Safe smooth paths between straight line obstacles

Yves Bertot

Safe smooth paths between straight line obstacles

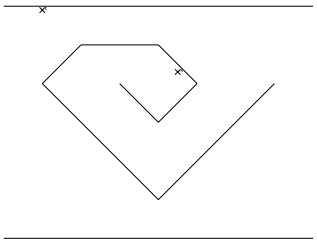
Yves Bertot

Inria Université Côte d'Azur

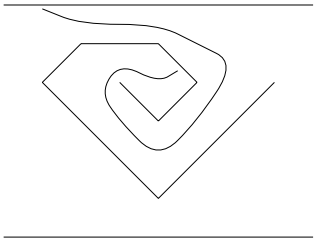
One benefit of formal verification is the removal of design errors at early stages of complex artifacts. This is being used extensively for software with great success. We wish to extend this to domains where the distance between formal models and the real application is bigger. We choose to work on robotics. For this field, there is a large distance between models and real implementations, because our knowledge of physics is incomplete, the world needs to be represented through geometrical abstractions, and there are limitations to what computers can compute accurately and what sensors and actuators can do in terms of precision.

To begin with we will look only at a question involving geometry. We wish to present a comprehensive Coq program to compute trajectories for a point between obstacles that are given by straight line segments.

The program we wish to describe formally receives as inputs descriptions of problems like this drawing, where the bottom and top straight lines delimit the workspace, the other straight lines represent obstacles, and the cross marks are represent the extremities of the path:



and produces smooth trajectories like the curved line that appears in this picture:

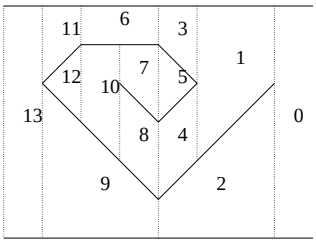


1 A combination of algorithms

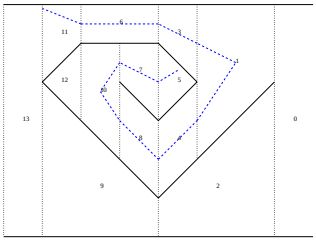
The Coq program contains four phases, where each corresponds to a different algorithm. For the first two phases, the basic blocks are concerned with converting the geometry of the problem

into a discrete problem, using sorting and a recursive treatment of the obtained sorted list. The result of the first two phases consists in a collection of cells, where each cell is a convex polygon connected to neighbors by safe boundaries that can be crossed without collision with the obstacles.

For our running example, the result of these first two phases is described in the following picture, where safe boundaries are shown as dashed lines and cell numbers have been added solely to help our understanding.



The problem becomes one of moving between cells. The collection of cells and safe boundaries is used to construct an abstract view of the trajectory, viewed simply as a path between the cross marks and cell boundaries. For this phase, a simple graph algorithm is used. The trajectory is built first as a broken line from the cross marks to safe boundaries of the cells that contain them, and then between safe boundaries. This broken line already represents a possible path, and we hope to prove that this path is safe thanks to the fact that cells are convex and the segments are each included in one cell at a time, except for their extremities which are members of a cell boundary or equal to one of the cross marks.



However, such a broken line is unsatisfactory as a robot trajectory, because each corner would impose that the robot stops to rotate and change direction. A last phase is added to find smooth curved lines in the vicinity of the broken line, and extra proofs are required to show that the smooth trajectories still avoid collisions with the obstacles. This phase can itself be decomposed in two parts, where one proposes a Bézier curve for testing, and the other checks that the given Bézier curve is safe. When he check fails, a curve that is closer to the initial broken line is proposed. In case where the repetitive process does not converge fast enough, it is possible to use the initial broken line as a fallback solution.

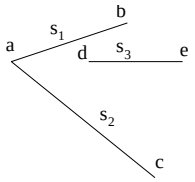
2 Producing a sequence of events

The first objective is to produce a collection of cells with the following properties:

1. The interior of each cell is free of obstacles
2. The parts of the boundaries of these cells that can be crossed safely to reach other cells are known precisely

To achieve this objective, we took inspiration from an algorithm known as *vertical cell decomposition* [7]. The intuitive operational model of this algorithm is that a vertical line is sweeping the working space from left to right. When the extremity of an obstacle segment or the intersection between two obstacle segments is encountered, some internal view of the cells is updated, ultimately producing a collection of closed cells. To represent this sweeping operations, we chose to implement a sorting algorithm that takes as input the list of segments and produces as output a list of *events*. Each event corresponds to an obstacle extremity and is annotated with the sequence of segments that have this event as leftmost extremity.

