 3.9: Polygon with Circle

For every segment P_iP_{i+1} with $i < m$, let P_i' and P_i'' be points such that $(P_i, P_i', P_i'', P_{i+1})$ forms a parallelogram with sides P_iP_i' and $P_{i+1}P_i''$ perpendicular to $OP_{i_{max}}$ and with $P_i'P_i''$ touching the circle. Additionally, let $a_i = \text{dist}(P_i, P_{i+1}) = \text{dist}(P_i', P_i'')$ be the base of the paralellogram, h_i be the altitude of triangle OP_iP_{i+1} and d_i be the height of the parallelogram, as shown in figure 3.10. Note that

$$h_i + d_i = \mathcal{R} \tag{3.3}$$

Similarly to the other segments, let P_m' and P_m'' be points such that (P_m, P_m', P_m'', P_O) forms a parallelogram with sides P_mP_m' and P_OP_m'' perpendicular to $OP_{i_{max}}$ and with $P_m'P_m''$ touching the circle. Additionally, let $a_m = \text{dist}(P_m, P_O) = \text{dist}(P_m', P_m'')$, h_m be the height of triangle OP_mP_O and d_m be the height of the parallelogram.

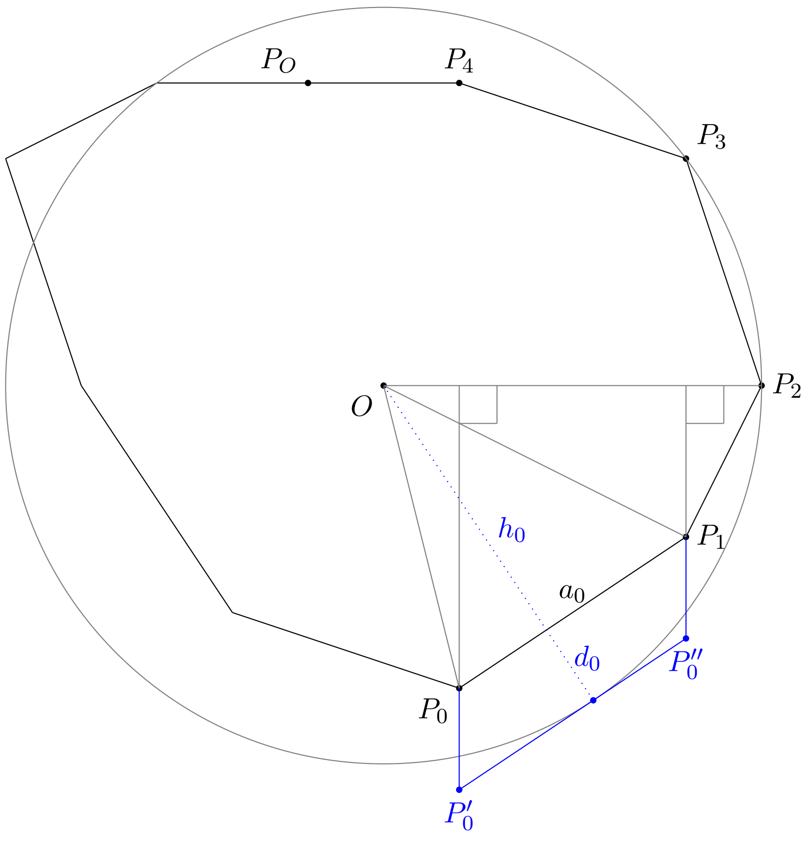

Figure 3.10: Polygon with one Parallelogram

Ultimately, we end up with m parallelograms. Let $\mathcal{A}_2$ denote the sum of all their areas. Note that these parallelograms, together with the polygon specified by $(P_0, \ldots, P_m, P_O)$, cover half of the circle, as shown in 3.11. This implies that

$$\mathcal{S} \leq \mathcal{A}_1 + \mathcal{A}_2 \tag{3.4}$$

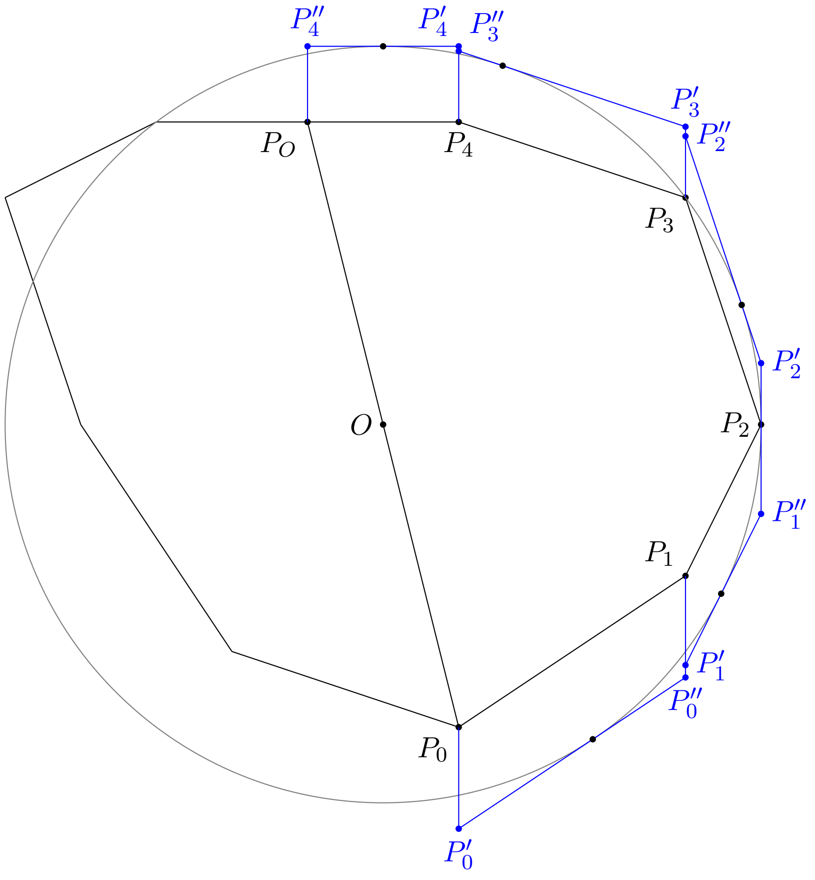

Figure 3.11: Polygon with all Parallelograms

At this stage we have enough information to deduce the Isoperimetric Theorem. First note that

$$\mathcal{A}_1 = \frac{1}{2}\sum_{i=0}^{m} a_i h_i$$

$$\mathcal{A}_2 = \sum_{i=0}^{m} a_i d_i \overset{3.3}{=} \sum_{i=0}^{m} a_i(\mathcal{R} - h_i) = \mathcal{R}\sum_{i=0}^{m} a_i - \sum_{i=0}^{m} a_i h_i = \mathcal{R}\cdot\frac{\mathcal{L}}{2} - 2\mathcal{A}_1$$

$$\frac{1}{2}\pi\mathcal{R}^2 \overset{3.2}{=} \mathcal{S} \overset{3.4}{\leq} \mathcal{A}_1 + \mathcal{A}_2 = \mathcal{R}\cdot\frac{\mathcal{L}}{2} - \mathcal{A}_1$$

$$\pi\mathcal{R}^2 - \mathcal{L}\mathcal{R} + 2\mathcal{A}_1 \leq 0$$

$$\pi\left(\mathcal{R} - \frac{\mathcal{L}}{2\pi}\right)^2 - \left(\frac{\mathcal{L}^2}{4\pi} - 2\mathcal{A}_1\right) \leq 0$$

$\pi\left(\mathcal{R} - \frac{\mathcal{L}}{2\pi}\right)^2$ cannot be negative, hence neither can $\frac{\mathcal{L}^2}{4\pi} - 2\mathcal{A}_1$.

$$0 \leq \frac{\mathcal{L}^2}{4\pi} - 2\mathcal{A}_1$$

$$4\pi\mathcal{A} \overset{3.1}{\leq} 4\pi\cdot 2\mathcal{A}_1 \leq \mathcal{L}^2$$

$\square$

Chapter 4

Formalisation

4.1 Definitions & Basic Theorems

HOL Light comes pre-loaded with a large amount of standard definitions and theorems. Additionally, we use the definitions and theorems defined in `vectors.ml` and `transc.ml`, as those definitions include vectors and π. The remaining definitions and theorems we have to formalise ourselves, which is described in this section.

