

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Verification of Convex Hull Algorithms in
Isabelle/HOL**

Author

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Verification of Convex Hull Algorithms in
Isabelle/HOL**

Titel der Abschlussarbeit

Author:	Author
Supervisor:	Prof. Nipkow
Advisor:	Lukas Stevens
Submission Date:	Submission date

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, Submission date

Author

Acknowledgments

Abstract

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
1.1 Section	1
1.1.1 Subsection	1
2 Definitions and Algorithms	3
2.1 Convex Hull	3
2.1.1 Basics	3
2.1.2 Orientation	4
2.1.3 Order	6
2.1.4 Convex Polygon	7
2.2 Jarvis-March Algorithm	9
2.2.1 Definition of the Algorithm	9
2.2.2 Jarvis March is Convex Hull	13
2.2.3 Computability	16
2.3 Graham Scan	17
2.3.1 Colinearity	20
2.3.2 Angle Comparator	21
2.4 Chans Algorithm	22
2.4.1 Subsection	22
3 Definitions and Algorithms	24
3.1 Convex Hull	24
3.2 Jarvis-March Algorithm	24
3.3 Graham Scan	25
3.4 Chans Algorithm	25
3.4.1 Subsection	25
Abbreviations	27

Contents

List of Figures	28
List of Tables	29
Bibliography	30

1 Introduction

1.1 Section

Citation test [Lam94].

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}`, `\ac{TUM}` \Rightarrow Technical University of Munich (TUM), TUM

For more details, see the documentation of the acronym package¹.

1.1.1 Subsection

See Table 3.1, Figure 3.1, Figure 3.2, ??.

Table 1.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

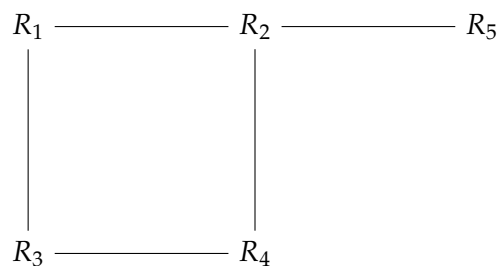


Figure 1.1: An example for a simple drawing.

!TeX root = ../main.tex

¹<https://ctan.org/pkg/acronym>

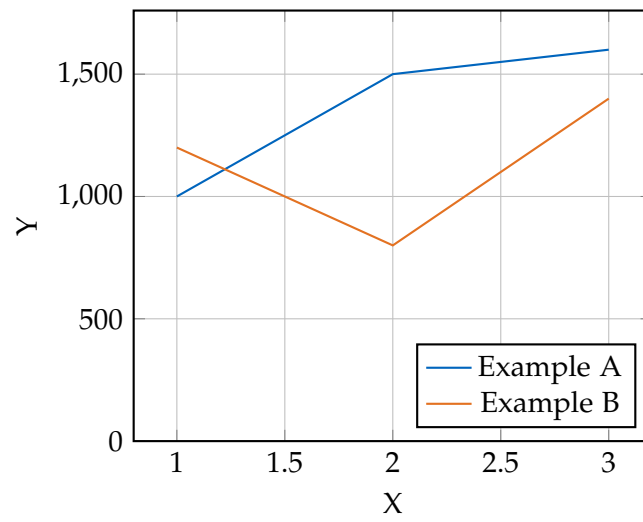


Figure 1.2: An example for a simple plot.

2 Definitions and Algorithms

2.1 Convex Hull

2.1.1 Basics

First the Convex Hull will be defined. A set $s \subseteq \mathbb{R}^2$ is convex if for every two points p and q in s it holds that all points on the line segment connecting p and q are in s again. This can be expressed, as the fact that the any convex combination of p and q has to be in s again. In Isabelle the convex predicate is defined exactly this way:

```
definition convex :: 'a real_vector set  $\Rightarrow$  bool where
convex s  $\longleftrightarrow$  ( $\forall x \in s. \forall y \in s. \forall u \geq 0. \forall v \geq 0. u + v = 1 \longrightarrow u * x + v * y \in s$ )
```

The convex hull of a set s is the smallest convex set in which s is contained. There are several alternative ways in which the convex hull can be defined. One possible way is to define it as the intersection of all convex sets containing s , which is also the definition used in Isabelle/HOL. We have already seen the convex predicate, the hull predicate is defined as the intersection of all sets t that contain s and fulfill the predicate S .

```
definition hull :: (a' set  $\Rightarrow$  bool)  $\Rightarrow$  a' set  $\Rightarrow$  a' set where
S hull s =  $\bigcap \{t. S t \wedge s \subseteq t\}$ 
```

Consequently `convex hull s` refers to the intersection of all convex sets that contain s and therefore the convex hull of the set s . In the two dimensional case for a finite $s \subset \mathbb{R}^2$, the convex hull CH of s is a convex polygon and all the corners of this convex polygon are points from S (see figure 1)[De 00]. As this thesis will focus on the two dimensional case and only give an outlook on the the three dimensional case, we will deal with computing the convex hull of $s \in \mathbb{R}^2$ in the following and therefore computing a convex polygon as representation of the convex hull of s . Assuming no three points in s are colinear, then the edges $E \subseteq s^2$ of the polygon can be described as exactly those $(p, q) \in s^2$ for which all points in s lie on the left of the vector \vec{pq} . Notice that the direction of the vector i.e. from p to q is relevant for expressing that a point lies on the left of the vector \vec{pq} . Of course the symmetric definition of E as those $(p, q) \in s^2$ for which all points in s lie on the right of the line \vec{pq} works as well. The only difference is that in the set of directed edges we get, every edge now points into the opposite direction. Both definitions make sense, but because there is already infrastructure in

place for first definition i.e. (p, q) is an edge if and only if all points in s are left of \vec{pq} , we will use this definition. But first we need to state the concept of a point q being left of the vector \vec{pq} more precicsely, especially when there can be three colinear points in s .

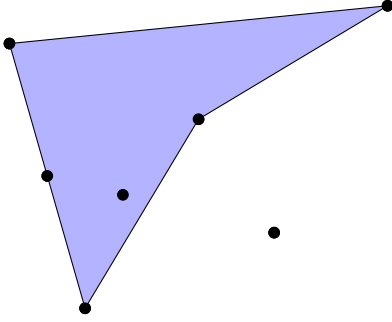


Figure 2.1: Non-convex set in 2D

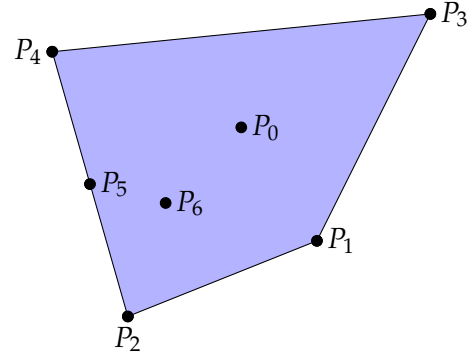


Figure 2.2: Convex polygon, which is the convex hull of the set of points $\{P_0, P_1, P_2, P_3, P_4, P_5, P_6\}$

2.1.2 Orientation

Figure 2.2 shows the convex hull of the points $s = \{P_0, P_1, P_2, P_3, P_4, P_5, P_6\}$ in the form of a convex polygon. When using the previous definition, (P_4, P_5) , (P_5, P_6) and (P_4, P_6) would be edges of the convex polygon, because it holds that all points in s are left of $\vec{P_4P_5}$, left of $\vec{P_5P_6}$ and left of $\vec{P_4P_6}$. This is an unintuitive definition which should be avoided. Therefore we define the condition for (p, q) to be an edge of the convex hull polygon more precicsely. $(p, q) \in s^2$ is an edge of the convex hull polygon if and only if all points $r \in s$ are either strictly left of the vector \vec{pq} (p, q and r are not colinear) or r is contained in the closed segment between p and q . The second part can be written as $r \in \text{closed_segment } p \ q$ in Isabelle where `closed_segment` is defined as:

```
definition closed_segment :: 'a::real_vector  $\Rightarrow$  'a  $\Rightarrow$  'a set
where closed_segment a b = {(1 - u) *R a + u *R b | u::real. 0  $\leq$  u  $\wedge$  u  $\leq$  1 }
```

The fact that r lies strictly left of \vec{pq} can be expressed differently by stating that (p, q, r) are making a strictly counterclockwise turn. The three points are written as a tuple as it is again necessary to state the order of p, q and r when talking about a

counterclockwise turn. In the following a counterclockwise turn will always refer to a strict counterclockwise turn. Checking if a point r lies strictly left of a vector is an operation that is essential for almost all convex hull algorithms. To check if the points $((x_1, y_1), (x_2, y_2), (x_3, y_3))$ make a counterclockwise turn, we can look at the sign of the determinant of the following matrix.