Here is an example with three segments and the corresponding list of events. For this example, we write $(a, [s_1; s_2])$ for an event whose location is givent by point a and whose sequences contains s_1 and s_2.



$$(a, [s_2; s_1]); (d, [s_3]); (b, []); (c, []); (e, [])$$

At this point in our explanation, note that we chose to simplify the problem by assuming that segments only intersect at their extremities. This assumption is not hard to satisfy, as it is possible to take a collection of segments which do not satisfy this assumption and produce a new collection of segments that contain exactly the same points of the plane but satisfy the assumption.

The sorting algorithm works as follows: every time an new segment is processed, two events are considered for addition in the sequence of events. The first event is for the leftmost extremity of the segment. If an event already exists at the same location, then the current segment is simply added to the annotation of the existing event. If not event exists with that location, the event is simply added, again with the current segment in the event annotation. The second event for the rightmost extremity of the segment is added similarly, except that nothing is done for event annotations.

For instance in our example, if segments s_1, s_2, and s_3 are processed in this order. The sequence of events after processing s_1 is

$$(a, [s_1]); (b, []).$$

After processing s_2, the sequence is

$$(a, [s_2; s_1]); (b, []); (c, []).$$

We see that in this case, no new event is added for the leftmost extremity of s_2, but s_2 is added to the sequence of segments annotating the event a location a.

Another simplification to our presentation of the problem is that vertical segments are not allowed in our input. Again, we think that it is possible to work around this limitation: If one wishes to add vertical obstacles, only there extremities should be added as events without outgoing edges in the sequence of events. We expect the cells to be created where a safe boundary appears between the two events. It is enough to remove this safe boundary from the cells to avoid collisions with the corresponding vertical obstacle. This approach is not implemented in our current version of the algorithm.

We assume we are working in some abstract ordered field to represent the coordinates of points. For the algorithm itself, we shall only use the field operations. For the proofs we need to use some properties of the order. The data types that we need to create to describe our algorithm are as follows:

- A datatype of points, which are ordered pairs of coordinates in the field,
- A datatype of edges (for the obstacles), which are ordered pairs of points, with the invariant that the first coordinate of the first point is strictly smaller than the first coordinate of the second point,
- A datatype of events, which have two components, the first is a point and the second is sequence of edges, all of which have this point as first extremity.

For this algorithm to be correct, we need to guarantee the following properties:

- All segments of the initial collection of obstacles must be present among the outgoing edges of events in the output
- All segments attached to a given event have this event as first extremity
- The sequence of events is strictly sorted lexicographically, as a consequence each edge appears as outgoing edge of a single event.

Processing the sorted list of events then implements the idea of the sweeping line from left to right. In our example, this means that the next phase of the algorithm will be processing events located at a, d, b, c, e in this order.

3 Producing cells

To understand how cells are produce, we can keep the mental model of a sweeping line, which is the vertical line passing through the location of current event. When producing cells, we distinguish between closed and open cells. The closed cells lie to the left of the sweeping line, they are complete and have well-defined left and right sides. The open cells intersect the sweeping line. They have a well-defined left side, but the right side is unknown, as it will be fixed when processing later events.

At a given position of the sweeping line, there is an ordered sequence of open cells. For each of these cells, the bottom and the top boundary are given by edges. Each of the open cells also has a left side, which was defined at the time the cell was created, possibly when processing several examples that all lie on the same vertical.

The processing of a new event works as follows:

1. The sequence of open cells is decomposed into three parts: a first sequence of untouched cells, a second sequence of cells in contact with the given event, and a third sequence of cells of untouched cells

2. All cells in contact are closed: their right side is described and they are added to the collection of closed cells

3. If the given event has n outgoing edges, then $n + 1$ new open cells are created. The outgoing edges are first sorted vertically. The first cell has for its low edge the same low edge as first cell in the first of the contact cells, and as high edge the first of the outgoing edges (after sorting). The last new cell is similar but with the last outgoing edge and the high edge of the last contact cell. All other newly created cells have a left side that is reduced to a single point, because the low and high edges meet at the event that is currently processed.

