A Case Study on Numerical Analysis of a Path Computation Algorithm

Grégoire Boussu
Thales Research & Technology
Palaiseau, France
gregoire.boussu@thalesgroup.com

Nikolai Kosmatov
Thales Research & Technology
Palaiseau, France
nikolai.kosmatov@thalesgroup.com

Franck Védrine
Université Paris-Saclay, CEA, List
Palaiseau, France
franck.vedrine@cea.fr

Lack of numerical precision in control software — in particular, related to trajectory computation — can lead to incorrect results with costly or even catastrophic consequences. Various tools have been proposed to analyze the precision of program computations. This paper presents a case study on numerical analysis of an industrial implementation of the fast marching algorithm, a popular path computation algorithm frequently used for trajectory computation. We briefly describe the selected tools, present the applied methodology, highlight some attention points, summarize the results and outline future work directions.

1 Introduction

Numerical precision of algorithms has become an important concern for modern critical software. Accumulation of rounding errors can lead to serious issues in programs involving floating-point numbers. Such accumulated errors can significantly affect the accuracy of computations and lead to incorrect results. Even for a mathematically correct algorithm — considered in real numbers — its computer implementation can give inaccurate or incorrect results if this implementation does not properly take into consideration numerical precision aspects of the resulting computation in floating-point numbers. In critical software, in particular in control software related to trajectory computation, lack of numerical precision can lead to incorrect results with costly or even catastrophic consequences. Well-known examples include the Patriot missile failure in 1991[1] and the crash of Ariane 5 in 1996[2].

The fast marching algorithm [10] is a popular path computation algorithm frequently used for trajectory computation in autonomous systems. It answers the question of which path is optimal between two given nodes, that is, has the shortest time or, more generally, the smallest weight. The algorithm works in two steps. A first step performs a forward wave front propagation from the given origin point, computing the time the wave front will take to reach each point (of the plan, or grid, or graph). A second step uses the resulting computation to perform a backward propagation from the final point to the origin point in order to compute an optimal path. This algorithm has various applications for trajectory computation and image segmentation. The purpose of this work is to investigate the numerical precision of an industrial implementation by Thales of this algorithm over a discrete grid.

[1]See `https://www-users.cse.umn.edu/~arnold/disasters/Patriot-dharan-skeel-siam.pdf`.
[2]See `https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html`.

Numerical analysis of such trajectory computation algorithms is a very challenging and time-consuming task. Execution paths in the code are typically very long and go through many instructions. Each of them can have an impact on precision and robustness of the algorithm. Indeed, such a path can go through many *unstable branches*, that is, branches after a conditional expression for which a small imprecision of computation or a small variation of input values can change the truth value of the condition and lead to another branch in the code (ex: *else* instead of *then* branch of a conditional statement, or one more loop iteration), possibly impacting the rest of the algorithm. Rounding a floating-point number to an integer can have a similar impact when the resulting integer is later used in the code: if a value around 100.0 can be rounded to 99 or 100, it can potentially have a significant impact. Moreover, since the algorithm simulates a continuous real space by a discrete grid, a deviation at one node can easily involve different nodes and thus lead to a quite different result.

Various techniques and tools have been proposed to analyze the precision of program computations. They include dynamic analysis and static analysis techniques. In this work, we use three popular numerical analysis tools: Cadna [7] and Verrou [6] realizing (possibly unsound) dynamic analysis, and FLDLib [12] performing a combination of sound abstract interpretation and dynamic path exploration.

Contributions. This paper presents a case study on numerical analysis of an industrial implementation of the fast marching algorithm. While the considered implementation is currently not publicly available, the underlying algorithm is classic, therefore we believe that the presented methodology and findings can be of interest for other implementations of similar (and possibly other) algorithms. We briefly describe the selected tools, present the applied methodology combining several tools, highlight some attention points, summarize the results and outline ongoing and future work directions.