$$\det \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

If the determinant is positive, we know that the sequence $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ makes a counterclockwise turn, if the determinant is zero we know that the three points are colinear and if the determinant is negative, we know the the sequence $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ makes a clockwise turn. In Isabelle the function that calculates the above determinant for three points is called `det3`.

```
fun det3:: point ⇒ point ⇒ point ⇒ real where
det3 (x1, y1) (x2, y2) (x3, y3) =
x1 * y2 + y1 * x3 + x2 * y3 - y2 * x3 - y1 * x2 - x1 * y3"
```

Based on `det3` the `ccw'` predicate is defined, which expresses that three points (p, q, r) make a counterclockwise turn.

```
definition ccw' p q r ⟷ 0 < det3 p q r
```

Lastly the predicate `ccw'_seg p q r` holds if and only if r either lies counterclockwise of \vec{pq} or r is contained in the closed segment between p and q .

```
definition ccw'_seg p q r = ccw' p q r ∨ r ∈ closed_segment p q
```

Intuition

It is not intuitively clear why $0 < \text{det3 } (x_0, y_0) (x_1, y_1) (x_2, y_2)$ ensures a counterclockwise orientation. The `det3` function can be interpreted as calculating the determinant of the previously described matrix, but the calculations of the `det3` function can also be reformulated.

```
lemma det_form: "det3 (x0,y0) (x1,y1) (x2,y2) =
(x1 - x0) * (y2 - y0) - (x2 - x0) * (y1 - y0)"
```

Using this definition $0 < \text{det3 } (x_0, y_0) (x_1, y_1) (x_2, y_2)$ means $(x_1 - x_0) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0) > 0$, which can be reformulated to $(x_1 - x_0) * (y_2 - y_0) > (x_2 - x_0) * (y_1 - y_0)$. Now assuming $y_2 \neq y_0 \wedge y_1 \neq y_0$ and $\text{sign}(y_2 - y_0) = \text{sign}(y_1 - y_0)$, we get $(y_2 - y_0) / (x_2 - x_0) > (y_1 - y_0) / (x_1 - x_0)$.

Therefore the points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ are orientated counterclockwise if the slope of $\overrightarrow{p_0 p_2}$ is greater than the slope of $\overrightarrow{p_0 p_1}$ with $p_0 = (x_0, y_0), p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. This should make the connection between the det3 function and the geometric statement p_2 lies counterclockwise of $\overrightarrow{p_0 p_1}$ more obvious. Lastly even if $\text{sign}(y_2 - y_0) \neq \text{sign}(y_1 - y_0)$ is the case and $y_2 \neq y_0 \wedge y_1 \neq y_0$ still holds, we are still comparing the slopes, the $>$ just changes to a $<$.

2.1.3 Order

In both algorithms we need to do the following operation. Given an corner p of the convex polygon, find another corner by searching for a point q such that for all other points $r \in s$ either $\text{ccw}'(p, q, r)$ or $r \in \text{closed_segment}(p, q)$ holds. In short, we search for a q that fulfills $\forall r \in s. \text{ccw}'_{\text{seg}}(p, q, r)$. Intuitively it makes sense that given a finite $s \subseteq \mathbb{R}^2$ and a corner of the convex hull polygon, we can find a unique next corner. Figuratively speaking, we rotate a line that starts in p counterclockwise until we hit a point q , which is going to be the next corner. If we hit several points at the same time, we are just going to take the point further away from p . Now to translate this into a formal framework, we start with the previous definition of finding a q that fulfills $\forall r \in s. (\text{ccw}'_{\text{seg}}(p) \ q \ r)$. If $(\text{ccw}'_{\text{seg}}(p))$ is a total order on s , we know that such a q exists. That's because q is the minimum respect to the ordering $(\text{ccw}'_{\text{seg}}(p))$. For $(\text{ccw}'_{\text{seg}}(p))$ to be a total order and for later proofs it is necessary that we derive some form of transitivity for the counterclockwise orientation. For example it should hold that if $(\text{ccw}'_{\text{seg}}(p) \ a \ b)$ and $(\text{ccw}'_{\text{seg}}(p) \ b \ c)$ holds, then $(\text{ccw}'_{\text{seg}}(p) \ a \ c)$ should hold as well. The same implication should hold when using the $(\text{ccw}'_{\text{seg}}(p))$ ordering instead of $(\text{ccw}'_{\text{seg}}(p))$. Although straightforward, this kind of transitivity does not always hold as the following example shows.

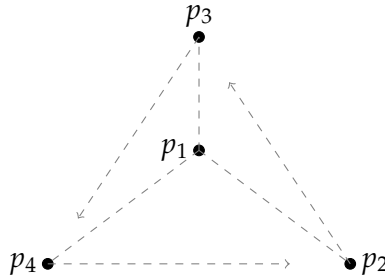


Figure 2.3

Clearly $(\text{ccw}'(p_1, p_2, p_3))$ holds and also $(\text{ccw}'(p_1, p_3, p_4))$, but $(\text{ccw}'(p_1, p_2, p_4))$ does not hold, instead $(\text{ccw}'(p_1, p_4, p_2))$ holds. So in order for transitivity to hold, we need

to restrict the set on which transitivity is supposed to hold. It can be shown that if there exists a p_0 such that for all $r \in s$ it holds that $\text{ccw}'_seg\ p_0\ p_1\ r$ holds, then $(\text{ccw}'_seg\ p_1)$ is transitive on s . This restriction avoids the counterexample for general transitivity from above. Transitivity also holds if there exists a point p_0 such that all $r \in s$ are lexicographically bigger than p_0 , meaning $\forall r \in s. \text{lex}\ p_0\ r$ holds, where lex is defined as.

```
definition lex :: point  $\Rightarrow$  point  $\Rightarrow$  bool where
"lex p q  $\longleftrightarrow$  (fst p < fst q  $\vee$  fst p = fst q  $\wedge$  snd p < snd q  $\vee$  p = q)"
```

To check if p is lexicographically smaller than q , we check if p_x is smaller than q_x . If they are equal we check if $p_y \leq q_y$ holds. Now given for our reference set $ps \subseteq \mathbb{R}^2$ if $(\forall q \in ps. \text{ccw}'_seg\ p_st1\ p_last\ q) \vee (\forall q \in ps. \text{lex}\ p_last\ q)$ holds, then the following lemmas can be proven.

```
lemma ccw'_seg_trans:
assumes "p  $\in$  ps" "q  $\in$  ps" "k  $\in$  ps"
assumes "ccw'_seg p_last p q" "ccw'_seg p_last k p"
shows "ccw'_seg p_last k q"
```

```
lemma ccw'_seg_total:
assumes "p  $\in$  ps" "q  $\in$  ps"
shows "ccw'_seg p_last p q  $\vee$  ccw'_seg p_last q p"
```

```
lemma ccw'_seg_antisymmetric:
assumes "ccw'_seg p_last p q  $\wedge$  ccw'_seg p_last q p"
shows "p = q"
```

Reflexivity directly follows from the definition of ccw'_seg . Therefore we know that there exists a unique q such that $\forall r \in ps. (\text{ccw}'_seg\ p_last)\ q\ r$. Notice how ps was defined using p_last .