For this algorithm to work, we add an extra constraint on the data: the whole working space is enclosed between a bottom and a top edge that do not cross and are long enough so that they define a quadrangle with no contacts with all other edges.

Special care must be taken to handle the case where events are vertically aligned. When this happens, a naive version of our algorithm produces cells that have the left side and the right side on the same vertical line. These cells are flat and do not satisfy two important properties that we like about nonempty cells. First in a non-flat cell, any path from a safe point on the left side to a safe point on the right side is safe. Second, in a non-flat cell, any path to a safe point on any side to the center of the cell is safe.

To recover these properties, we modified our algorithm to memorize the position of the last processed event, together with the last opened cell and the last closed cell. When the current event is vertically aligned with the last processed event, we can avoid creating a flat closed cell by updating the last opened cell and the last closed cell. For last closed cell, only the obstacles on the right side need to be updated. However, this adds complexity in the proofs about the algorithm.

At the moment of writing these lines, this modified vertical cell decomposition algorithm is still being proved correct. We proceed by showing that a collection of properties are invariant through the processing of events. The properties that we have proved invariant so far are as follows:

- The sequence of open cells is sorted vertically, the cells are adjacent, and they cover the whole space between the bottom and top edges,

- All low and high edge of the current open cells contain a point that is vertically aligned with the currently processed event,

- All cells are disjoint,

- The obstacles are covered by the high edges of existing cells, closed or open and the outgoing edges of events that have not been processed yet.

The last two properties put together should be enough to guarantee that the interior of all closed cells is safe. We have yet to understand how we will express and prove safety properties for points on the left and right sides of closed cells.

For now, the left and right sides of closed edges are equipped with list of points, where the first one belongs to the cell's high edge and the last one belongs to the cells low edge. The

other points on these lists correspond to extremities of edges that end or start on this side. The points on the side of an edge that are distinct from the elements of these lists are also safe. The intervals between the points of these lists correspond to vertical segments that are safe boundaries of the cell. From now on, we will call these *doors* between the cell and neighbor cells. There is a safe horizontal passage between two cells if the left side of one cell and the right side of the other have such a door in common. We use this criterion to construct a graph whose nodes are the cells and whose edges are doors between one cell and the other.

4 Making a broken line trajectory between two points

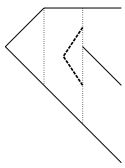
With the graph of cells and doors between these cells, we can already construct safe paths between two points in the space, which we shall the source and target points for clarity. We only need the source and target to be either inside the same polygon made of obstacles, or inside the work space and outside of any polygon.

The first step is to find the two cells that contain the source and the target. If the source and target are in the same cell, it is enough to draw a straight line segment from the source to the target.

If the two cells are different, we can explore the graph to check for the existence of a path in this graph from the source cell to the target cell. This path is composed of edges of the graph and each of the graph edges is actually a door between two cells. It is enough to construct a sequence of paths between the middle points of each door that is an element of this path. If two successive doors are not on the same vertical line, these two doors are on both sides of a same cell. A straight segment from the middle of the first door to the middle of the second door is a safe path in the interior of the cell (this property would not be satisfied if the cell was flat).

When two successive doors are on the same vertical line, a straight line between the center of one door and the center of the other is bound to be unsafe, because it will touch the lowest bound of the highest of the two doors, and this lowest bound is the extremity of some obstacle. To avoid this, we build a path composed of two segments. The first segment goes from the middle of the first door to the center of the cell. The second segment goes from the center of the cell to the middle of the second door. Such a two segment path is safe because the two extremities are safe, and the interior of these paths lie inside the interior of the cell, which is safe.