4.1.1 Cross Product

Traditionally, the cross product is only defined on two three-dimensional vectors. However, in this formalisation we are exclusively working with two-dimensional vectors and only interested in the third-dimensional component of the resulting vector of the cross product. Hence, we define a slightly altered version of the cross product.

$$\times' : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$$

$$x \times' y = (\iota(x) \times \iota(y))_3 = x_1 \cdot y_2 - x_2 \cdot y_1$$

Which we formalise in HOL as follows:

```
# parse_as_infix("cross",(20,"right"));;

# let cross = new_definition
    `(x:real^2) cross (y:real^2) = x$1 * y$2 - x$2 * y$1`;;
```

This definition of the cross product satisfies many of the properties the traditional cross product satisfies, such as its antisymmetry and distributivity over addition.

$$\forall x, y \in \mathbb{R}^2 : x \times' y = -y \times' x$$

$$\forall x, y, z \in \mathbb{R}^2 : x \times' (y + z) = x \times' y + x \times' z$$

```
# let CROSS_SYM = prove (
    `!x y. x cross y = --(y cross x)`,
    REWRITE_TAC[cross]
    THEN ARITH_TAC
);;

# let CROSS_ADD_LDISTRIB = prove (
    `!x y z. x cross (y + z) = (x cross y) + (x cross z)`,
    REWRITE_TAC[cross; VECTOR_ADD_COMPONENT]
    THEN ARITH_TAC
);;
```

We have listed many more of these properties in appendix B. Note that all of these properties are proven with a similar tactic. We write out the definition of our altered cross product, do some more rewriting and solve the goal using arithmetic.

4.1.2 Area Triangle

We use this altered cross product to define the area of a triangle. Since for any two vectors the area of the parallelogram they span is equal to the length of their cross product, the area of the triangle they span is equal to half this length. Using the altered cross product can also yield negative values, hence we get the signed area of a triangle.

```
# let triangle_area = new_definition
    `triangle_area x y z = ((y - x) cross (z - x)) / &2`;;
```

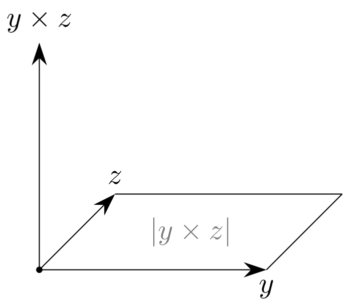

Figure 4.1: Correspondence between the cross product (perpendicular to the plane) and the area of the parallelogram

This definition has a few basic properties we would like to formalise. Firstly, we would like to prove that the area of the triangle does not change as we choose a different base point (as shown in figure 4.2).

Secondly, we would like to prove that the area flips sign as the rotation of the points on the triangle flips (as shown in figure 4.3).

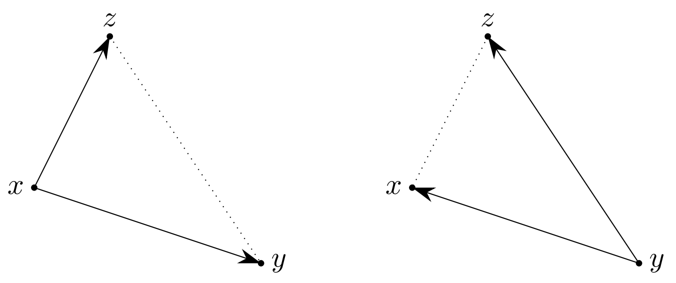

Figure 4.2: Geometric meaning behind `triangle_area x y z` and `triangle_area y z x`

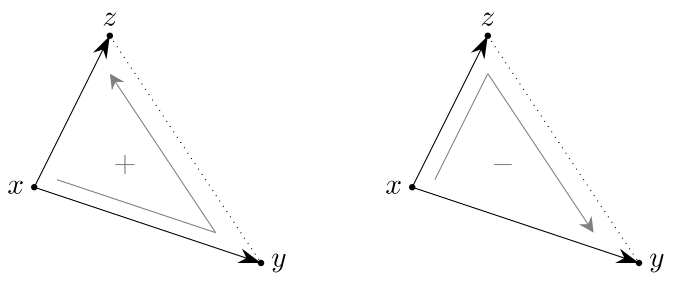

Figure 4.3: Geometric meaning behind `triangle_area x y z` and `triangle_area x z y`

These two properties are sufficient to relate all orderings of three points. Finally, we would like to prove a property of two adjacent triangles. Namely, if two triangles are adjacent to form a quadrilateral, both diagonals of the quadrilateral divide the quadrilateral into triangles such that the sums of their areas are equal (as shown in figure 4.4).

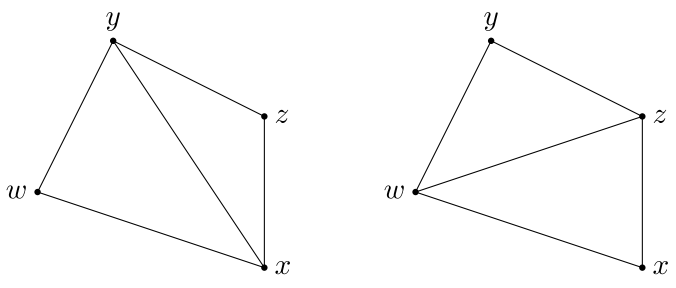

Figure 4.4: Dividing the quadrilateral two ways

Fortunately, these properties are proven similarly to how we solved the properties of the altered cross product.

```
# let TRIANGLE_AREA_ROTATE = prove (
    `!x y z. triangle_area x y z = triangle_area z x y`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;

# let TRIANGLE_AREA_REFLECT = prove (
    `!x y z. triangle_area x y z = --triangle_area x z y`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;

# let TRIANGLE_AREA_ADD = prove (
    `!w x y z. triangle_area w x y + triangle_area z y x =
   triangle_area x z w + triangle_area y w z`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;
```

A glossary of the code in this section can be found in appendix C.

4.1.3 Polygon Area

First we define the area of a polygonal chain, by choosing a point O (not necessarily the origin), drawing triangles between this point and the segments of the polygonal chain and summing their areas (as shown in figure 4.5). For the sake of easier formalisation, this is defined recursively with polygonal chains specified by lists of two-dimensional points. Of course, if the polygonal chain does not contain at least a single segment, the area of the polygonal chain is 0.

```
# let polychain_area = define
    `polychain_area _ _' [] = &0
     /\
     polychain_area O x (CONS y l) =
        triangle_area O x y + polychain_area O y l`;;
```

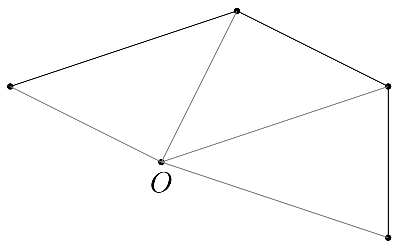

Figure 4.5: Polygonal chain area division

Now we define the area of a polygon by ensuring that the polygonal chain is closed, which we do by adding the last point to the front of the chain (equivalent to adding a point $P_0 = P_n$). Again, if the polygon does not contain any points, we define its area to be 0.

```
# let polygon_area = new_definition
    `polygon_area O l = if l = []
                        then &0
                        else polychain_area O (LAST l) l`;;
```

This definition raises the question of whether the area of a polygon depends on the choice of point O. Fortunately, we can prove that this is not the case.

```
# g `!O O' l. polygon_area O l = polygon_area O' l`;;
val it : goalstack = 1 subgoal (1 total)

`!O O' l. polygon_area O l = polygon_area O' l`
```

As is customary, we first get rid of the universal quantifiers. However this time, list l needs to remain in order to apply list induction.

```
# e (GEN_TAC THEN GEN_TAC);;
val it : goalstack = 1 subgoal (1 total)

`!l. polygon_area O l = polygon_area O' l`

# e LIST_INDUCT_TAC;;
val it : goalstack = 2 subgoals (2 total)

  0 [`polygon_area O t = polygon_area O' t`]

`polygon_area O (CONS h t) = polygon_area O' (CONS h t)`

`polygon_area O [] = polygon_area O' []`
```

Application of list induction yields two subgoals: the base case where l is the empty list, and the induction step where we add a point h to the list t under the assumption that the claim holds for t. The base case holds by the definition of a polygon's area.

```
# e (REWRITE_TAC [polygon_area]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`polygon_area O t = polygon_area O' t`]

`polygon_area O (CONS h t) = polygon_area O' (CONS h t)`
```

For ease of rewriting, we add the assumption as an antecedent to the goal.

```
# e (POP_ASSUM MP_TAC);;
val it : goalstack = 1 subgoal (1 total)

`polygon_area O t = polygon_area O' t
 ==> polygon_area O (CONS h t) = polygon_area O' (CONS h t)`
```