2.1.4 Convex Polygon

Both algorithms calculate the convex polygon that corresponds to the convex hull of the input set $s \subseteq \mathbb{R}^2$. This convex polygon is described by a list of points from s that are the corners of this convex polygon. So far, we just always just stated that the convex polygon corresponds to the convex hull, yet it is not obvious that this is the case. Therefore we require a description of a convex polygon in Isabelle/HOL and we need to know that this description is indeed equivalent to `convex hull`, which is defined as Intersection of all convex sets that contain s . To be more precise, we require a proof that the convex hull of the corners of such a convex polygon corresponds to the set of

all points that lie within the polygon. This fact was proven for a list of corners $p_0 \# ps$ that should represent a convex polygon by Simon Hanssen.

```
lemma polygon_eq_convex_hull:
  assumes turns_only_left (p0 # ps)
    and sorted_wrt (ccw' p0) ps
    and  $2 \leq \text{length } ps$ 
  shows list_all (encompasses p) (polychain_of (p0 # ps @ [p0]))
     $\longleftrightarrow p \in \text{convex hull (set (p0 \# ps))}$ "
```

To understand this proof, we need to first look at the definitions of all the predicates used. First `turns_only_left l` for a list l expresses that every three consecutive points in the list are turning counterclockwise. This ensures that every interior angle of the polygon is less than 180° , which is one of the common definitions of a convex polygon.

```
fun turns_only_left :: "point list  $\Rightarrow$  bool" where
  "turns_only_left (p#q#r#ps)  $\longleftrightarrow$  ccw' p q r  $\wedge$  turns_only_left (q#r#ps)" |
  "turns_only_left _ = True"
```

Next `sorted_wrt (ccw' p0)`, where p_0 is the start of our list of corners, states that for every corner p in the list, all corners that are behind it in the list, lie counterclockwise of $\overrightarrow{p_0 p}$.

```
fun sorted_wrt :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool" where
  "sorted_wrt P [] = True" |
  "sorted_wrt P (x # ys) = (( $\forall y \in \text{set } ys. P x y$ )  $\wedge$  sorted_wrt P ys)"
```

This ensures that the list of corners even represents a polygon without degenerations like the one shown in Figure 2.4. Clearly `turns_only_left [p0, p1, p2, p3, p4, p5, p6]` holds, but `sorted_wrt (ccw' p0) [p1, p2, p3, p4, p5, p6]` does not hold, as the structure in Figure 2.4 is not a valid polygon.

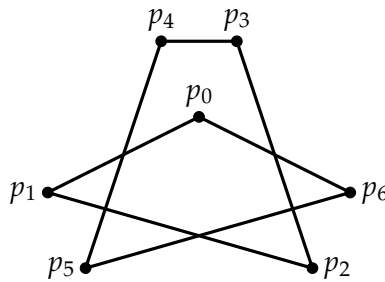


Figure 2.4

Lastly $2 \leq \text{length } ps$ is needed, as the definition does not work in the case of two corners, where the polygon is just a closed segment between two points. Now given

a list $p_0 \# ps$ fulfills these properties, then we know that this list describes a list of corners of a convex polygon and the following statement holds.

```
list_all (encompasses p) (polychain_of (p0 # ps @ [p0]))
   $\longleftrightarrow$   $p \in \text{convex hull (set (p0 \# ps))}$ 
```

Where $\text{polychain_of } (p_0 \# ps @ [p_0])$ is just the list of all tuples of two consecutive points in the list and $\text{encompasses } p \text{ seg} = \det3 \text{ (fst seg) (snd seg) } p \geq 0$ states that p lies counterclockwise (or colinear) of the vector $\overrightarrow{(\text{fst seg})(\text{snd seg})}$. With (fst seg) being the first point in the tuple seg and (snd seg) being the second point in the tuple seg .

```
fun polychain_of where
  "polychain_of [] = []"
  "polychain_of [p2] = []"
  "polychain_of (p1#p2#ps) = (p1, p2) # polychain_of (p2 # ps)"
```

The $\text{list_all } P \text{ l}$ predicate states that the condition P has to hold for every element in the list l . Consequently $\text{list_all (encompasses } p) (\text{polychain_of } (p_0 \# ps @ [p_0]))$ states that p lies inside the polygon defined by $p_0 \# ps$ as it requires that p lies counterclockwise (or colinear) of every edge of the polygon. Therefore the lemma $\text{polygon_eq_convex_hull}$ states that a point p lies inside the convex polygon defined by $p_0 \# ps$ if and only if p is in the convex hull of $\text{set } (p_0 \# ps)$. Now we have the definition of a convex polygon and the proof that the convex hull of the corners of such a polygon corresponds to the set of all points that lie within the polygon. Based on this we can show that the inspected algorithms, for an input set s , compute a convex polygon according to the definition and that the convex hull that corresponds to this convex polygon is indeed the convex hull of s .

2.2 Jarvis-March Algorithm

2.2.1 Definition of the Algorithm

The Jarvis March or Gift-Wrapping Algorithm is a simple output-sensitive way of calculating the convex hull of a given finite set $S \subseteq \mathbb{R}^2$ of points. It calculates the convex hull by calculating the corresponding convex polygon and returning an ordered list of the corners of the polygon. The algorithm has runtime $O(n * h)$, where n is the number of points in S and h is the number of points that lie on the convex hull or the number of corners on the calculated polygon to be more precise. The algorithm starts by choosing a point that is guaranteed to lie on the convex hull. We will use the lexicographical minimum $p_0 = \min_y \min_x S$, or in Isabelle terms the $p0$, which fulfills

$\forall q \in r. \text{lex } p_0 \ r$. Then the next corner of the convex polygon is found by searching a p_1 such that every point $r \in s$ lies counterclockwise of $\vec{p_0 p_1}$ or is contained in the closed segment between p_0 and p_1 , meaning $\forall r \in ps. (\text{ccw}'_seg \ p_0) \ p_1 \ r$ should hold. As explained in 2.1.3 we know that such a p_1 exists, because $\forall r \in ps. \text{lex } p_0 \ r$ holds. In Isabelle the definition for finding the minimum with respect to the total order $(\text{ccw}'_seg \ p_0)$ is.

```
definition ccw'_seg_min :: " point set  $\Rightarrow$  point" where
"ccw'_seg_min ps = (THE p. p  $\in$  ps  $\wedge$  ( $\forall$  q  $\in$  ps. ccw'_seg p0 p q))"
```

Now from 2.1.1, we know that (p_0, p_1) is an edge of the wanted convex polygon and we know that p_1 is once again a point on the convex hull and a corner of the polygon as (p_0, p_1) fulfills $\forall r \in ps. (\text{ccw}'_seg \ p_0) \ p_1 \ r$. Therefore we can repeat the previous step and search for a p_2 that fulfills $\forall r \in ps. (\text{ccw}'_seg \ p_1) \ p_2 \ r$. Once again according to 2.1.3, we know that $\forall r \in ps. (\text{ccw}'_seg \ p_0) \ p_1 \ r$ holds and therefore $(\text{ccw}'_seg \ p_1)$ is a total order and a unique p_2 exists. Again p_2 has to be a corner of the convex polygon and (p_1, p_2) an edge on of the polygon. The algorithm continues until a $p_h = p_0$ is found to be the next point and stops, because the first corner of the polygon is encountered again. The ordered sequence of points p_0, p_1, \dots, p_{h-1} are the corners of the convex polygon and $(p_0, p_1), (p_1, p_2), \dots, (p_{h-2}, p_{h-1}), (p_{h-1}, p_0)$ are the edges of the polygon. Figure x shows the steps of Jarvis March computing the convex hull of a input set ps .