An example of safe path between two doors on the same side of a cell is given in the following figure:

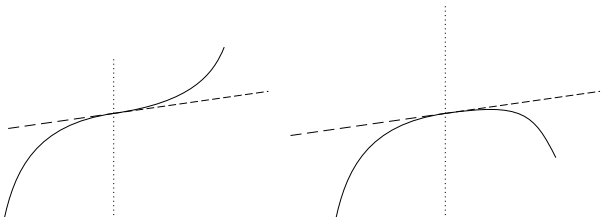


For the formal proofs of this part, we have not started working on this, but this should rely on the properties of convex sets, in particular that any segment between two points of a convex set is included in this convex set.

5 Making a smooth trajectory

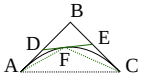
The trajectory described in the previous section is given by a sequence of segments, where each segment shares an extremity with its predecessor and the other extremity with its successor. We shall call these extremities the *corners* of the trajectory. We wish to replace each corner by a smooth curve and to connect each curve fragment in a smooth fashion, in other words, so that a point can travel the curve with continuous speed (in fact, we also wish to have continuous acceleration).

We expect that the smooth connection constraint can also be rephrased with the following sufficient criteria: the curve fragments are connected, their tangents at the connection are parallel, and these curves are locally on opposite sides of a straight line that crosses these tangents. This is illustrated in the following figures, each where two curves connect at the point where several straight lines cross. These straight lines are the two tangents of the curves at the point of intersection (two distinct dashed lines) and the line crossing the tangents that separates locally the two curves (dotted line). In the first figure the two connected curves are on both sides of the tangent. In the second figure, the two connected curves are on the same side of the tangent.



Such a property is easy to achieve with Bézier curves. In our case, it is enough to use quadratic Bézier curves. A quadratic Bézier curve is given by 3 points (in the same way that a polynomial of degree 2 is given by three coefficients). Given three points A, B, C a point on the corresponding Bézier curve is given by choosing a parameter t between 0 and 1 and computing three more points: D is the barycenter of A with weight $1-t$ and B with weight t, E is the barycenter of B with weight $1-t$ and C with weight t, and F is the barycenter of D with weight $1-t$ and E with weight t. The point F is the point on the Bézier curve for the

parameter t. When t varies between 0 and 1, the point moves on the Bézier curve from A to C, generally without passing through B. The curve is tangent to the segment (A, B) in A and tangent to the segment (B, C) in C.



A broken line $A\ B\ C\ D$ can be transformed into a sequence of smoothly connected Bézier curves by adding a point M on the segment between B and C and replacing the three segments (A, B), (B, C), and (C, D) by the two Bézier curves given by points $A\ B\ M$ on the one hand and $M\ C\ D$ on the other hand. These two curves connect in M so that their tangents are respectively parallel to the line BM and the line MC, which are the same line. Moreover, any line that crosses BC in M has B on one side and C on the other side. Locally, the points of the first curve close to M are on the same side as B and the points of the second curve close to M are on the same side as C. We hope to be able to prove this thanks to the properties Bernstein polynomials, which are useful in describing the properties of Bézier curves.

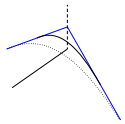
Our Bézier curve construction starts with a broken line that is safe, but the curves stray away from the corners, and in doing so they may collide with the obstacles. We designed two more algorithms, one to check that a Bézier curve stays in safe territory, and the other to repair trajectories whose Bézier curve do collide with the obstacles.

These algorithms are based on a strong property of Bézier curves, which we shall call the *dichotomy property*. In the description above where we showed how to construct the point F which belongs to the Bézier curve, it turns out that the triplets $A\ D\ F$ and $F\ E\ C$ define two new Bézier curves that decompose the original Bézier curve in two complementary subsets. The triplet $A\ D\ F$ defines a curves that covers all the points of the curve defined by $A\ B\ C$ for a parameter $t' \leq t$, while the triplet $F\ E\ C$ defines a curves that covers all the points of the curve defined by $A\ B\ C$ for a parameter $t' \geq t$. The convex hull for $A\ D\ F$ is much smaller than the convex hull for $A\ B\ C$, and thus the dichotomy operation makes it possible to replace a gross approximation of the Bézier curve by a more precise one.

Our checking algorithm repeats this dichotomy operation until it gets enough information to conclude that there is a guarantee that the Bézier curve is included in the union of the interior of two cells and the door between these cells, or that there is a guarantee that the Bézier curve steps out of this union, in which case we have proved that the Bézier curve is unsafe.