In order to prove the induction step, we once again need to differentiate between an empty list t and a nonempty list.

```
# e (ASM_CASES_TAC `t:(real^2)list = []`);;
val it : goalstack = 2 subgoals (2 total)

  0 [`~(t = [])`]

`polygon_area O t = polygon_area O' t
 ==> polygon_area O (CONS h t) = polygon_area O' (CONS h t)`

  0 [`t = []`]

`polygon_area O t = polygon_area O' t
 ==> polygon_area O (CONS h t) = polygon_area O' (CONS h t)`
```

The case with the empty list is once again easily solved by writing out definitions.

```
# e (ASM_REWRITE_TAC [polygon_area; NOT_CONS_NIL; LAST;
                      polychain_area; triangle_area;
                      CROSS_REFL]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`~(t = [])`]

`polygon_area O t = polygon_area O' t
 ==> polygon_area O (CONS h t) = polygon_area O' (CONS h t)`
```

The remaining case can unfortunately not be solved by simply writing out the definitions. The idea is to use the lemma we proved in the previous section on triangles $O, h,$ `LAST` t and $O, h,$ `HD` t to get triangles $O,$ `HD` $t,$ `TAIL` t and $h,$ `HD` $t,$ `TAIL` t, as shown in figure 4.6. The former triangle combines with the rest of the polygonal chain to form the induction hypothesis, and the latter is independent of O.

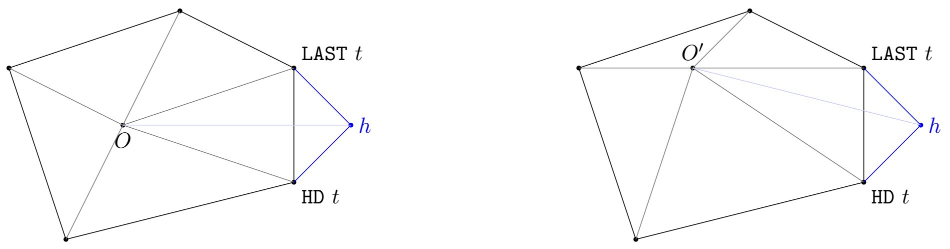

Figure 4.6: Idea behind proof

The first step of applying this idea is writing out the definitions of the area of a polygon and a polygonal chain, and simplifying slightly.

```
# e (ASM_REWRITE_TAC [polygon_area; NOT_CONS_NIL; LAST;
                      polychain_area]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`~(t = [])`]

`polychain_area O (LAST t) t = polychain_area O' (LAST t) t
 ==> triangle_area O (LAST t) h + polychain_area O h t =
     triangle_area O' (LAST t) h + polychain_area O' h t`
```

Since t is nonempty, we can split it into its head and tail elements.

```
# e (POP_ASSUM (fun asm0 -> ONCE_REWRITE_TAC
    [(GEN_ALL o MK_COMB)
     (REFL `polychain_area O h`, MATCH_MP CONS_HD_TL asm0)]));;
val it : goalstack = 1 subgoal (1 total)

`polychain_area O (LAST t) (CONS (HD t) (TL t)) =
 polychain_area O' (LAST t) (CONS (HD t) (TL t))
 ==> triangle_area O (LAST t) h +
     polychain_area O h (CONS (HD t) (TL t)) =
     triangle_area O' (LAST t) h +
     polychain_area O' h (CONS (HD t) (TL t))`
```

We can write out the definition of a polychain again. After which the antecedent is finally of a usable form, hence we move it to the assumptions.

```
# e (REWRITE_TAC [polychain_area] THEN STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

  0 [`triangle_area O (LAST t) (HD t) +
      polychain_area O (HD t) (TL t) =
      triangle_area O' (LAST t) (HD t) +
      polychain_area O' (HD t) (TL t)`]

`triangle_area O (LAST t) h +
 triangle_area O h (HD t) +
 polychain_area O (HD t) (TL t) =
 triangle_area O' (LAST t) h +
 triangle_area O' h (HD t) +
 polychain_area O' (HD t) (TL t)`
```

Now we would like to apply the lemma. However, getting the points of the triangles in the right order is a hassle. Hence, we use MESON to solve it for us.

```
# e (SUBGOAL_THEN `!O. triangle_area O (LAST t) h +
                       triangle_area O h (HD t) =
                       triangle_area h (HD t) (LAST t) +
                       triangle_area O (LAST t) (HD t)`
                      (fun th ->
                        REWRITE_TAC [REAL_ADD_ASSOC; th]
                        THEN REWRITE_TAC [GSYM REAL_ADD_ASSOC])
     THENL [
        MESON_TAC [TRIANGLE_AREA_ROTATE; TRIANGLE_AREA_ADD];
        ALL_TAC
     ]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`triangle_area O (LAST t) (HD t) +
      polychain_area O (HD t) (TL t) =
      triangle_area O' (LAST t) (HD t) +
      polychain_area O' (HD t) (TL t)`]

`triangle_area h (HD t) (LAST t) +
 triangle_area O (LAST t) (HD t) +
 polychain_area O (HD t) (TL t) =
 triangle_area h (HD t) (LAST t) +
 triangle_area O' (LAST t) (HD t) +
 polychain_area O' (HD t) (TL t)`
```

The remaining goal holds by the induction hypothesis.

```
# e (ASM_REWRITE_TAC []);;
val it : goalstack = No subgoals
```

Another interesting property to consider is what happens to the area of a polygon if we concatenate two polygonal chains. We prove that the resulting area is equal to the sum of both areas, plus the area of the quadrilateral that forms in between.

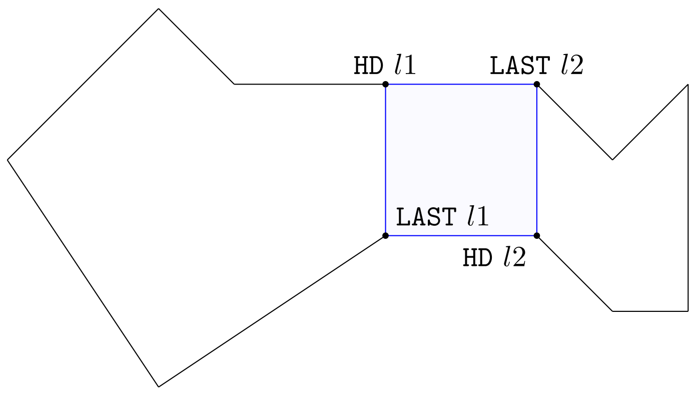

Figure 4.7: Adding two polygons

```
# g `!0 l1 l2. polygon_area 0 (APPEND l1 l2)
               = polygon_area 0 l1 + polygon_area 0 l2
               + if l1 = [] \/ l2 = []
                 then &0
                 else polygon_area 0 [HD l1; LAST l1;
                                      HD l2; LAST l2]`;;
val it : goalstack = 1 subgoal (1 total)

`!0 l1 l2.
     polygon_area 0 (APPEND l1 l2) =
     polygon_area 0 l1 +
     polygon_area 0 l2 +
     (if l1 = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD l1; LAST l1; HD l2; LAST l2])`
```

We first remove the unnecessary universal quantifier and apply list induction.

```
# e (GEN_TAC THEN LIST_INDUCT_TAC);;
val it : goalstack = 2 subgoals (2 total)

  0 [`!l2. polygon_area 0 (APPEND t l2) =
          polygon_area 0 t +
          polygon_area 0 l2 +
          (if t = [] \/ l2 = []
           then &0
           else polygon_area 0 [HD t; LAST t;
                                HD l2; LAST l2])`]

`!l2. polygon_area 0 (APPEND (CONS h t) l2) =
      polygon_area 0 (CONS h t) +
      polygon_area 0 l2 +
      (if CONS h t = [] \/ l2 = []
       then &0
       else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                            HD l2; LAST l2])`

`!l2. polygon_area 0 (APPEND [] l2) =
      polygon_area 0 [] +
      polygon_area 0 l2 +
      (if [] = [] \/ l2 = []
       then &0
       else polygon_area 0 [HD []; LAST []; HD l2; LAST l2])`
```