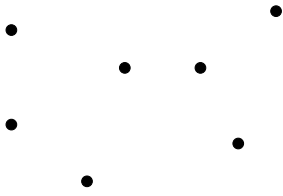


Figure 2.5

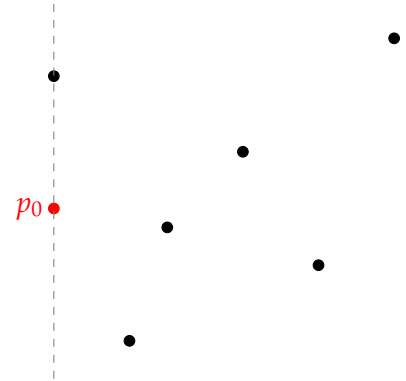


Figure 2.6

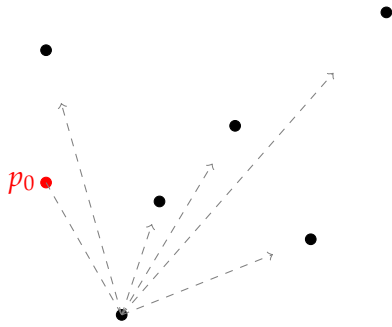


Figure 2.7

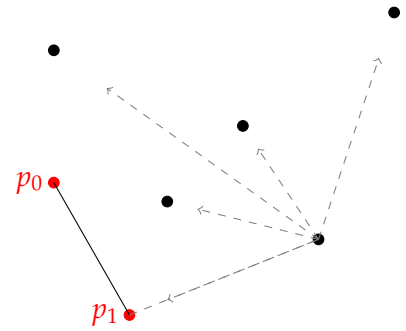


Figure 2.8

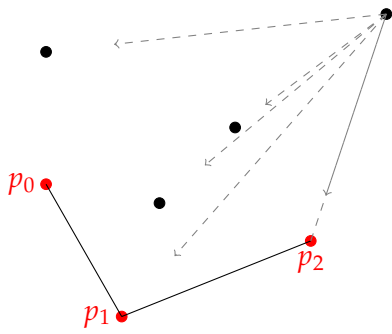


Figure 2.9

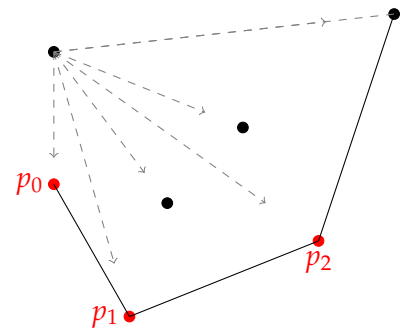


Figure 2.10

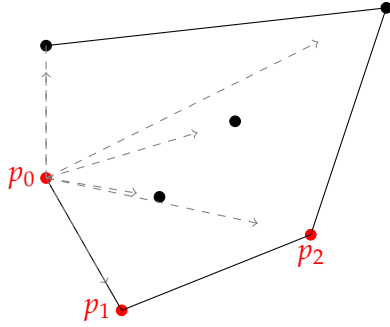


Figure 2.11

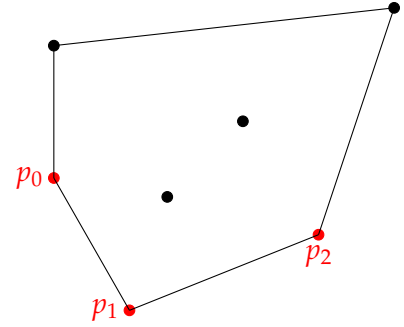


Figure 2.12

Jarvis March in Isabelle/HOL

This repeated finding of the next corner is defined as the function `wrap`, where q is the last minimum that was found and ps is the current set of points we want to find the convex hull of.

```
function wrap :: "point  $\Rightarrow$  point set  $\Rightarrow$  point list" where
  "wrap q ps =
    (if q = p0 then [] else q#(wrap (ccw'_seg_min q ps) (ps - {q})) )"
```

The last minimum q is prepended to the list of corners we will return, if we not yet arrived at the first corner p_0 again. The next corners are found by recursively calling `wrap` with the next corner or minimum `ccw'_seg_min q ps` and the set $ps - \{q\}$. q can be removed from the set of points we search for the next corner, as q can not be a corner of the polygon again. Lastly the algorithm Jarvis March is defined by an initial call to `wrap`, but p_0 is this time not removed from the set ps we search for the next corner, because p_0 is the only corner we can and must encounter twice.

```
definition "jarvis_march = to_set (wrap (ccw'_seg_min p0 ps) ps)"
```

The `to_set` function just turns the list of corners into the appropriate definition of the set of points that lie inside the polygon (see 2.1.4).

```
fun to_set :: "point list  $\Rightarrow$  point set" where
  o_set [] = {p0} |
  o_set [p] = closed_segment p0 p |
  o_set qs = {p. list_all (encompasses p) (polychain_of (p0#qs@[p0]))}"
```

The special cases of `wrap` returning an empty list or an list with only one element need more explanation. If $(wrap (ccw'_seg_min p_0 ps) ps) = []$, then we know $ccw'_seg_min p_0 ps = p_0$ has to hold and therefore $\forall r \in ps. ccw'_seg p_0 p_0 r$. Intuitively it should be clear, that the only point r that fulfills $ccw'_seg p_0 p_0 r$ is p_0 .

itself and therefore ps has to only contain $p0$ and the convex hull of a single point is a set containing this very point. If $(\text{wrap } (\text{ccw}'_seg_min \ p0 \ ps) \ ps) = [p]$, then we know $\forall r \in ps. \text{ccw}'_seg \ p0 \ p \ r$ and $\forall r \in ps. \text{ccw}'_seg \ p \ p0 \ r$. Again from geometric intuition it should be clear that $\forall r \in ps. r \in \text{closed_segment } p0 \ p$ should hold, as $\text{ccw}' \ p0 \ p \ r$ or $\text{ccw}' \ p \ p0 \ r$ instantly leads to a contradiction. The last case of the to_set function just applies the definition for the set of points inside inside a convex polygon, as introduced in 2.1.4.

2.2.2 Jarvis March is Convex Hull

In the following let $ps \subseteq \mathbb{R}^2$ be the finite set of points of which we want to calculate the convex hull and let $p0 = \min_y \min_x ps$ be the lexicographical minimum with which we start Jarvis March, i.e. our first corner of the convex polygon. In Isabelle terms, we assume $\forall p \in ps. \text{lex } p0 \ p$, $p0 \in ps$ and $\text{finite } ps$. First we need to show that the recursive wrap function terminates.

```
lemma wrap_dom:
  assumes  $q \in qs \wedge p0 \in qs$ 
  assumes " $qs \subseteq ps$ "
  assumes " $q = p0 \vee (\forall q' \in qs. \text{ccw}'\_seg \ p\_stl \ q \ q')$ "
  shows "wrap_dom (q,qs)"
```

This lemma follows from the step by step description of 2.2.1. In every step our last minimum q was either equal to $p0$ (in the beginning) which fulfills $\forall r \in ps. \text{lex } p0 \ r$ or our last minimum fulfilled $\forall r \in ps. \text{ccw}'_seg \ p \ q \ r$ (found with wrap) for some p . In both cases $(\text{ccw}'_seg \ q)$ is a total order and a new minimum q_{next} such that $\forall r \in ps. \text{ccw}'_seg \ q \ q_{next} \ r$ holds, exists (see 2.1.3). So $\text{ccw}'_seg_min \ q \ qs$ and therefore every recursive call to wrap is well-defined. Additionally the size of the set with which wrap is recursively called decreases in every iteration. Hence the call $(\text{wrap } (\text{ccw}'_seg_min \ p0 \ ps) \ ps)$ will terminate. Now we need to show that the list that $(\text{wrap } (\text{ccw}'_seg_min \ p0 \ ps) \ ps)$ returns represents a correct convex polygon.

```
lemma wrap_sorted:
  shows "sorted_wrt (ccw' p0) (wrap (ccw'_seg_min p0 ps) ps)"
```

```
lemma wrap_turns_left:
  shows "turns_only_left (wrap (ccw'_seg_min p0 ps) ps)"
```

We will start with the proof of $\text{sorted_wrt } (\text{ccw}' \ p0) \ (\text{wrap } (\text{ccw}'_seg_min \ p0 \ ps) \ ps)$. To do this, we first show that the inner call $\text{wrap } (\text{ccw}'_seg_min \ q \ qs) \ (qs - \{q\})$, where we assume that $\forall r \in qs. \text{ccw}'_seg \ p \ q \ r$ holds for the last minimum q and some p , produces a list that is $\text{sorted_wrt } (\text{ccw}' \ p0)$.

```

lemma wrap_sorted_ind:
  assumes wrap (ccw'_seg_min q qs) (qs - {q}) = ls
  assumes q ∈ qs ∧ p0 ∈ qs
  assumes qs ⊆ ps
  assumes (∀r ∈ qs. ccw'_seg p q r) ∧ (p0 ≠ q)
  shows sorted_wrt (ccw' p0) ls

```

The proof works by induction over the list ls . In the inductive case, we assume $ls = a \# b \# rs$ and that the induction hypothesis holds for $b \# rs$. Meaning we want to show $sorted_wrt (ccw' p0) a \# b \# rs$ and assume that $sorted_wrt (ccw' p0) b \# rs$ already holds additional to the other assumptions like $q \in qs$ and $\forall r \in qs. ccw'_seg p q r$. As the $sorted_wrt (ccw' p0)$ predicate only makes sense for list of at least length two, we can assume $ls = a \# b \# rs$ in the inductive case. Due to the properties of $wrap$, a and b are minima defined by the ccw'_seg_min function, for example $a = (ccw'_seg_min q qs)$ has to hold. From this, one can show that $\forall r \in (qs - \{a, b\}). ccw'_seg a b r$ and $a \neq p0 \wedge b \neq p0$ holds. From $a \neq p0 \wedge b \neq p0$ and $p0 \in qs$, we know that $p0 \in (qs - \{a, b\})$ holds and with that $ccw'_seg a b p0$ has to hold. But because we assumed $p0$ to be the lexicographical minimum of ps and $a \in ps \wedge b \in ps$, we know $lex p0 a \wedge lex p0 b$ has to hold. From geometric intuition it should be clear, that if $lex p0 a \wedge lex p0 b$ and $a \neq p0 \wedge b \neq p0$, it follows that $p0 \in closed_segment a b$ can not hold. Hence with $ccw'_seg a b p0$ we know, that $ccw' a b p0$ and therefore $ccw' p0 a b$ has to hold. Using the induction hypothesis $sorted_wrt (ccw' p0) b \# rs$, we know that $\forall r \in set rs. ccw' p0 b r$ holds. From $\forall r \in ps. lex p0 r$, we know that $(ccw' p0)$ is a total order on ps and therefore also transitive on ps . Each element in the list rs is indirectly picked from ps , which implies $(set rs) \subseteq ps$. So in the end, from $\forall r \in set rs. ccw' p0 b r$ and $ccw' p0 a b$ follows with transitivity $\forall r \in set (b \# rs). ccw' p0 a r$. Again together with the induction hypothesis $sorted_wrt (ccw' p0) a \# b \# rs$ follows, which is what we wanted to show. The final lemma for $sorted_wrt (ccw' p0) (wrap (ccw'_seg_min p0 ps) ps)$ works very similar, but a slightly different approach is needed, because in the first step the old minimum $p0$ is not removed from ps for the call to $wrap$.