Outline. Section 2 presents the considered algorithm. Section 3 describes the verification methodology, the selected tools and our findings. Section 4 provides a conclusion and future work perspectives.

2 The Verification Target: the Fast Marching Algorithm

This section provides a simplified presentation of the problem and the implemented algorithm without giving all technical and theoretical details (which are not mandatory for understanding the paper). For a more thorough description of theory behind the Fast Marching Algorithm, one may refer to [11].

The problem under consideration for the study is named the *minimum-cost path problem*. On a finite graph with weighted edges, this problem can be stated as follows: which path to take between two specified vertices so that the sum of weights along this path is the lowest among all possible paths between the two nodes. When the weight is (seen as) the distance between the nodes, this problem is also called the *shortest path problem*, and the cost is (seen as) the time to reach the point. Different algorithms exist to solve the shortest path problem (e.g. Dijkstra, Bellman-Ford).

In our case, we are interested in the definition of the *minimum-cost path problem* in the continuous case: let us consider the problem in $\mathbb{R}^n$. A cost density function $\tau : \mathbb{R}^n \to (0,\infty)$ gives the cost at each point of the space. The *minimum-cost path problem* between A and B, two points in $\mathbb{R}^n$, is to find a path $c(s) : [0,\infty) \to \mathbb{R}^n$ that minimizes the cumulative cost (often interpreted as the arrival time) from A to B. The *cumulative cost* for a path c between A and M is:

$$T_c(M) = \int_0^l \tau(c(s))\, ds$$

where l is the length of path c between A and M, $c(0) = A$ and $c(l) = M$. Therefore, c_{sol} is a solution to the problem if and only if $c_{sol} \in \mathscr{C}$ where $\mathscr{C}$ is the set of all paths c between A and B with the minimum $T_c(B)$.

As stated in [10], if c_{sol} is a solution to the problem, it satisfies the equation below, named the Eikonal equation, for all $M \in c_{sol}$:

$$||\nabla T_{c_{sol}}(M)|| = \tau(M)$$

where ∇ denotes the gradient, and $||\cdot||$ denotes the Euclidean norm. This equation is, in particular, a way to describe the propagation of a wave front initiated in point A. The front speed at point M is given by $1/\tau(M)$, and $T_{c_{sol}}(M)$ is the time of arrival of the front from point A to point M.

In the presented definition of the problem, the value of τ at a given point depends only on the point's location. This case is qualified as *isotropic*. If τ also depends on the direction of the path at the point, the cost function is *anisotropic*. A method to solve this equation in the case of an isotropic problem discretized on a Cartesian grid was proposed by Sethian in 1995 [10] and has become the starting point for many extensions. This method, named *fast marching method* (FMM), shares many aspects with Dijsktra's algorithm. Once the equation is solved for all points of the grid, a second step is necessary to figure out the (or one of the) optimal path solution(s) to the minimum-cost path problem. We will go over the two steps sequentially.

2.1 First Step: Solving the Eikonal Equation

Basically, given a grid and a starting point A of the grid, Sethian's method allows one to calculate the arrival time $T_c(M)$ to any point M of the grid over a minimum-cost path c starting from point A. Each point M of the grid has 4 neighbors as shown in Fig. 1. Based on a relevant approximation scheme, $T_c(M)$ is calculated considering the possibilities that the wave about to reach the point M comes from North-East (with a contribution from the neighbors above and on the right), or South-East (with a contribution from the neighbors below and on the right), or South-West (with a contribution from the neighbors below and on the left, as shown in Fig. 1) or North-West (with a contribution from the neighbors above and on the left). Starting the algorithm with $T_c(A) = 0$ and $T_c(M) = \infty$ for all $M \neq A$, and picking the next point M to study in an appropriate order, the process progressively computes the arrival time $T_c(M)$ (or more precisely, its approximation due to the discretization) for all points M of the grid.

We can further explain the process using the interpretation of the equation with a wave front, illustrated in Fig. 2. The black points of the grid have their final value $T_c(M)$ computed, the gray ones have a tentative value $T_c(M)$ computed, and the white ones still have $T_c(M) = \infty$, as set at the initialization step.