Again, the base case boils down to writing out definitions.

```
# e (REWRITE_TAC [APPEND; polygon_area; REAL_ADD_RID;
                  REAL_ADD_LID]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`!l2. polygon_area 0 (APPEND t l2) =
          polygon_area 0 t +
          polygon_area 0 l2 +
          (if t = [] \/ l2 = []
           then &0
           else polygon_area 0 [HD t; LAST t;
                                HD l2; LAST l2])`]

`!l2. polygon_area 0 (APPEND (CONS h t) l2) =
      polygon_area 0 (CONS h t) +
      polygon_area 0 l2 +
      (if CONS h t = [] \/ l2 = []
       then &0
       else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                            HD l2; LAST l2])`
```

The universal quantifiers are no longer necessary, and again we add the assumption as the antecedent for the sake of easier formalisation.

```
# e (GEN_TAC THEN POP_ASSUM
    (fun asm0 -> MP_TAC (SPEC_ALL asm0)));;
val it : goalstack = 1 subgoal (1 total)

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
```

We differentiate between the case that $l2$ is empty and the case that $l2$ is nonempty.

```
# e (ASM_CASES_TAC `l2:(real^2)list = []`);;
val it : goalstack = 2 subgoals (2 total)

  0 [`~(l2 = [])`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
  0 [`l2 = []`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
```

The case that *l*2 is empty is easily solved by writing out definitions.

```
# e (ASM_REWRITE_TAC [APPEND_NIL; polygon_area; REAL_ADD_RID])
    ;;
val it : goalstack = 1 subgoal (1 total)

  0 [`~(l2 = [])`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
```

For the last time, we have to differentiate between an empty list t and a nonempty one.

```
# e (ASM_CASES_TAC `t:(real^2)list = []`);;
val it : goalstack = 2 subgoals (2 total)

  0 [`~(l2 = [])`]
  1 [`~(t = [])`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`

  0 [`~(l2 = [])`]
  1 [`t = []`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
```

In order to solve the following goal we need to use a few properties. Firstly, we use that if two points of a triangle coincide, it has area 0. Secondly, the area of a triangle flips in sign if the rotation of the triangle flips. And finally, since *l*2 is nonempty, we can split it into a first element and a rest. Apart from these properties we write out the definitions of the area of a polygon and of a polygonal chain, and simplify it.

```
# e (FIRST_ASSUM (fun asm0 -> ASM_REWRITE_TAC [
        APPEND; NOT_CONS_NIL; HD; LAST;
        polygon_area; polychain_area;
        (GEN_ALL o MK_COMB)
            (REFL `polychain_area O h`,
                  MATCH_MP CONS_HD_TL asm0);
        REWRITE_RULE [CROSS_REFL; real_div; REAL_MUL_LZERO]
            (ISPECL [`h:real^2`; `h:real^2`; `O:real^2`]
                    triangle_area);
        ISPECL [`O:real^2`; `(HD l2):real^2`;
                `(LAST l2):real^2`] TRIANGLE_AREA_REFLECT]));;
val it : goalstack = 1 subgoal (2 total)

  0 [`~(l2 = [])`]
  1 [`t = []`]

`triangle_area O (LAST l2) (HD l2) +
polychain_area O (HD l2) (TL l2) =
 &0 +
 (triangle_area O (LAST l2) (HD l2) +
 polychain_area O (HD l2) (TL l2)) +
 &0
 ==> triangle_area O (LAST l2) h +
     triangle_area O h (HD l2) +
     polychain_area O (HD l2) (TL l2) =
     (&0 + &0) +
     (triangle_area O (LAST l2) (HD l2) +
     polychain_area O (HD l2) (TL l2)) +
     triangle_area O (LAST l2) h +
     -- &0 +
     triangle_area O h (HD l2) +
     --triangle_area O (LAST l2) (HD l2) +
     &0`
```

We end up with some arithmetic.

```
# e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

  0 [`~(l2 = [])`]
  1 [`~(t = [])`]

`polygon_area 0 (APPEND t l2) =
 polygon_area 0 t +
 polygon_area 0 l2 +
 (if t = [] \/ l2 = []
  then &0
  else polygon_area 0 [HD t; LAST t; HD l2; LAST l2])
 ==> polygon_area 0 (APPEND (CONS h t) l2) =
     polygon_area 0 (CONS h t) +
     polygon_area 0 l2 +
     (if CONS h t = [] \/ l2 = []
      then &0
      else polygon_area 0 [HD (CONS h t); LAST (CONS h t);
                           HD l2; LAST l2])`
```

The final goal uses many of the properties we used to prove the previous goal. Though this time we also use that t is nonempty to split it into a first element and a rest.

```
# e (POP_ASSUM (fun asm1 -> POP_ASSUM (fun asm0 -> REWRITE_TAC[
    APPEND; APPEND  _EQ_NIL; NOT_CONS_NIL;
    HD; LAST; LAST_APPEND; HD_APPEND; TL;
    polygon_area; polychain_area; asm0; asm1;
    (GEN_ALL o MK_COMB) (REFL `polychain_area 0 h`,
                           MATCH_MP CONS_HD_TL asm0);
    (GEN_ALL o MK_COMB) (REFL `polychain_area 0 h`,
                           MATCH_MP CONS_HD_TL asm1);
    (GEN_ALL o MK_COMB)
        (REFL `polychain_area 0 h`, MATCH_MP CONS_HD_TL
        (REWRITE_RULE [DE_MORGAN_THM; asm0]
        (((MATCH_MP MONO_NOT) o fst o EQ_IMP_RULE)
        (ISPECL [`t:(real^2)list`; `l2:(real^2)list`]
                APPEND_EQ_NIL))));
    REWRITE_RULE [CROSS_REFL; real_div; REAL_MUL_LZERO]
        (ISPECL [`h:real^2`; `h:real^2`; `0:real^2`]
                triangle_area);
    ISPECL [`0:real^2`; `(HD l2):real^2`; `(LAST l2):real^2`]
        TRIANGLE_AREA_REFLECT])));;
val it : goalstack = 1 subgoal (1 total)
```

```
`triangle_area 0 (LAST l2) (HD t) +
 polychain_area 0 (HD t) (TL (APPEND t l2)) =
 (triangle_area 0 (LAST t) (HD t) +
  polychain_area 0 (HD t) (TL t)) +
 (triangle_area 0 (LAST l2) (HD l2) +
  polychain_area 0 (HD l2) (TL l2)) +
 triangle_area 0 (LAST l2) (HD t) +
 --triangle_area 0 (LAST t) (HD t) +
 triangle_area 0 (LAST t) (HD l2) +
 --triangle_area 0 (LAST l2) (HD l2) +
 &0
 ==> triangle_area 0 (LAST l2) h +
     triangle_area 0 h (HD t) +
     polychain_area 0 (HD t) (TL (APPEND t l2)) =
     (triangle_area 0 (LAST t) h +
      triangle_area 0 h (HD t) +
      polychain_area 0 (HD t) (TL t)) +
     (triangle_area 0 (LAST l2) (HD l2) +
      polychain_area 0 (HD l2) (TL l2)) +
     triangle_area 0 (LAST l2) h +
     --triangle_area 0 (LAST t) h +
     triangle_area 0 (LAST t) (HD l2) +
     --triangle_area 0 (LAST l2) (HD l2) +
     &0`
```

Again we end up with some arithmetic, and promptly solve the goal.

```
# e ARITH_TAC;;
val it : goalstack = No subgoals
```

A glossary of the code in this section can be found in appendix C.

4.2 Arithmetic

Formalising complicated arithmetic in HOL is a comparatively high-effort endeavour. Technically, one could rewrite equations using elementary theorems, such as the associativity of addition or the symmetry of multiplication. However, in practice this quickly becomes infeasible. In particular when a theorem can be applied to multiple parts of an equation, as in HOL there is no convenient way to specify which parts should be rewritten. Fortunately, there are tools that automate this process, such as `ARITH_RULE` and `REAL_FIELD`. These conversions are capable of solving most arithmetic. `ARITH_RULE` is capable of solving equations with addition, subtraction, and to some extend multiplication. `REAL_FIELD` extends on this by solving equations with more complicated multiplication and division. Both conversions are capable of also solving inequalities, however they become considerably less powerful.