Now we want to show the second part $turns_only_left (wrap (ccw'_seg_min p0 ps) ps)$. Again we start with the inner call $wrap (ccw'_seg_min q qs) (qs - \{q\})$.

```

lemma wrap_turns_left_ind:
  assumes "wrap (ccw'_seg_min q qs) (qs - {q}) = ls"
  assumes q ∈ qs ∧ p0 ∈ qs
  assumes qs ⊆ ps
  assumes (∀r ∈ qs. ccw'_seg p q r) ∧ (p0 ≠ q)
  shows " turns_only_left ls"

```

Similar to before in the inductive case, we assume $ls = k \# q \# p \# rs$ and can show using the other assumptions and the induction hypothesis that $turns_only_left \ q \# p \# rs$ already holds. Now we want to show $turns_only_left \ k \# q \# p \# rs$, which in this case only requires to show $ccw' \ k \ q \ p$, because of the induction hypothesis. Again the points $k \ q$ and p are defined by ccw_seg_min and we can show that $\forall r \in (qs - \{k, q\})$. $ccw_seg \ k \ q \ r$ and $ccw_seg \ k \ q \ p$ holds. Now if $p \in closed_segment \ k \ q$ would hold, then k, q and p would be colinear. Also from the previously shown $wrap_sorted_ind$ we know, that $ccw' \ p0 \ k \ q$ and $ccw' \ p0 \ q \ p$ hold. From this and the fact that $k \ q$ and p are colinear it would follow that $q \in closed_segment \ k \ p$, which should be apparent from an geometric viewpoint. But if $p \in closed_segment \ k \ q$ and $q \in closed_segment \ k \ p$ holds, then it can be shown that $p = q$ has to hold, which is a contradiction to for example $ccw' \ p0 \ q \ p$. Therefore $p \in closed_segment \ k \ q$ can not hold and with $ccw_seg \ k \ q \ p$, it follows that $ccw' \ k \ q \ p$ has to hold, which shows the lemma. So $(wrap \ (ccw_seg_min \ p0 \ ps) \ ps)$ does indeed produce a convex polygon. Finally we can show the lemma `jarvis_eq_convex_hull`.

`lemma jarvis_eq_convex_hull:`

`"jarvis_march p0 ps = convex_hull ps"`

Applying the definition of the `jarvis_march` function, we have to show

`to_set (wrap (ccw'_seg_min p0 ps) ps) = convex_hull ps`. Why this equality holds if `wrap (ccw'_seg_min p0 ps) ps` returns an empty list or a list with one element was already explained when the `to_set` function was defined (see 2.2.1). If the list $ls = wrap \ (ccw_seg_min \ p0 \ ps) \ ps$ contains at least two points, we know from `wrap_sorted` and `wrap_turns_left`, that ls correctly represents a convex polygon. As ls represents a correct convex polygon and contains at least two points, we know `jarvis_march p0 ps = to_set ls` is the set of points inside the corresponding polygon.

`to_set ls = {p. list_all (encompasses p) polychain_of (p0#ls@[p0])}`

With `wrap_sorted`, `wrap_turns_left` and `polygon_eq_convex_hull`, we then know that the set of points inside the polygon is equal to the convex hull of the corners.

`{p. list_all (encompasses p) polychain_of (p0#ls@[p0])} = convex_hull (set p0#ls)`

This is almost what we wanted to show. All points in ls are points from ps , they are precisely the points that were identified as corners of the convex polygon that is the convex hull of ps . When we start with a call `(wrap (ccw'_seg_min p0 ps) ps)` to the `wrap` function, we know that in every recursive call to `wrap`, one point will be removed from ps until $p0$ is encountered and ls is returned. The points that are removed from ps throughout the recursion are exactly the points in ls . With that we also know, that the points $ps - (set \ ls)$ are in the set that is considered for the next

corner in every recursive step of $(\text{wrap } (\text{ccw}'_seg_min \text{ p0 ps}) \text{ ps})$. This implies that for every r in $ps - (\text{set } ls)$ and every two consecutive corners p, q in ls it holds that $\text{ccw}'_seg \text{ p q r}$. This is because q is found as minimum with respect to $(\text{ccw}'_seg \text{ p})$ and r has to be in the set in which we search the minimum. Therefore this r lies inside the polygon defined by ls , as $\text{ccw}'_seg \text{ p q r} \implies \det3 \text{ p q r} \geq 0$ and therefore we know that r lies counterclockwise (or colinear) of every edge of the polygon. The edges of the polygon are exactly the elements of the list $\text{polychain_of } (\text{p0\#ls@[p0]})$. Meaning we know $\text{list_all } (\text{encompasses } r) \text{ polychain_of } (\text{p0\#ls@[p0]})$ holds and the inside of the polygon is equal to the convex hull of the corners, hence we also know $r \in \text{convex hull } (\text{set } \text{p0\#ls})$. Finally, we know $ps - \text{set } (ls) \subseteq \text{convex hull } (\text{set } \text{p0\#ls})$ and $\text{set } (ls) \subseteq \text{convex hull } (\text{set } \text{p0\#ls})$, which implies $\text{convex hull } (\text{set } \text{p0\#ls}) = \text{convex hull } ps$. Using this equality and the previously obtained $\text{jarvis_march } \text{p0 ps} = \text{convex hull } (\text{set } \text{p0\#ls})$, we get the final assertion.

2.2.3 Computability

In , the lemma wrap_dom used, that if we call wrap with a valid last minimum like $\text{p0} = \min_y \min_x ps$ in $(\text{wrap } (\text{ccw}'_seg_min \text{ p0 ps}) \text{ ps})$, then in every recursive step with the last minimum being $p \in qs$, $(\text{ccw}'_seg \text{ p})$ is a total order and transitive on the current set $qs \subseteq ps$. From the fact that $(\text{ccw}'_seg \text{ p})$ is transitive it should be clear, that the minimum can be found by looking at every element in qs once, which takes $O(|qs|) \leq O(|ps|) = O(n)$, if n is the number of points in the input set for Jarvis March. Comparing two points according to the $(\text{ccw}'_seg \text{ p})$ predicate can be achieved with the following computable function, which can be shown to be equivalent to the ccw'_seg predicate.

```
definition ccw'_seg_fun :: "point  $\Rightarrow$  point  $\Rightarrow$  point  $\Rightarrow$  bool" where
  "ccw'_seg_fun p q r =
    ( $\det3 \text{ p q r} > 0 \vee (\det3 \text{ p q r} = 0 \wedge \text{dist } \text{p r} \leq \text{dist } \text{p q})$ )"
```

```
lemma ccw'_seg_fun_iff_ccw'_seg:
  assumes  $p \in ps \wedge q \in ps$ 
  shows  $\text{ccw}'\_seg\_fun \text{ p\_last } \text{p q} \longleftrightarrow \text{ccw}'\_seg \text{ p\_last } \text{p q}$ 
```

p and q have to be in some ps , which fulfills $\forall r \in ps. \text{ccw}'_seg \text{ p_stl } \text{p_last } r \vee \forall q \in ps. \text{lex } \text{p_last } r$ for everything to be well-defined. Computing the determinant and distance between two points (dist) only takes a few arithmetic operations, therefore comparing two points can be seen as an operation that takes $O(1)$. To sum up, given there are h corners in the convex polygon and n points in the input set ps , we have to find h times the minimum with respect to a total order which takes $O(n)$ steps and

therefore we get a runtime of $O(h \cdot n)$. The algorithm is simpler than the Graham Scan or the Chan's algorithm and has a worse runtime than both unless h is small. Graham Scan achieves a $O(n \cdot \log(n))$ runtime and Chan's algorithm a $O(n \cdot \log(h))$ runtime. If h is small Jarvis March can be faster than Graham Scan.