When we have an unsafe Bézier curve, we can repair the situation by replacing the Bézier curve $A\ B\ C$ by the composite path obtained by adding two new points M and N which are the centers of (A, B) and (B, C) respectively, and taking the segment (A, M), the Bézier curve $M\ B\ N$, and finally the segment (N, C). This composite path is smooth, because the junctions at M and N have the same tangent and direction, and it is also closer to the initial broken line trajectory. This new path needs to be checked again, and maybe the replacement process can be repeated. Ultimately, if the replacement process were repeated indefinitely the limit of the smooth path would the broken line path.

The following figure illustrates the case where the faulty Bézier curve drawn as a dotted line is repaired into the path composed of two straight line segments and a smaller Bézier curve. The vertical dashed line represents a door and the solid straight line below represents an obstacle.



In our program, we first produce a candidate Bézier curve for each corner, and we then check whether this curve is safe. If not, we produce Bézier curves, that are closer to the broken line and repeatedly check their safety. After a limited number of iterations, the program falls back to the initial broken line corner if need be. We only wish to prove that the checking procedure is correct. For this program, we will not be able to prove that the resulting trajectory is smooth.

6 Exploiting the program and visual feedback

This program can be run inside Coq, using rational numbers for the number system. To compute vertical cells, most computations are ring operations, except when we want to know the vertical projection of a point on an edge, where we use a division operation, so the exact division operation provided for rational numbers suffices.

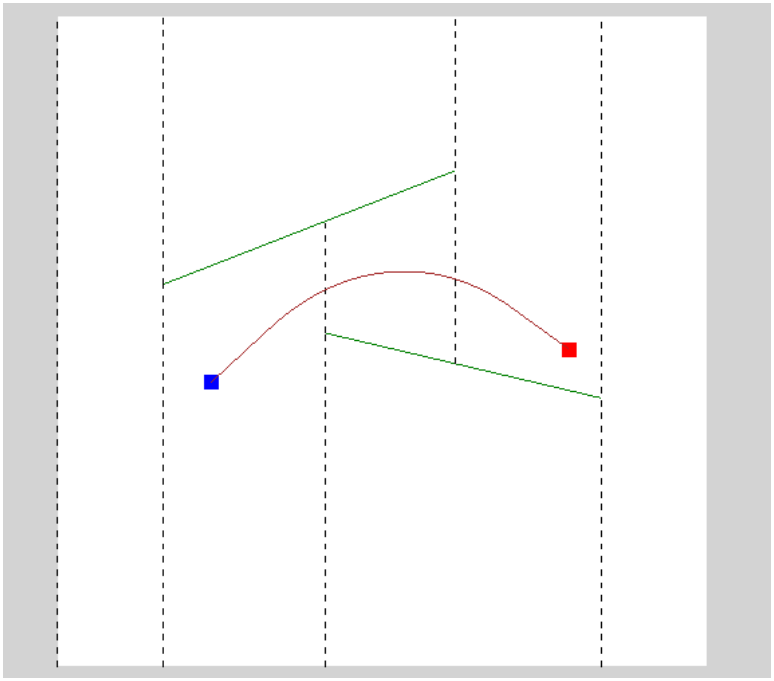
When defining Bézier curves, we only need to construct midpoints of existing points to define the new control points for the successive Bézier curves that we consider. Working with rational numbers suffices to have exact computation.

The output can be displayed by simply traversing the output data structure and generating simple postscript commands for each of the straight line or Bézier curve segment. Our program uses quadratic Bézier curves, while postscript supports cubic Bézier curves, but converting from one to the other is easy. In the end, Postscript uses limited precision numbers so approximations are needed, but these approximations are only useful for display purposes.

The Coq system also provides an extraction facility, so the algorithmic part of our program can be translated to a conventional programming language. We used this to produce OCaml code, which was later translated to Javascript, and we developed a hosting webpage for this Javascript code, visible at the following link:

https://stamp.gitlabpages.inria.fr/trajectories.html.

An example of trajectory computed in this setup is given by the following illustration:



7 Related work

In this section, we mostly concentrate on related work that entails formal verification.