The set of gray points is named the *narrow band*. Intuitively, at each step of the algorithm, the black points have already been reached by the wave, and at least one of the gray points will be reached next, before any of the white points will be reached. Just like in a classic implementation of Dijkstra's algorithm, a priority queue is used to store the gray points. When a point M is selected from the queue, its neighbors M' enter the queue (if they were not already part of it) and get their arrival time values $T_c(M')$ calculated or updated based on $T_c(M)$. The next point M to be selected in the queue is the one with the lowest tentative arrival time $T_c(M)$. When selected, such a gray point gets its tentative value $T_c(M)$ turned to the final one, and the point itself is removed from the queue and labeled as black. Intuitively, since the tentative arrival time of the wave to this point is the smallest one among the gray nodes, it cannot be reached even faster through some other node (for which the arrival time will necessarily be bigger) hence the computed arrival time to it is final. At the beginning of the algorithm, the priority queue is initialized with A.

The fast marching method has two interesting features:

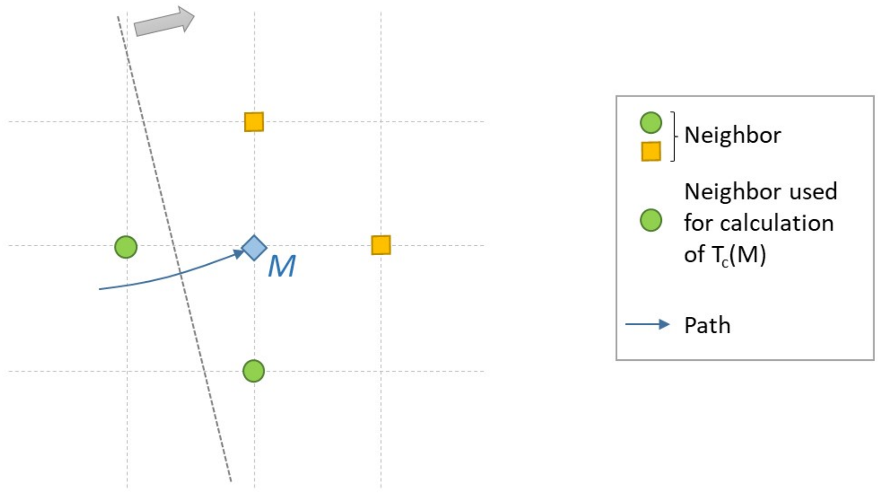

Figure 1: Neighbors selected for calculation of $T_c(M)$

- It is efficient in terms of computational complexity. Indeed, its complexity is similar to that of Dijsktra's algorithm, and is of $\mathscr{O}(n\lg(n))$, with n being the number of points of the grid.
- It can be proven that the FMM produces a solution that satisfies everywhere the discrete version of the Eikonal equation, leading to an approximation of its so-called *viscosity* solution (see for example [4] on viscosity solutions). So when the grid spacing tends to 0, the solution provided by the FMM algorithm tends to the continuous solution of the equation.

Though, as $T_c(M)$ is calculated based on the T_c of the neighbors of M, the calculation errors may propagate over the entire grid. Added up, these errors may lead to a discrepancy for points far from point A and impact the precision of the global result expected from using FMM. Studying the order of magnitude of this discrepancy is of great interest to be confident in the implementation of the algorithm.

2.2 Second Step: Finding an Optimal Path by a Backward Propagation

The theory provides a way to find an optimal path, thanks to a property of such a path: its direction is always normal to the wave front [1]. To produce the result, a so-called back-propagation from the final point (supposed to be on the grid) is realized, based on a gradient descent following the direction perpendicular to the wave front curve.