2.3 Graham Scan

In Simon Hanssen's Bachelor Thesis, it was already shown that the Graham Scan Algorithm calculates the convex hull of a input set of points. However the Graham Scan Algorithm works by first sorting the points and then calculating the convex hull. So far it was only shown that given a list of sorted points ps , the second phase of the algorithm correctly calculates the convex hull. Assuming the points of which we want to calculate the convex hull are given as a list ps and $p0 = \min_y \min_x (\text{set } ps)$ is the lexicographical minimum, then the proof assumes $\text{sorted_wrt } (ccw' p0) ps$. Therefore we want to implement the sorting phase and show that it produces a list ps , which is $\text{sorted_wrt } (ccw' p0) ps$. To avoid duplication, an existing framework for sorting elements according to an order should be used. The Comparator theory provides a definition for comparing two elements of an arbitrary type.

```

locale comparator =
  fixes cmp :: "'a  $\Rightarrow$  'a  $\Rightarrow$  comp"
  assumes refl: " $\forall$  a. cmp a a = Equiv"
    and trans_equiv: " $\forall$  a b c. cmp a b = Equiv  $\implies$ 
      cmp b c = Equiv  $\implies$  cmp a c = Equiv"
  assumes trans_less: "cmp a b = Less  $\implies$  cmp b c = Less  $\implies$  cmp a c = Less"
    and greater_iff_sym_less: " $\forall$  b a. cmp b a = Greater  $\longleftrightarrow$  cmp a b = Less"

datatype comp = Less | Equiv | Greater

```

The operation `cmp` that compares two values of type `'a` is defined to be of the type `'a comparator`.

```

typedef 'a comparator = "{cmp :: 'a  $\Rightarrow$  'a  $\Rightarrow$  comp. comparator cmp}"

```

Lastly `sort` is the already implemented sorting that uses a Comparator, takes in an arbitrary list and produce a sorted list.

```

definition sort :: "'a comparator  $\Rightarrow$  'a list  $\Rightarrow$  'a list"

```

```

lemma sorted_sort :
  "sorted cmp (sort cmp xs)"

```

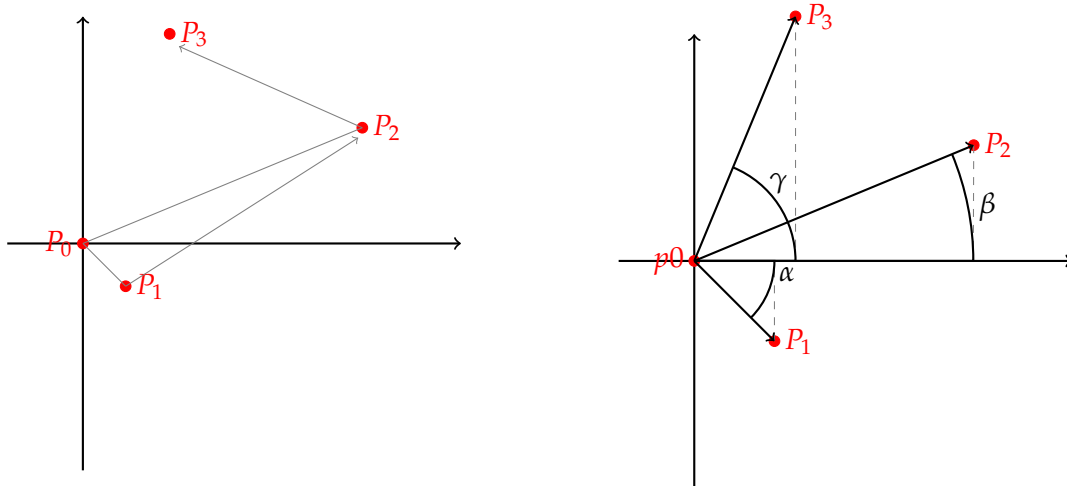

The Comparator framework uses the sorted predicate to express that a list is sorted, therefore we need to translate from the sorted predicate to the sorted_wrt predicate.

```
lemma sorted_wrt_if_sorted: "sorted cmp ls  $\Rightarrow$ 
sorted_wrt ( $\lambda$  a b.(compare cmp a b = Equiv)  $\vee$  (compare cmp a b = Less)) ls"
```

The compare function just applies a Comparator cmp to two elements a and b . From the fact that the order $(ccw' \ p0)$ is in general not transitive (see 2.1.3), it should become clear that an intuitive definition like $ccw' _comparator$ will not work.

```
lift_definition ccw'_comparator :: "point comparator"
is " $\lambda$  p q. if ccw' p0 p q then Less else if ccw' p0 q p then Greater
      else Equiv"
```

For arbitrary p, q and r , $ccw' \ p0 \ p \ q \wedge ccw' \ p0 \ q \ r \implies ccw' \ p0 \ p \ r$ does not always hold. If, for example $\text{lex } p, \text{lex } q$ and $\text{lex } r$ is the case, then the implication does hold. A possible solution to this problem uses a different viewpoint on what $\text{sorted_wrt } (ccw' \ p0) \ ps$ actually means. Instead of interpreting ps as being sorted according to $(ccw' \ p0)$, we can interpret ps as being sorted with respect to the angle the points in ps form with the x-axis, that goes through $p0$.



In Figure x, you can see the lexicographical minimum $p0$ and a list of points $ps = [P_1, P_2, P_3]$ in a coordinate system. Notice that $p0$ is the lexicographical minimum $p0 = \min_y \min_x (\text{set } ps \cup \{p0\})$, where we first minimize x and then y , therefore all points in ps have greater or equal x -values than $p0$ and only if they have the same x -value as $p0$, their y -value is greater or equal than the y -value of $p0$. Clearly the list $ps = [P_1, P_2, P_3]$ fulfills $\text{sorted_wrt } (ccw' \ p0) \ ps$, as $ccw' \ p0 \ P_1 \ P_2$ and $ccw' \ p0 \ P_2 \ P_3$ holds. But ps is also sorted with respect to the angles that the points form with x' , the line parallel to the x -axis, that goes through $p0$, as $\alpha = -\frac{\pi}{4}$, $\beta = \frac{\pi}{8}$,

$\gamma = \frac{3\pi}{8}$ and $\alpha \leq \beta \leq \gamma$ holds. It would also be possible to take the angles, that the points form with y' , the line parallel to the y-axis that goes through p_0 . In this case, we would have angles from 0 to π , instead of angles from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$. For our purpose taking the angles with respect to x' will be simpler. In the following, we are first going to calculate the angles, that the points in ps form with x' , then we are going to sort the points in ascending order according to their angles and lastly we have to show that this order is always going to be the same as if they were ordered according to $(ccw' p_0)$. The function `angle` calculates the angle formed with x' , where $p_0 = (x_0, y_0)$ and (x_1, y_1) is going to be a point in ps . To be more precise, the function calculates the angle $\angle (x_1, y_1)(x_0, y_0)(x_0, y_1)$.

```
fun calc_angle1 :: "point  $\Rightarrow$  point  $\Rightarrow$  real" where
"calc_angle1 (x0,y0) (x1,y1) = (if x1 = x0  $\wedge$  y1 = y0 then -pi/2 else
    if x1 = x0 then pi/2 else arctan ((y1 - y0)/(x1 - x0))) "
```

Arcussinus or Arcuscosinus could be used to calculate the angles as well, but we choose Arcustanges, as it translates to the $(ccw' p_0)$ ordering more easily. Essentially we are calculating the Arcustanges of the slope of the vector $\overrightarrow{(x_0, y_0)(x_1, y_1)}$. However the case $x_1 = x_0$ would be undefined, as $(x_1 - x_0) = 0$, therefore we have to handle this case differently. As already explained, if a point p_1 has the same x-value as p_0 , we know that the y-value of p_1 has to be greater or equal than the y-value of p_0 . Therefore, assuming $lex p_0 p_1$, we know that $y_1 \geq y_0$ has to hold, if $x_1 = x_0$. Therefore it makes sense to assign the angle $\frac{\pi}{2}$, if $x_1 = x_0$ and $lex p_0 p_1$, which implies $y_1 \geq y_0$, holds. The case where $x_1 = x_0 \wedge y_1 = y_0$ will be explained later. Now we can prove that if two points p_1 and p_2 have the same angle with respect to p_0 , then p_0, p_1 and p_2 are colinear.

```
lemma angle_to_det0:
assumes "calc_angle1 (x0,y0) (x1,y1) = calc_angle1 (x0,y0) (x2,y2)"
assumes "x1  $\leq$  x0  $\wedge$  x2  $\leq$  x0"
shows "det3 (x0,y0) (x1,y1) (x2,y2) = 0"
```