4.2.1 Tactics

In order to streamline the rewriting process, we have developed a new conversions and a set of two tactics. The idea was to create a tactic that

- replaces the goal with a term that is specified by the user
- proves this is a sound replacement using `REAL_FIELD`
- can use both the assumptions and a user-specified list of theorems in this proof

We achieve the first part by using `MATCH_MP_TAC`. `MATCH_MP_TAC` takes an implication $A \implies B$ and replaces the goal B with the goal A. The logic behind this is that if we can prove A, the implication would prove the goal B, hence it suffices to prove A. This brings us to the second part. We prove that the implication $A \implies B$ holds by using `REAL_FIELD`. Finally, we want `REAL_FIELD` to also use a list of theorems to prove the implication. We do this by first creating the implication

$$Theorem_1 \wedge \ldots \wedge Theorem_n \implies (A \implies B)$$

then proving this implication using `REAL_FIELD` and finally removing the antecedent using the modus ponens rule: $A \wedge (A \implies B) \implies B$. All in all this yields the following conversion and tactic.

```
# let (REAL_ARITH_REWRITE_CONV: term -> thm list -> conv) =
    fun A ths B ->
        if ths = []
        then REAL_FIELD (mk_imp (A, B))
        else MP (REAL_FIELD (mk_imp
            (list_mk_conj(map(snd o strip_forall o concl) ths),
            mk_imp (A, B))))
            (end_itlist CONJ (map SPEC_ALL ths));;

# let (REAL_ARITH_REWRITE_TAC: term -> thm list -> tactic) =
    fun A ths ((assums, B) as goal) ->
        MATCH_MP_TAC (REAL_ARITH_REWRITE_CONV A ths B) goal;;
```

For the most part the conversion is a literal translation of the logic above, except for `snd o strip_forall`. This is because of a peculiarity with `REAL_FIELD`. It starts misbehaving if an antecedent has a universal quantifier. Mapping `snd o strip_forall` over the list of assumptions removes all universal quantifiers, thus preventing the issue. We can then create a tactic that also uses the assumptions by composing this tactic with the function `ASM`.

```
# let ASM_REAL_ARITH_REWRITE_TAC =
    ASM o REAL_ARITH_REWRITE_TAC;;
```

4.2.2 Formalisation

We want to prove that, given the (in)equalities that we derived from following the proof by Dergiades, the isoperimetric inequality holds.

$$\frac{\mathcal{A}}{2} \overset{3.1}{\leq} \mathcal{A}_1 \wedge \mathcal{S} \overset{3.2}{=} \frac{1}{2}\pi\mathcal{R}^2 \wedge \left(\forall_{0\leq i\leq m} h_i + d_i \overset{3.3}{=} \mathcal{R}\right) \wedge \mathcal{S} \overset{3.4}{\leq} \mathcal{A}_1 + \mathcal{A}_2$$

$$\wedge \mathcal{A}_1 = \frac{1}{2}\sum_{i=0}^{m} a_i h_i \wedge \mathcal{A}_2 = \sum_{i=0}^{m} a_i d_i \wedge \frac{\mathcal{L}}{2} = \sum_{i=0}^{m} a_i$$

$$\implies 4\pi\mathcal{A} \leq \mathcal{L}^2$$

```
# g `!A A_1 A_2 L R S a d h m.
    A / &2 <= A_1
    /\ S = &1 / &2 * pi * R pow 2
    /\ (!i. 0 <= i /\ i <= m ==> h i + d i = R)
    /\ S <= A_1 + A_2
    /\ A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)
    /\ A_2 = sum (0..m) (\i. a i * d i)
    /\ L / &2 = sum (0..m) a
    ==> &4 * pi * A <= L pow 2`;;
val it : goalstack = 1 subgoal (1 total)

`!A A_1 A_2 L R S a d h m.
    A / &2 <= A_1
    /\ S = &1 / &2 * pi * R pow 2
    /\ (!i. 0 <= i /\ i <= m ==> h i + d i = R)
    /\ S <= A_1 + A_2
    /\ A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)
    /\ A_2 = sum (0..m) (\i. a i * d i)
    /\ L / &2 = sum (0..m) a
    ==> &4 * pi * A <= L pow 2`
```

As with most proofs, we first remove all universal quantifiers and add the antecedent as assumptions.

```
# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`&4 * pi * A <= L pow 2`
```

We want to rewrite this goal as

$$0 \leq \frac{\mathcal{L}^2}{4\pi} - 2\mathcal{A}_1$$

for which we need the assumption:

$$\frac{\mathcal{A}}{2} \leq \mathcal{A}_1$$

and the theorems:

$$0 < z \implies \left(\frac{x}{z} \leq \frac{y}{z} \iff x \leq y\right)$$

$$y \neq 0 \implies \frac{y \cdot x}{y} = x$$

$$0 < \pi$$

```
# e (ASM_REAL_ARITH_REWRITE_TAC
    `&0 <= L pow 2 / (&4 * pi) - &2 * A_1`
        [SPECL [`&4 * pi * A`; `L pow 2`; `&4 * pi`]
               REAL_LE_DIV2_EQ;
         SPECL [`A:real`; `&4 * pi`]
               REAL_DIV_LMUL;
         PI_POS]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`&0 <= L pow 2 / (&4 * pi) - &2 * A_1`
```

Next, we want to rewrite this goal as

$$\pi\left(\mathcal{R} - \frac{\mathcal{L}}{2\pi}\right)^2 - \left(\frac{\mathcal{L}^2}{4\pi} - 2\mathcal{A}_1\right) \leq 0$$

This comes down to proving that

$$0 \leq \pi\left(\mathcal{R} - \frac{\mathcal{L}}{2\pi}\right)^2$$

for which we need the following theorems:

$$0 \leq x^2$$

$$0 \leq x \wedge y \leq z \implies x \cdot y \leq x \cdot z$$

$$0 < \pi$$

```
# e (REAL_ARITH_REWRITE_TAC
    `pi * (R - L / (&2 * pi)) pow 2 -
     (L pow 2 / (&4 * pi) - &2 * A_1) <= &0`
        [SPEC `R - L / (&2 * pi)` REAL_LE_POW_2;
         SPECL [`pi`; `&0`; `(R - L / (&2 * pi)) pow 2`]
               REAL_LE_LMUL;
         PI_POS]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]
`pi * (R - L / (&2 * pi)) pow 2 -
 (L pow 2 / (&4 * pi) - &2 * A_1) <= &0`
```

At this point it becomes apparent that REAL_FIELD becomes less powerful when converting inequalities. We want to use that

$$\pi\left(\mathcal{R}-\frac{\mathcal{L}}{2\pi}\right)^2-\left(\frac{\mathcal{L}^2}{4\pi}-2\mathcal{A}_1\right)=\pi\mathcal{R}^2-\mathcal{L}\mathcal{R}+2\mathcal{A}_1$$

which REAL_FIELD can solve without any issue. However, the tactic REAL_ARITH_REWRITE_TAC attempts to prove

$$0<\pi \implies \pi\mathcal{R}^2-\mathcal{L}\mathcal{R}+2\mathcal{A}_1\le 0 \implies \pi\left(\mathcal{R}-\frac{\mathcal{L}}{2\pi}\right)^2-\left(\frac{\mathcal{L}^2}{4\pi}-2\mathcal{A}_1\right)\le 0$$

using REAL_FIELD, which it cannot. Hence, we resort to rewriting the goal using the equality it *can* solve.

```
# e (REWRITE_TAC [MP (REAL_FIELD `&0 < pi ==>
        pi * (R - L / (&2 * pi)) pow 2 -
        (L pow 2 / (&4 * pi) - &2 * A_1) =
        pi * R pow 2 - R * L + &2 * A_1`)
        PI_POS]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`pi * R pow 2 - R * L + &2 * A_1 <= &0`
```

We can rearrange these terms to get an inequality that resembles one of our assumptions.

```
# e (REAL_ARITH_REWRITE_TAC
    `&1 / &2 * pi * R pow 2 <= R * L / &2 - A_1` []);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`&1 / &2 * pi * R pow 2 <= R * L / &2 - A_1`
```

By the assumption

$$S = \frac{1}{2}\pi\mathcal{R}^2$$

this goal is equivalent to

$$S \leq \mathcal{R} \cdot \frac{\mathcal{L}}{2} - \mathcal{A}_1$$

and since we have the assumption

$$S \leq \mathcal{A}_1 + \mathcal{A}_2$$

it suffices to prove

$$\mathcal{A}_1 + \mathcal{A}_2 = \mathcal{R} \cdot \frac{\mathcal{L}}{2} - \mathcal{A}_1$$

which is equivalent to

$$\mathcal{A}_2 = \mathcal{R} \cdot \frac{\mathcal{L}}{2} - 2\mathcal{A}_1$$

```
# e (ASM_REAL_ARITH_REWRITE_TAC `A_2 = R * L / &2 - &2 * A_1`
   []);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`A_2 = R * L / &2 - &2 * A_1`
```