Though, once the arrival time values $T_c(M)$ are calculated for each point of the grid, we are still in a discrete space and the gradient calculation is not straightforward. The gradient descent can be approximated by selecting for each point its predecessor among the neighbors, the appropriate one being the (or one of the) neighbor(s) with the lowest $T_c(M)$. But this approach leads to a path made of following segments that can be perpendicular one to the next. Moreover, aggregating the length of each segment of the path will generally lead to a value overestimated compared to the optimal path length in a continuous space. It would be preferable to provide a visually smooth path with its length approximating the length of the viscosity solution of the Eikonal equation.

Such an alternative can be implemented with the following approach: starting from the final point of

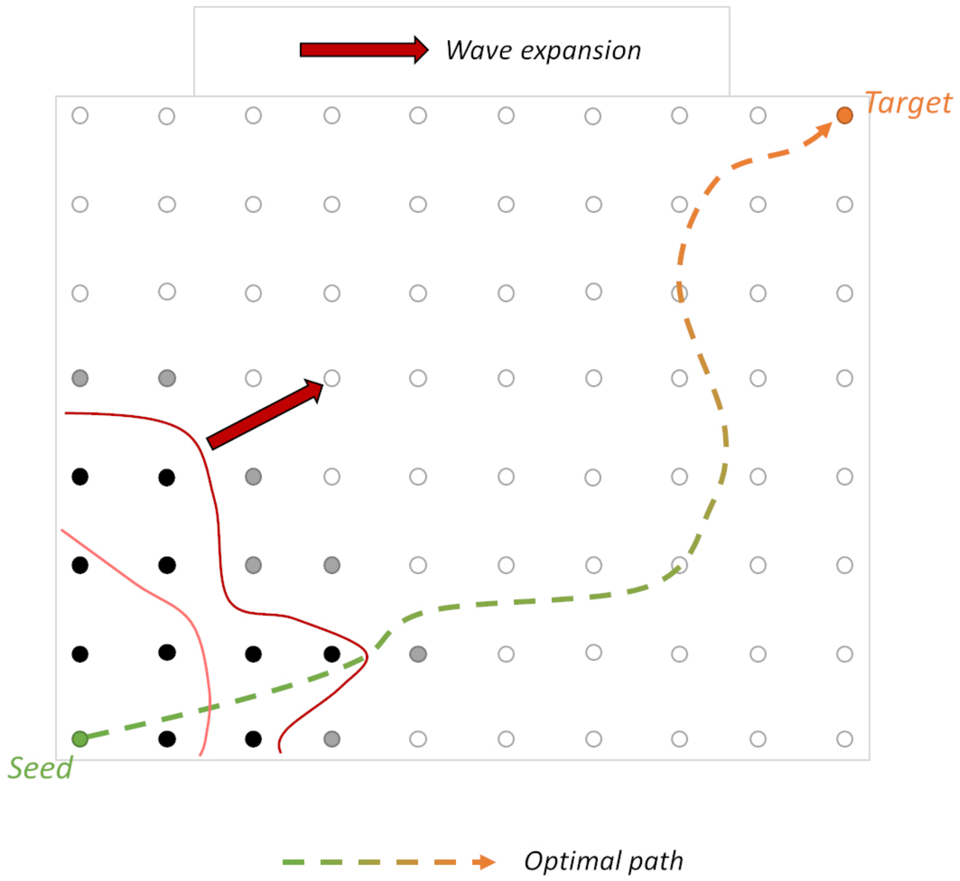

Figure 2: Propagation of the wave front

the path, a *pseudo*-gradient is calculated on each segment around this point, as shown in Fig. 3. The best point on the segments, i.e. the point (or one of the points) minimizing $\Delta T_c/distance$ (that is, maximizing the speed of the wave) is selected as the previous point of the approximated path. A similar approach is taken to find out the best point when the gradient is calculated from any point in the middle of a segment. This leads to a much smoother path, whose length provides a good approximation of the expected length of the viscosity solution.

Just as in the case of the fast marching algorithm, the calculation is made one point after the other. Therefore, the calculation errors due to the implementation can lead to an aggregated discrepancy. An analysis of sensitivity of the implementation to these errors is thus required.