The only interesting case is that $x_1 \neq x_0 \wedge x_2 \neq x_0$ and `calc_angle1 (x0,y0) (x1,y1) = calc_angle1 (x0,y0) (x2,y2)` holds, because `arctan ((y1 - y0)/(x1 - x0)) = arctan ((y2 - y0)/(x2 - x0))` holds. Then because of the injectivity of Arcustanges, we know $\frac{y_1 - y_0}{x_1 - x_0} = \frac{y_2 - y_0}{x_2 - x_0}$ holds. From this we get $(y_1 - y_0) \cdot (x_2 - x_0) = (y_2 - y_0) \cdot (x_1 - x_0)$ and using the `det_form` lemma from 2.1.2, we get `det3 (x0,y0) (x1,y1) (x2,y2) = 0`.

Next we show the central lemma stating that if (x_2, y_2) forms a greater angle with x' than (x_1, y_1) does, we know that $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ are oriented counterclockwise.

```

lemma angle_to_det:
  assumes "calc_angle1 (x0,y0) (x1,y1) < calc_angle1 (x0,y0) (x2,y2)"
  assumes "lex (x0,y0) (x1,y1) ∧ lex (x0,y0) (x2,y2) ∧
           (x0,y0) ≠ (x1,y1) ∧ (x0,y0) ≠ (x2,y2)"
  shows " det3 (x0,y0) (x1,y1) (x2,y2) > 0"

```

Again for now we ignore the case where both arguments of `angle` are the same point and `angle` would evaluate to $-\frac{\pi}{2}$. Then the only interesting case is where $x1 \neq x0 \wedge x2 \neq x0$ holds and `angle` on both sides evaluates to `arctan`, meaning $\arctan((y1-y0)/(x1-x0)) < \arctan((y2-y0)/(x2-x0))$ holds. Then due to Arcus-tanges being strictly monotonically increasing, we know $\frac{y1-y0}{x1-x0} < \frac{y2-y0}{x2-x0}$. From $x1 \neq x0 \wedge x2 \neq x0$, we know that both sides are well-defined and from $\text{lex}(x0,y0)(x1,y1) \wedge \text{lex}(x0,y0)(x2,y2)$, we know $x1 \geq x0$ and $x2 \geq x0$ holds. Therefore, we can reformulate $\frac{y1-y0}{x1-x0} < \frac{y2-y0}{x2-x0}$ to $(y1-y0) \cdot (x2-x0) < (y2-y0) \cdot (x1-x0)$, which again using the `det_form` lemma from 2.1.2 implies that $\text{det3}(x0,y0)(x1,y1)(x2,y2) > 0$ holds.

Using the `arctan` function in `angle` is not necessary, it would be fine to just directly map a point p to the slope of $\overrightarrow{p0 p}$ instead of `arctan` of the slope. Using `arctan` and comparing angles instead of slopes is still preferable, as then `angle` is a bounded function. If we would work with slopes directly, a case where $x1 = x0$ would have to be mapped to ∞ or a different special value and a different datatype than `real` might be necessary. Comparing angles is easier for proofs and `arctan` guarantees useful properties like boundedness.

2.3.1 Colinearity

So far, we ignored a non-trivial problem. The lemma which shows that the Graham Scan algorithm calculates the convex hull assumes that the sorting phase produced a list ps , which is `sorted_wrt (ccw' p0) ps`. Therefore ps cannot contain any $p, q \in (\text{set } ps)$ with $p \neq q$ such that $p0, p$ and q are colinear. Now we cannot assume this for an arbitrary input list or set of points. Therefore we will show that our implementation of the sorting phase produces a list ps that is `sorted_wrt (ccw'_seg_rev p0) ps`, where `ccw'_seg_rev` is defined similar to `ccw'_seg`.

```

definition "ccw'_seg_rev p q r = ccw' p q r ∨ q ∈ closed_segment p r"

```

The fact that `sorted_wrt (ccw'_seg_rev) ps` holds, is just the formal way of expressing that if two points p, q cannot be ordered according to `(ccw' p0)`, because neither `ccw' p0 p q` nor `ccw' p0 q p` holds, they are ordered according to `(dist p0)`. Therefore if $\text{dist } p0 p \leq \text{dist } p0 q$, then p will precede q in the list ps . Again this is very similar to `ccw'_seg`, just that q would precede p if $\text{dist } p0 p \leq \text{dist } p0 q$ holds and

ps is sorted according to ccw'_{seg} . As we cannot ensure that the second phase of Graham Scan gets a ps with $sorted_wrt (ccw' p0) ps$, we want the sorting phase to produce a ps , which is sorted according to a weaker relation that still ensures that the second phase correctly computes the convex hull. For Graham Scan to still produce a correct convex hull, ccw'_{seg_rev} should be used.

2.3.2 Angle Comparator

We have already seen from `angle_to_det` and `angle_to_det0` that comparing the angles of points with `angle` translates to comparing points with `ccw'`. Now we want to get a comparator `angle_comparator`, such that the already implemented sort according to this comparator produces a ps that fulfills $sorted_wrt (ccw'_{seg_rev} p0) ps$.

```
lemma ccw'_seg_rev_if_sorted_w_angle_comp:
shows "sorted_wrt (ccw'_seg_rev p0) (sort angle_comparator ps)"
```

This lemma essentially uses three lemmas. First we use the previously explained `sorted_wrt_if_sorted` lemma to translate from sorted to `sorted_wrt`. Second we need to prove the `weaker_rel` lemma for our specific `angle_comparator` and use the `sorted_wrt_mono_rel` lemma to translate from the `angle_comparator` to the $(ccw'_{seg_rev} p0)$ predicate.

```
lemma weaker_rel:
assumes ins: "x ∈ set ps ∧ y ∈ set ps"
assumes rel: "(compare angle_comparator x y = Equiv)
  ∨ (compare angle_comparator x y = Less)"
shows "ccw'_seg_rev p0 x y"
```

```
lemma sorted_wrt_mono_rel:
"((∧ x y. [ x ∈ set xs; y ∈ set xs; P x y ] ⇒ Q x y)
 ⇒ sorted_wrt P xs ⇒ sorted_wrt Q xs"
```

A first definition of the `angle_comparator` could look like this.

```
lift_definition angle_comparator :: "point comparator"
is "λ x y. if angle p0 x < angle p0 y then Less
  else if angle p0 x > angle p0 y then Greater else Equiv"
```

But we also need to deal with the $angle p0 x = angle p0 y$ case in more detail, we cannot declare these points as `Equiv`. As we have seen before, $angle p0 x = angle p0 y$ implies that $p0$, $p1$ and $p2$ are colinear and as explained in 2.3.1, we want to compare the distance to $p0$ in this case. Therefore the final `angle_comparator` will look like this.

definition "angle_comparator =
key ($\lambda p.(\text{calc_angle1 } p0 \ p \ , \text{dist } p0 \ p)$) lex_comparator"

key f cmp creates a comparator, that works as follows. The new Comparator will first apply f to the two elements to compare and then compare them with the Comparator cmp.

lift_definition key :: "('b \Rightarrow 'a) \Rightarrow 'a comparator \Rightarrow 'b comparator"

The lex_comparator translates the lex predicate into a comparator in the obvious way. Therefore the angle_comparator takes two points p, q and first maps them to tuples $(\text{angle } p0 \ p, \text{dist } p0 \ p), (\text{angle } p0 \ q, \text{dist } p0 \ q)$ and then compares them-lexicographically, meaning first the angles are compared and if they are the same, the distance to $p0$ is compared and if the distances are the same, the points are declared as equivalent. Now with this definition of angle_comparator it is possible to show the weaker_rel lemma and with that the final lemma ccw'_seg_rev_if_sorted_w_angle_comp can be shown.

2.4 Chans Algorithm

Citation test [Lam94].

Acronyms must be added in main.tex and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. $\backslash\text{ac}\{\text{TUM}\}$, $\backslash\text{ac}\{\text{TUM}\} \Rightarrow \text{TUM}$, TUM

For more details, see the documentation of the acronym package¹.