The computation of sub-areas of the plane is already studied in work on convex hulls [8] and triangulations [4, 1]. The algorithm we use to decompose the workspace into cells relies extensively on the orientation predicate taken from the work of Knuth [6].

We argued in this article that the program we designed only uses rational computations. Up to the problem of reasoning about collisions between the broken line trajectories and the obstacles, rational numbers are sufficient. However, smooth curves based on Bézier curve pose a new problem. The coordinates of potential collision points are real algebraic numbers. So while the program description can rely on rational numbers, the proofs concerning collision avoidance require working in a real closed field. Work on the decidability of equality between algebraic numbers is related to the work done in CoRN on the fundamental theorem of Algebra [5]. The Mathematical Components library also contains descriptions of procedures to descide problems with polynomials [3].

Bézier curves can actually be viewed as parametric curves, where the polynomials giving each coordinate are easily described from the control points using Benstein polynomials as a basis of the vector space of polynomials, so this work is also related to previous work on Bernstein polynomials [2].

A piece of work whose motivation has many similarities with ours is an experiment combining a motion planner programmed in Matlab with a formal verification performed by Isabelle [9]. This experiments describes a system that is more powerful than ours, as it relies on analysis, especially tools to reasons about solutions to differential equations, to check solutions that have been elaborated by a Matlab program. However, our objective is to describe a program that

is eventually independent from the interactive theorem prover being used (in our case Coq, in their case Isabelle). To our understanding, their approach is to have Matlab and Isabelle being used at runtime to produce trajectories that are formally verified. On the other hand, our work contains a formal verification of the computations that happen at the time of elaborating the trajectories, while their work receives the results of these computations as if they were provided by an oracle.

8 Conclusion

It is unfortunate that this presentation is more a description of work in progress than a statement of achievement. At the time of writing these lines, the formal verification is focusing on proving the correctness of the vertical cell decomposition algorithm. The naive solution seems rather easy to verify, but the constraint of obtaining cells adds a lot of complexity in the program, as we need to pay attention to the case where two successive events are vertically aligned.

The full program can be seen on the public repository https://github.com/math-comp/trajectories. The incomplete proofs of the first two phases can be seen on the public repository https://github.com/ybertot/VerticalCells.

Bézier curves provide a nice trade-off between the desire to provide smooth trajectories, the expected difficulty of proofs, and the techniques involved in trajectory repair. However, they may turn out to be unsatisfactory with respect to the smoothness of acceleration. For instance, road and railway designers rely on a different kind of curves, known as clothoids or Euler spirals. These are more specific to trajectories for wheeled vehicles. We plan to integrate their study in our work.

Acknowledgements

The initial work on the vertical cell decomposition algorithms was done by Thomas Portet. Studies of potential collisions between Bézier curves and straight line segments were done by Quentin Vermande. Laurent Théry added the possibility to visualize the results on a web-page. x

References

[1] Yves Bertot. Formal verification of a geometry algorithm: A quest for abstract views and symmetry in coq proofs. In Bernd Fischer and Tarmo Uustalu, editors, *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, volume 11187 of *Lecture Notes in Computer Science*, pages 3–10. Springer, 2018.

[2] Yves Bertot, Frédérique Guilhot, and Assia Mahboubi. A formal study of Bernstein coefficients and polynomials. *Mathematical Structures in Computer Science*, 21(04):731–761, August 2011.

[3] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, February 2012.

[4] Jean-François Dufourd and Yves Bertot. Formal study of plane Delaunay triangulation. In Paulson, Lawrence, Kaufmann, and Matt, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 211–226, Edinburgh, United Kingdom, July 2010. Springer.

[5] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000,*

Durham, UK, December 8-12, 2000, Selected Papers, volume 2277 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2000.

[6] Donald Knuth. *Axioms and Hulls*. Number 606 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[7] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, USA, 1991.

[8] David Pichardie and Yves Bertot. Formalizing convex hull algorithms. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *LNCS*, pages 346–361. Springer-Verlag, 2001.

[9] Albert Rizaldi, Fabian Immler, Bastian Schürmann, and Matthias Althoff. A formally verified motion planner for autonomous vehicles. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2018.