2.3 Applications of These Algorithms

Many fields of application exist for these algorithms. The first is of course related to path calculation leading to the shortest time between two points, considering the speed of the mobile agent depending on its position in an area. A less obvious application could be image segmentation [3]. To allow for different and more specific situations, many extensions to the method have been developed. To name a few: the possibility to deal with anisotropic costs [8], or with time-dependent costs with no restriction on sign [2], or the extension taking into consideration constraints like minimum turning radius of the

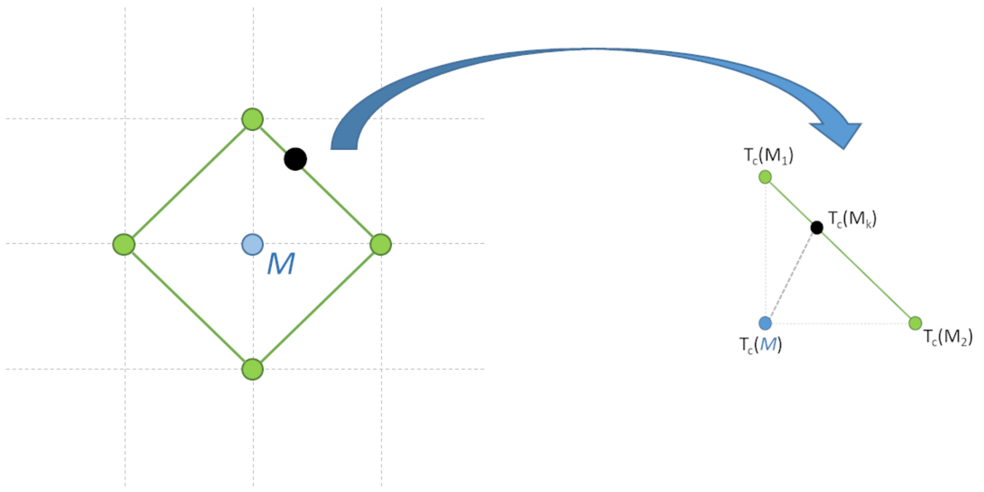

Figure 3: Calculation of pseudo-gradient on a segment

moving agent [9].

In our case, the fast marching method is a general approach to produce paths optimizing any kinds of criteria (or a mix of criteria). The most straightforward situation would be to aim at minimizing time to reach a location, while the area through which we can move is made of danger-free zones where the speed can be high, and others surrounded by dangers (mountains, ...) where the velocity should be reduced. Let us consider another example where time is not the criterion to optimize: the pilot of a plane wishes to avoid turbulence areas ahead (considered as static). He may want to find a good balance between disturbance due to very strong winds and the additional distance incurred by avoiding these areas. If the plane has a steady cruising speed, by defining the cost function τ with high values in the center of the turbulence areas and decreasing values towards the outside, the fast marching method can provide an appropriate path to follow (see Fig. 4).

For use cases where a lack of precision can generate additional risks (e.g. air traffic, autonomous drones), aggregated calculation errors can significantly impact the result of the computation. This concern motivated the current study.

3 The Verification Approach and Results

The target implementation of the path computation algorithm contains more than 6,200 lines of C++ code and provides several test cases. They include realistic test cases over a square grid with 200x200 nodes and obstacles simulated by higher weights, as illustrated by Fig. 4. Internally, the code uses some C++ STL (Standard Template Library) containers, like vectors, maps and priority queues. So, formal numerical analysis of this implementation and its adequacy with respect to the underlying mathematical formulas within a short period of time requires concentrating on successive research questions:

RQ1: What is the accuracy of the computed path cost?

RQ2: Is the computation robust (meaning that a small perturbation of inputs leads only to a small variation of outputs)?