In our assumptions, these terms are defined as sums:

$$\mathcal{A}_1 = \frac{1}{2}\sum_{i=0}^{m} a_i h_i$$

$$\mathcal{A}_2 = \sum_{i=0}^{m} a_i d_i$$

$$\frac{\mathcal{L}}{2} = \sum_{i=0}^{m} a_i$$

Hence we can rewrite the goal as

$$\mathcal{A}_2 = \sum_{i=0}^{m} a_i d_i = \mathcal{R}\sum_{i=0}^{m} a_i - \sum_{i=0}^{m} a_i h_i$$

```
# e (ASM_REAL_ARITH_REWRITE_TAC `sum (0..m) (\i. a i * d i) = R
    * sum (0..m) a - sum (0..m) (\i. a i * h i)` []);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`sum (0..m) (\i. a i * d i) =
 R * sum (0..m) a - sum (0..m) (\i. a i * h i)`
```

The right hand side we can unify into a single sum by applying the theorems:

$$\sum_i cf(i) = c\sum_i f(i)$$

$$\sum_i f(i) - g(i) = \sum_i f(i) - \sum_i g(i)$$

Ultimately resulting in the goal

$$\sum_{i=0}^{m} a_i d_i = \sum_{i=0}^{m} \mathcal{R}a_i - a_i h_i$$

```
# e (REWRITE_TAC [GSYM SUM_LMUL; GSYM SUM_SUB_NUMSEG]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`sum (0..m) (\i. a i * d i) =
 sum (0..m) (\i. R * a i - a i * h i)`
```

By applying MATCH_MP_TAC with the theorem

$$\forall_{0\leq i\leq m} f(i) = g(i) \implies \sum_{i=0}^{m} f(i) = \sum_{i=0}^{m} g(i)$$

we are left to prove

$$\forall_{0\leq i\leq m} a_i d_i = \mathcal{R} a_i - a_i h_i$$

```
# e (MATCH_MP_TAC SUM_EQ_NUMSEG);;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`!i. 0 <= i /\ i <= m ==> h i + d i = R`]
  3 [`S <= A_1 + A_2`]
  4 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  5 [`A_2 = sum (0..m) (\i. a i * d i)`]
  6 [`L / &2 = sum (0..m) a`]

`!i. 0 <= i /\ i <= m
 ==> (\i. a i * d i) i = (\i. R * a i - a i * h i) i`
```

Once more we remove the universal quantifier. But unlike with our initial tactic, this time we don't add the antecedent to the assumption list. Instead, we use it to simplify the already existing assumption

$$\forall_{0\leq i\leq m} h_i + d_i = R$$

which becomes

$$h_i + d_i = R$$

```
# e (GEN_TAC
    THEN DISCH_THEN (fun th1 -> FIRST_X_ASSUM (fun th2 ->
        REWRITE_TAC [GSYM (MP (SPEC_ALL th2) th1)]));;
val it : goalstack = 1 subgoal (1 total)

  0 [`A / &2 <= A_1`]
  1 [`S = &1 / &2 * pi * R pow 2`]
  2 [`S <= A_1 + A_2`]
  3 [`A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)`]
  4 [`A_2 = sum (0..m) (\i. a i * d i)`]
  5 [`L / &2 = sum (0..m) a`]

`a i * d i = (h i + d i) * a i - a i * h i`
```

At which point the goal is solvable using `ARITH_TAC`.

```
# e ARITH_TAC;;
val it : goalstack = No subgoals
```

A glossary of the code in this section can be found in D.

Chapter 5

Conclusions, discussion and future work

5.1 Conclusions

Our research focused on two major subjects. Firstly, we took a proof of the Isoperimetric Theorem by Nikolaos Dergiades and worked it out into great detail. Each step of the proof should logically lead into a formal proof using an interactive theorem prover.

Secondly, we developed definitions of polygon and triangle area in HOL Light. We then formalised elementary properties of these definitions. Additionally, we formally worked out the arithmetic at the conclusion of the proof by Dergiades, using automated tools we developed for this purpose.

5.2 Discussions

We chose to do research into the Isoperimetric Theorem primarily to challenge ourselves. Out of 100 theorems, we decided to work on 1 of 2 theorems which much more capable mathematicians decided to leave be. We had no expectations of formalising the complete theorem, and we are ultimately not surprised that we did not.

Unfortunately, not all of our attempts at formalisation were successful. Since the proof by Dergiades works with convex polygons, we attempted to formally define the notion of convexity. However, these attempts did not lead to meaningful definitions and lemmas, hence we did not include these results in our thesis. In the same sense, we attempted to formally define circular sectors, with similar results.

Even less fortunately, during the majority of our research we were convinced that there was an issue with the proof by Dergiades. During this time, we spent many resources into developing a workaround in order to correct the proof, since trying to formalise an incorrect proof is a hopeless endeavor. Only whilst working out our reasoning in this thesis did we notice the mistake in our reasoning.

Ultimately, we were not able to formally proof the Isoperimetric Theorem, even in the more specialised form used in the proof by Dergiades. In large part, this was due to the aforementioned unsuccessful attempts at formally defining convexity and circular sectors. In hindsight, it could have been beneficial to not want to reinvent the wheel and to look more closely into already existing formalisations.

5.3 Future work

The formalisation of many steps of the proof by Dergiades have been left for the future, due to both lack of time and in our competence in HOL Light. Perhaps with more practice and most importantly more knowledge of already existing formalisations we would be able to completely formalise all steps.

Mathematicians might decide that the Isoperimetric Theorem has to be proven in its most general form (for any surface and volume in any dimension) before it can be crossed off as formalised in Freek Wiedijk's list of 100 theorems. Unfortunately, this work of Dergiades does not prove this, nor do we suspect that any step in it might be useful in proving the general case. Hence future researcher will have to look to other proofs to formalise.

Bibliography

[1] Paul Abad and Jack Abad. The hundred greatest theorems.

[2] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq Formalization of Lebesgue Integration of Nonnegative Functions. Research Report RR-9401, Inria, France, April 2021.

[3] Anthony Bordg. Projective geometry. *Archive of Formal Proofs*, June 2018. `https://isa-afp.org/entries/Projective_Geometry.html`, Formal proof development.

[4] Christophe Brun, Jean-François Dufourd, and Nicolas Magaud. Designing and proving correct a convex hull algorithm with hypermaps in coq. *Comput. Geom. Theory Appl.*, 45(8):436–457, oct 2012.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Third Edition)*. The MIT Press, 2009.

[6] Marie Cousin, Mnacho Echenim, and Hervé Guiol. The hahn and jordan decomposition theorems. *Archive of Formal Proofs*, November 2021. `https://isa-afp.org/entries/Hahn_Jordan_Decomposition.html`, Formal proof development.

[7] Nikolaos Dergiades. An elementary proof of the isoperimetric inequality. *Forum Geometricorum*, 2:129–130, Oct 2002.

[8] William Dunham. *The mathematical universe.* John Wiley & Sons, Nashville, TN, February 1997.

[9] Penelope Gehring. The isoperimetric inequality: Proofs by convex and differential geometry. *Rose-Hulman Undergraduate Mathematics Journal*, 30(3), 2019.

[10] Gtgith. *Kakeya Needle Set.* June 2009.

[11] John Harrison. Hol light tutorial, January 2017.

[12] John Harrison. Proof of the jordan curve theorem. `https://github.com/jrh13/hol-light/blob/master/Jordan/jordan_curve_theorem.ml`, 2017.

[13] John Harrison. Lebesgue measure, measurable functions (defined via the gauge integral). `https://github.com/jrh13/hol-light/blob/master/Multivariate/measure.ml`, 2020.

[14] John Harrison. Results intended for flyspeck. `https://github.com/jrh13/hol-light/blob/master/Multivariate/flyspeck.ml`, 2021.

[15] John Harrison. Some geometric notions in realn̂. `https://github.com/jrh13/hol-light/blob/master/Multivariate/geom.ml`, 2021.

[16] John Harrison. 100 theorems. `https://github.com/jrh13/hol-light/tree/master/100`, 2022.

[17] Thomas Heath. *A History of Greek Mathematics Volume II.* Clarendon Press, 1921.

[18] Andreas Hehl. Handout: The isoperimetric inequality, February 2013.

[19] Peter Jones. *Reading Virgil: Aeneid I and II.* Cambridge University Press, Cambridge, England, March 2011.

[20] Gerwin Klein. The top 100 theorems in isabelle.

[21] Filip Marić and Danijela Simić. Complex geometry. *Archive of Formal Proofs*, December 2019. `https://isa-afp.org/entries/Complex_Geometry.html`, Formal proof development.