2.4.1 Subsection

See Table 3.1, Figure 3.1, Figure 3.2, ??.

Table 2.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

!TeX root = ../main.tex

¹<https://ctan.org/pkg/acronym>

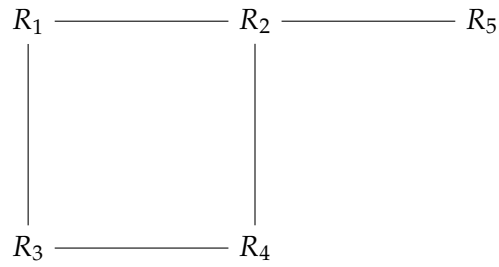


Figure 2.13: An example for a simple drawing.

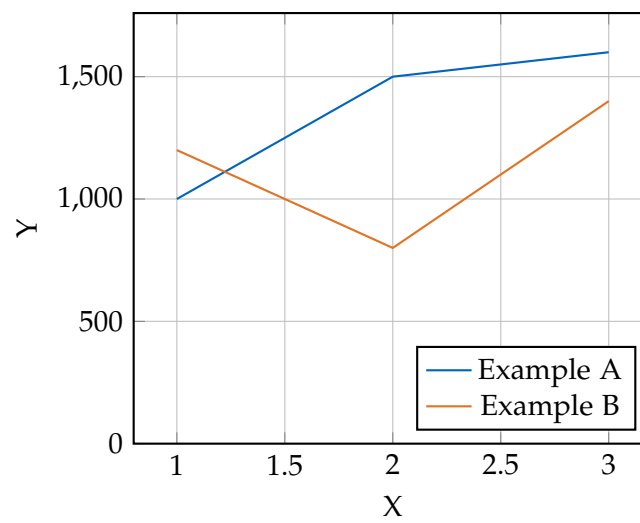


Figure 2.14: An example for a simple plot.

3 Definitions and Algorithms

3.1 Convex Hull

First the Convex Hull will be defined. A set $S \subseteq \mathbb{R}^2$ is convex if for every two points p and q in S it holds that all points on the line segment connecting p and q are in S again. This can be expressed, as the fact that the any convex combination of p and q has to be in S again, i.e. $\{x | \exists u, v \geq 0. p * u + q * v = x\} \subseteq S$ has to hold. The convex hull of a set S is the smallest convex set in which S is contained. There are several ways in which the convex hull can be defined. The convex hull CH of S is the intersection of all convex sets containing S , which is also the definition Isabelle/HOL is going to use. But the convex hull can also be defined as the set of all convex combinations of points in S , which can be proven equivalent to the previous definition. In the two dimensional case for a finite $S \subset \mathbb{R}^2$, the convex hull CH of S is a convex polygon, where the corners of this convex polygon are points from S (see figure 1). [De 00] An edge connecting two points $(p, q) \in S^2$ is an edge of the convex polygon iff. all points lie to the left of the line \overline{pq} connecting p and q . Similarly the set of all edges of the convex polygon can be defined as all $(p, q) \in S^2$ for which all points in S lie to the right of \overline{pq} . As this thesis will focus on the two dimensional case and only give an outlook on the the three dimensional case, the examined algorithms compute a convex polygon for a given $S \subset \mathbb{R}^2$.

3.2 Jarvis-March Algorithm

The Jarvis March or Gift-Wrapping Algorithm is a simple output-sensitive way of calculating the convex hull of a given finite set $S \subseteq \mathbb{R}^2$ of points. It calculates the convex hull by calculating the corresponding convex polygon and returning an ordered list of the corners of the polygon. The algorithm has runtime $O(n * h)$, where n is the number of points in S and h is the number of points that lie on the convex hull or the number of corners on the calculated polygon to be more precise. First we will assume that no three points in S are colinear. The algorithm starts by choosing a point that is guaranteed to lie on the convex hull, for example a $p_0 = \min_y \min_x S$. Then the next corner of the convex polygon is found by searching a p_1 such that all points in S lie

to the left of the line $\overline{p_0 p_1}$. As explained in 3.1 we know that (p_0, p_1) is an edge of the wanted convex polygon and we know that q is once again a point on the convex hull, i.e. a corner of the polygon. Therefore we can repeat the previous step and search for a p_2 such that all points in S lie left to the line $\overline{p_1 p_2}$. Again p_2 has to be a corner of the convex polygon and (p_1, p_2) an edge on of the polygon. The algorithm continues until a $p_h = p_0$ is found to be the next point and stops, because the first corner of the polygon is encountered again. The ordered sequence of points p_0, p_1, \dots, p_{h-1} are the corners of the convex polygon and $(p_0, p_1), (p_1, p_2), \dots, (p_{h-2}, p_{h-1}), (p_{h-1}, p_0)$ are the edges of the polygon. Now without the assumption that no three points are colinear, we require more rigorous definitions. Given a p_i that is a corner of the convex polygon the next corner p_{i+1} has to fulfill the following condition for all $q \in S$. Either q lies strictly left of $\overline{p_i p_{i+1}}$ (p_i, p_{i+1} and q are not colinear) or q is contained in the closed segment between p_i and p_{i+1} . In the following a point q lying strictly left of a line $\overline{p_i p_{i+1}}$ will be expressed as q lying counterclockwise of the line $\overline{p_i p_{i+1}}$. This clarification avoids, that points which are not a corner but still lie on the convex hull are ignored (see figure 2). The algorithm is simpler than the Graham Scan or the Chan's algorithm and has a worse runtime than both unless h is small. Graham Scan achieves a $O(n \log(n))$ runtime and Chan's algorithm a $O(n \log(h))$ runtime. If h is small Jarvis March can be faster than Graham Scan.

`lemma turns_only_right st \implies
turns_only_right (grahamsmarch qs st)`

3.3 Graham Scan

3.4 Chans Algorithm

Citation test [Lam94].

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}`, `\ac{TUM}` \Rightarrow TUM, TUM

For more details, see the documentation of the acronym package¹.

3.4.1 Subsection

See Table 3.1, Figure 3.1, Figure 3.2, ??.

¹<https://ctan.org/pkg/acronym>

Table 3.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

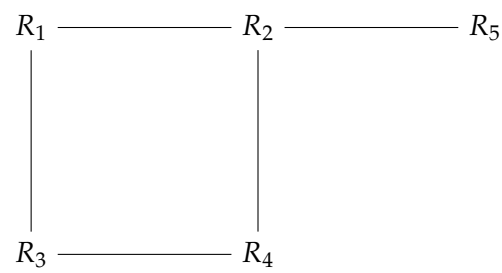


Figure 3.1: An example for a simple drawing.

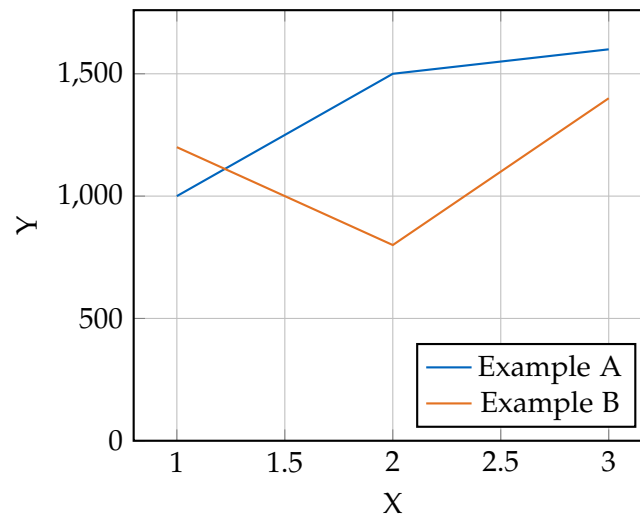


Figure 3.2: An example for a simple plot.

Abbreviations

TUM Technical University of Munich

List of Figures

1.1	Example drawing	1
1.2	Example plot	2
2.1	Non-convex set in 2D	4
2.2	Convex polygon, which is the convex hull of the set of points $\{P_0, P_1, P_2, P_3, P_4, P_5, P_6\}$	4
2.3	6
2.4	8
2.5	11
2.6	11
2.7	11
2.8	11
2.9	11
2.10	11
2.11	12
2.12	12
2.13	Example drawing	23
2.14	Example plot	23
3.1	Example drawing	26
3.2	Example plot	26

List of Tables

1.1	Example table	1
2.1	Example table	22
3.1	Example table	26

Bibliography

- [De 00] M. De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.
- [Lam94] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.