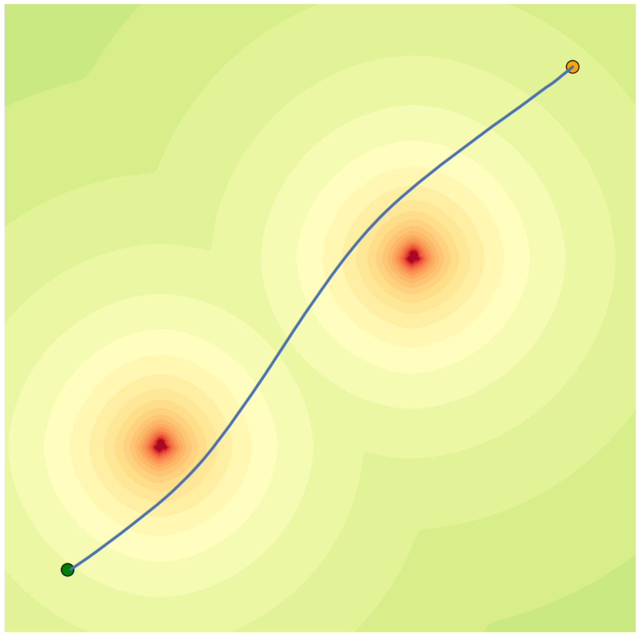

Figure 4: Result of calculation of a path avoiding turbulence areas

RQ3: How does the computed path compare to the path obtained on a more or less precise grid (say, with twice more or twice fewer points on each side)?

This paper focuses on RQ1 and RQ2, while RQ3 is left for future work. To address these questions, we decided to apply the following methodology that was successfully applied earlier on some simpler numerical use-cases, except the last item that is more related to deductive verification:

- instrument the tests with different numerical analysis libraries to identify the difficulties in obtaining relevant analysis results and then refine the verification objectives, such as accuracy requirements;
- enlarge the tests into analysis scenarios to check whether the analysis scales up and still provides precise results. Fine-grained analysis scenarios typically replace concrete input values by very small input intervals and then apply conservative interval operations; larger analysis scenarios can also be considered;
- apply modular formal verification to the components of the target implementation and assemble the reasoning results to provide a proof of global correctness.

After presenting the common instrumentation in the next section, we will apply Cadna to address RQ1, Verrou to address RQ1 and RQ2 by comparing with Cadna results, and FLDLib to address RQ1 and RQ2 to investigate the unstable branches that may have a significant impact on the robustness results.

3.1 A Common Instrumentation Mechanism for Different Verifications

The mechanism for building the target implementation of the fast marching algorithm uses the `cmake` tool; so a slight modification of the file `CMakeLists.txt` enables adding some new executable targets compiled with specific compilation flags. This feature enables the source code to be easily compiled with analysis libraries into a single executable. For the verification purposes, this instrumentation mechanism competes with abstract interpreters when the abstractions to be used are generic (intervals, affine forms). But our case study also requires the elaboration of specific abstractions. So, to quickly explore and debug these newly created abstractions, an instrumentation based on C++ operator overloading and template/-macros mechanisms seemed to us more efficient than using an existing generic abstract interpreter.

Our default instrumentation mechanism replaces `double` and `float` types with data structures carrying analysis information like accuracy, as it is often done by instrumentation libraries [7, 12]. Such data structures implement an overload for the arithmetic operations (`+`, `-`, `*`, `/`, `pow`) to infer numerical properties like the accumulation of round-off errors in numerical computations. Integer types like `int` or `unsigned int` are not instrumented by default. But, the source code can use explicit intrumentation for these types by replacing `int` by `EnhancedInteger<int>` whenever it makes sense for some `EnhancedInteger` template class to define. The C++ compiler helps then to statically propagate these custom types on the source code since an operation manipulating `int` and `EnhancedInteger<int>` generates an error if its result is assigned to an `int` and not to an `EnhancedInteger<int>`.