[22] Jan Moraal. Measure theory in waterproof : Interactive theorem-proving for measure theory in an educational setting. Bachelor's thesis, Eindhoven University of Technology, 2020.

[23] Paul J Nahin. *When least is best.* Princeton Science Library. Princeton University Press, Princeton, NJ, May 2021.

[24] Terence Tao. Dvir's proof of the finite field kakeya conjecture, March 2008.

[25] Ivor Thomas. *Selections Illustrating the History of Greek Mathematics.* Harvard University Press, 1941.

[26] Virgil. *The Aeneid.* Vintage Books, March 1990.

[27] Eric W. Weisstein. Isoperimetric problem. From MathWorld—A Wolfram Web Resource. Last visited on 21-8-2022.

[28] Freek Wiedijk. Formalizing 100 theorems.

Appendix A

Code Preliminaries

```
(* ---------------------------------------------------------- *)
(* Formalisations of all theorems discussed in the            *)
(* preliminaries.                                             *)
(* ---------------------------------------------------------- *)
needs "Multivariate/vectors.ml";;

(* ---------------------------------------------------------- *)
(* Definition of the cross product.                           *)
(* ---------------------------------------------------------- *)

parse_as_infix ("cross", (20,"right"));;
let cross = new_definition
    `(x:real^3) cross (y:real^3) =
       vector [x$2 * y$3 - x$3 * y$2;
               x$3 * y$1 - x$1 * y$3;
               x$1 * y$2 - x$2 * y$1] :real^3`;;

(* ---------------------------------------------------------- *)
(* Newly defined tactic and conversion to streamline the      *)
(* formalisation process when working with vectors in three   *)
(* dimensions.                                                *)
(* ---------------------------------------------------------- *)

let VEC3_TAC =
    SIMP_TAC [CART_EQ; LAMBDA_BETA; FORALL_3; SUM_3;
              DIMINDEX_3; VECTOR_3;
              VECTOR_ADD_COMPONENT; VECTOR_SUB_COMPONENT;
              VECTOR_NEG_COMPONENT; VECTOR_MUL_COMPONENT;
              vector_add; vec; dot; cross; orthogonal; basis;
              ARITH]
    THEN CONV_TAC REAL_RING;;

let VEC3_RULE tm = prove (tm, VEC3_TAC);;
```

```
(* ----------------------------------------------------------- *)
(* Simple example usage of the above defined conversion.     *)
(* ----------------------------------------------------------- *)

let CROSS_REFL = VEC3_RULE `!x. x cross x = vec 0`;;
let CROSS_RZERO = VEC3_RULE `!x. x cross vec 0 = vec 0`;;
let CROSS_MUL_LDISTRIB =
    VEC3_RULE`!c x y. (c % x) cross y = c % (x cross y)`;;

(* ----------------------------------------------------------- *)
(* Slightly more difficult example formalisation.             *)
(* ----------------------------------------------------------- *)

let PARALLEL_SCALAR_MULT_EQUIV = prove (
    `!x y. (?c. x = c % y) \/ y = vec 0 <=> x cross y = vec 0`,
    REPEAT GEN_TAC
    THEN EQ_TAC
    THENL [
        STRIP_TAC THEN POP_ASSUM SUBST1_TAC THEN VEC3_TAC;

        MATCH_MP_TAC (TAUT `(~B ==> ~A) ==> A ==> B`)
        THEN REWRITE_TAC [cross; CART_EQ; LAMBDA_BETA;
                          FORALL_3; DIMINDEX_3; VECTOR_3;
                          VEC_COMPONENT; VECTOR_MUL_COMPONENT;
                          NOT_EXISTS_THM; DE_MORGAN_THM]
        THEN STRIP_TAC
        THEN POP_ASSUM (fun th1 -> POP_ASSUM (fun th2 ->
            let i = (rand o rand o rator o rand o concl) th1 in
                MP_TAC (SPEC (vsubst [i,`i:num`; i, `j:num`]
                    `(x:real^3)$i / (y:real^3)$j`) th2)
                THEN MP_TAC th1))
        THEN CONV_TAC REAL_FIELD
    ]
);;
```

Appendix B

Code Altered Cross Product

```
(* ------------------------------------------------------------ *)
(* Altered version of the cross product.                        *)
(* ------------------------------------------------------------ *)

parse_as_infix("cross",(20,"right"));;

let cross = new_definition
  `(x:real^2) cross (y:real^2) = x$1 * y$2 - x$2 * y$1`;;

(* ------------------------------------------------------------ *)
(* Elementary properties of the altered cross product.          *)
(* ------------------------------------------------------------ *)

let CROSS_REFL = prove (
  `!x. x cross x = &0`,
  REWRITE_TAC[cross]
  THEN ARITH_TAC
);;

let CROSS_SYM = prove (
  `!x y. x cross y = --(y cross x)`,
  REWRITE_TAC[cross]
  THEN ARITH_TAC
);;

let CROSS_LZERO = prove (
  `!x. vec 0 cross x = &0`,
  REWRITE_TAC[cross; VEC_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_RZERO = prove (
  `!x. x cross vec 0 = &0`,
  REWRITE_TAC[cross; VEC_COMPONENT]
  THEN ARITH_TAC
);;
```

```
let CROSS_ADD_LDISTRIB = prove (
  `!x y z. x cross (y + z) = (x cross y) + (x cross z)`,
  REWRITE_TAC[cross; VECTOR_ADD_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_ADD_RDISTRIB = prove (
  `!x y z. (x + y) cross z = (x cross z) + (y cross z)`,
  REWRITE_TAC[cross; VECTOR_ADD_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_SUB_LDISTRIB = prove (
  `!x y z. x cross (y - z) = (x cross y) - (x cross z)`,
  REWRITE_TAC[cross; VECTOR_SUB_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_SUB_RDISTRIB = prove (
  `!x y z. (x - y) cross z = (x cross z) - (y cross z)`,
  REWRITE_TAC[cross; VECTOR_SUB_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_MUL_LDISTRIB = prove (
  `!c x y. (c % x) cross y = c * (x cross y)`,
  REWRITE_TAC[cross; VECTOR_MUL_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_MUL_RDISTRIB = prove (
  `!c x y. x cross (c % y) = c % (x cross y)`,
  REWRITE_TAC[cross; VECTOR_MUL_COMPONENT]
  THEN ARITH_TAC
);;

let CROSS_SUM = prove (
  `!w x y z. w cross x + y cross z =
             (w - y) cross (x - z) + w cross z + y cross x`,
  REWRITE_TAC[cross; VECTOR_SUB_COMPONENT]
  THEN ARITH_TAC
);;

let LAGRANGE_ID = prove (
  `!w x y z. (w cross x) * (y cross z) =
             (w dot y) * (x dot z) - (w dot z) * (x dot y)`,
  REWRITE_TAC[cross; DOT_2]
  THEN ARITH_TAC
);;
```

Appendix C

Code Polygon Area

```
(* ---------------------------------------------------------- *)
(* Formalisations of polygon perimeter & area, along with    *)
(* elementary lemmas.                                         *)
(* ---------------------------------------------------------- *)

(* ---------------------------------------------------------- *)
(* Lemmas for the area / surface of a triangle                *)
(* ---------------------------------------------------------- *)

let triangle_area = new_definition
    `triangle_area x y z = ((y - x) cross (z - x)) / &2`;;

let TRIANGLE_AREA_ADD = prove (
    `!w x y z. triangle_area w x y + triangle_area z y x =
               triangle_area x z w + triangle_area y w z`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;

let TRIANGLE_AREA_ROTATE = prove (
    `!x y z. triangle_area x y z = triangle_area z x y`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;

let TRIANGLE_AREA_REFLECT = prove (
    `!x y z. triangle_area x y z = --triangle_area x z y`,
    REWRITE_TAC[triangle_area; cross; VECTOR_SUB_COMPONENT]
    THEN ARITH_TAC
);;
```

```
(* ---------------------------------------------------------- *)
(* Lemmas for the area / surface of a polygon                 *)
(* ---------------------------------------------------------- *)

let polychain_area = define
    `polychain_area _ _' [] = &0
     /\ polychain_area O x (CONS y l) =
        triangle_area O x y + polychain_area O y l`;;

let polygon_area = new_definition
    `polygon_area O l = if l = []
                       then &0
                       else polychain_area O (LAST l) l`;;

let POLYGON_AREA_ORIGIN_INDEPENDENCE = prove (
    `!O O' l. polygon_area O l = polygon_area O' l`,
    GEN_TAC THEN GEN_TAC
    THEN LIST_INDUCT_TAC
    THENL [REWRITE_TAC [polygon_area]; ALL_TAC]
    THEN POP_ASSUM MP_TAC
    THEN ASM_CASES_TAC `t:(real^2)list = []`
    THEN ASM_REWRITE_TAC [polygon_area; NOT_CONS_NIL; LAST;
                          polychain_area]
    THENL [REWRITE_TAC [triangle_area; CROSS_REFL]; ALL_TAC]

    THEN POP_ASSUM (fun th1 -> ONCE_REWRITE_TAC
        [(GEN_ALL o MK_COMB) (REFL `polychain_area O h`,
            MATCH_MP CONS_HD_TL th1)])
    THEN REWRITE_TAC [polychain_area]
    THEN STRIP_TAC

    THEN SUBGOAL_THEN `!O. triangle_area O (LAST t) h +
                           triangle_area O h (HD t) =
                           triangle_area h (HD t) (LAST t) +
                           triangle_area O (LAST t) (HD t)`
                      (fun th ->
                       REWRITE_TAC [REAL_ADD_ASSOC; th]
                       THEN REWRITE_TAC [GSYM REAL_ADD_ASSOC])
    THENL [MESON_TAC [TRIANGLE_AREA_ROTATE; TRIANGLE_AREA_ADD];
           ALL_TAC]
    THEN ASM_REWRITE_TAC []
);;




let POLYGON_AREA_ADD = prove (
    `!O l1 l2. polygon_area O (APPEND l1 l2) =
               polygon_area O l1 + polygon_area O l2 +
               if l1 = [] \/ l2 = []
               then &0
               else polygon_area O [HD l1; LAST l1;
                                    HD l2; LAST l2]`,
    GEN_TAC
    THEN LIST_INDUCT_TAC
    THENL [REWRITE_TAC [APPEND; polygon_area; REAL_ADD_RID;
                        REAL_ADD_LID]; ALL_TAC]
    THEN GEN_TAC
    THEN POP_ASSUM (fun th -> MP_TAC (SPEC_ALL th))
    THEN ASM_CASES_TAC `l2:(real^2)list = []`
    THENL [ASM_REWRITE_TAC [APPEND_NIL; polygon_area;
                            REAL_ADD_RID]; ALL_TAC]
    THEN ASM_CASES_TAC `t:(real^2)list = []`
    THENL [
        FIRST_ASSUM (fun th -> ASM_REWRITE_TAC [
            APPEND; NOT_CONS_NIL; HD; LAST;
            polygon_area; polychain_area;
            (GEN_ALL o MK_COMB)
            (REFL `polychain_area 0 h`,
                MATCH_MP CONS_HD_TL th);
            REWRITE_RULE [CROSS_REFL; real_div; REAL_MUL_LZERO]
                (ISPECL [`h:real^2`; `h:real^2`; `0:real^2`]
                        triangle_area);
            ISPECL [`0:real^2`; `(HD l2):real^2`;
                `(LASTl2):real^2`] TRIANGLE_AREA_REFLECT
        ]) THEN ARITH_TAC;

         POP_ASSUM (fun th1 -> POP_ASSUM (fun th2 ->
         REWRITE_TAC [
            APPEND; APPEND_EQ_NIL; NOT_CONS_NIL;
            HD; LAST; LAST_APPEND; HD_APPEND; TL;
            polygon_area; polychain_area; th2; th1;
            (GEN_ALL o MK_COMB)
                (REFL `polychain_area 0 h`,
                    MATCH_MP CONS_HD_TL th2);
            (GEN_ALL o MK_COMB)
                (REFL `polychain_area 0 h`,
                    MATCH_MP CONS_HD_TL th1);
            (GEN_ALL o MK_COMB)
                (REFL `polychain_area 0 h`,
                    MATCH_MP CONS_HD_TL
                (REWRITE_RULE [DE_MORGAN_THM; th2]
                    (((MATCH_MP MONO_NOT) o fst o EQ_IMP_RULE)
                    (ISPECL[`t:(real^2)list`;`l2:(real^2)list`]
                            APPEND_EQ_NIL))));
            REWRITE_RULE [CROSS_REFL; real_div; REAL_MUL_LZERO]
                (ISPECL [`h:real^2`; `h:real^2`; `0:real^2`]
                        triangle_area);
            ISPECL [`0:real^2`; `(HDl2):real^2 `;
                `(LAST l2):real^2`] TRIANGLE_AREA_REFLECT
        ])) THEN ARITH_TAC
    ]
);;
```

Appendix D

Code Arithmetic

```
(* ---------------------------------------------------------- *)
(* Formalisation of all arithmetic in the conclusion of the   *)
(* proof.                                                     *)
(* ---------------------------------------------------------- *)
needs "Library/transc.ml";;

(* ---------------------------------------------------------- *)
(* Newly defined conversions and tactics to streamline the    *)
(* formalisation process when complicated arithmetic and      *)
(* assumption are involved.                                   *)
(* ---------------------------------------------------------- *)

let (REAL_ARITH_REWRITE_CONV: term -> thm list -> conv) =
    fun post ths ->
        if ths = []
        then curry (REAL_FIELD o mk_imp) post
        else C MP (end_itlist CONJ (map SPEC_ALL ths))
               o REAL_FIELD
               o (curry mk_imp ((list_mk_conj o map
                    (snd o strip_forall o concl)) ths))
               o (curry mk_imp post);;

let (REAL_ARITH_REWRITE_TAC: term -> thm list -> tactic) =
    fun post ths ->
        W (MATCH_MP_TAC o REAL_ARITH_REWRITE_CONV post ths
           o snd);;


let ASM_REAL_ARITH_REWRITE_TAC = ASM o REAL_ARITH_REWRITE_TAC;;
```

```
(* ---------------------------------------------------------- *)
(* Main theorem                                                *)
(* ---------------------------------------------------------- *)

prove (
    `!A A_1 A_2 L R S a d h m.
       A / &2 <= A_1
    /\ S = &1 / &2 * pi * R pow 2
    /\ (!i. 0 <= i /\ i <= m ==> h i + d i = R)
    /\ S <= A_1 + A_2
    /\ A_1 = &1 / &2 * sum (0..m) (\i. a i * h i)
    /\ A_2 = sum (0..m) (\i. a i * d i)
    /\ L / &2 = sum (0..m) a
    ==> &4 * pi * A <= L pow 2`,

    REPEAT STRIP_TAC

    THEN ASM_REAL_ARITH_REWRITE_TAC
        `&0 <= L pow 2 / (&4 * pi) - &2 * A_1`
        [SPECL [`&4 * pi * A`; `L pow 2`; `&4 * pi`]
               REAL_LE_DIV2_EQ;
        SPECL [`A:real`; `&4 * pi`] REAL_DIV_LMUL;
        PI_POS]

    THEN REAL_ARITH_REWRITE_TAC
        `pi * (R - L / (&2 * pi)) pow 2 -
         (L pow 2 / (&4 * pi) - &2 * A_1) <= &0`
        [SPEC `R - L / (&2 * pi)` REAL_LE_POW_2;
        SPECL [`pi`; `&0`; `(R - L / (&2 * pi)) pow 2`]
              REAL_LE_LMUL;
        PI_POS]

    THEN REWRITE_TAC [MP (REAL_FIELD `&0 < pi ==>
        pi * (R - L / (&2 * pi)) pow 2 -
        (L pow 2 / (&4 * pi) - &2 * A_1) =
        pi * R pow 2 - R * L + &2 * A_1`)
        PI_POS]

    THEN REAL_ARITH_REWRITE_TAC
        `&1 / &2 * pi * R pow 2 <= R * L / &2 - A_1` []
    THEN ASM_REAL_ARITH_REWRITE_TAC
        `A_2 = R * L / &2 - &2 * A_1` []
    THEN ASM_REAL_ARITH_REWRITE_TAC
        `sum (0..m) (\i. a i * d i) =
         R * sum (0..m) a -sum (0..m) (\i. a i * h i)` []
    THEN REWRITE_TAC [GSYM SUM_LMUL; GSYM SUM_SUB_NUMSEG]
    THEN MATCH_MP_TAC SUM_EQ_NUMSEG
    THEN GEN_TAC
    THEN DISCH_THEN (fun th1 -> FIRST_X_ASSUM (fun th2 ->
        REWRITE_TAC [GSYM (MP (SPEC_ALL th2) th1)]))
    THEN ARITH_TAC
);;
```