For each analysis target, the file `CMakeLists.txt` adds specific compilation flags like `-I.../analysis_include -include std_header.h -DFLOAT_MY_ANALYSIS` to build the target. The directory `.../analysis_include` contains the file `std_header.h` that conditionally loads the appropriate analysis data structures for the flag `FLOAT_MY_ANALYSIS` and replaces the `double` type with the macro `double` defined by `#define double EnhancedFloatingPoint<double>`.

The research questions stated in the beginning of this section systematically compare two or more executions. These executions can (and do) follow different control flows (that is, different execution paths) in the target program. In our experiments, we instrument the code with three different strategies:

1. A single run of a synchronous analysis with a single control flow: this single analysis run propagates complete analysis information for every variable at every point of the execution path until the end of the program.
2. Multiple runs of asynchronous analyses with a single control flow: a run propagates partial analysis information until the end of the program. With multiple runs, the user can compute the analysis result as a model from the correlated input/output data.
3. A single run of a synchronous analysis with multiple control flows: this single analysis run propagates complete analysis information and thanks to additional local loops, it covers all possible execution paths (which corrects a weakness of Strategy 1 with an additional instrumentation and execution cost).

For the last strategy, we use `SPLIT`/`MERGE` macros introduced and used by FLDLib [12]. A pair of such macros (`SPLIT` and `MERGE`) define a so-called `SPLIT`/`MERGE` section: it expands into a local loop that iterates over all the reachable control flows of the `SPLIT`/`MERGE` section in order to analyze them one after another. A local memory defined in the `SPLIT` macro saves the memory before the section and restores it at the beginning of the loop body for an exploration of a new control flow. `SPLIT` also saves a control flow identifier for an exploration of a new execution path of the section. It then increments this identifier to cover another execution path in the next loop iteration. At the end of the loop, `MERGE` incrementally synchronizes the results of the local analyses to create a single analysis summary per variable. The analysis is then continued with this summary until the end of the program.

Beyond these generic principles, the instrumentation may encounter some problems listed below, which may require minor adaptations to the source code for analysis purposes. In practice, the first two problems are absent in the modern C++ implementation of the fast marching algorithm.

- dynamic allocations with C functions `malloc` and `free` should be replaced by C++ `new` operator with smart pointers (or `delete`): `EnhancedFloatingPoint` often has non-trivial constructor and destructor and the `malloc` and `free` functions do not call them unlike `new` and `delete`.
- C functions with variable number of arguments and specialized format specifiers (such as `scanf` and `printf(“%e”, ...)`) should be replaced with `std::cout`, `std::cin` calls because `“%e”` does not recognise `EnhancedFloatingPoint`.
- In a divide-and-conquer analysis approach, we typically instrument certain parts of the code and leave others unchanged. But, replacing `int` with `EnhancedInteger<int>` also generates many other replacements. In the case of the fast marching algorithm, the forward propagation part (Sect. 2.1) and the backward propagation for path generation (Sect. 2.2) share some common methods. However, replacing `int` with `EnhancedInteger<int>` is only required in the backward path generation. In this case we rename the original method as a template method in the private section of the class. Then, we duplicate the public method, one with `int` arguments and the other with `EnhancedInteger<int>` arguments. The bodies of the original method and its duplication just call the template private method. From the caller's perspective, the C++ “name lookup” generally generates correct calls.
- The second argument of binary operators whose first argument is of type `EnhancedInteger<int>` may be `int`, `unsigned`, `double`, `EnhancedFloatingPoint<double>`. The instrumentation needs precise overloaded operators to be called by all the constructs of the source code. In C++-03, providing an interface for this instrumentation that correctly connects the source operator with the correct overloading for all type combinations was a very complex task and ultimately produced a resulting interface that was difficult to manage. That is probably the reason why Cadna 2.1 does not support `long double`. Then, the *SFINAE* (Substitution failure is not an error) [14] feature allows the definition of such a robust interface, but this remains very technical with maintainance difficulties. Our libraries use recent C++-20 concepts to manage this `class` interactions, which makes the instrumentation more robust.

3.2 Results of the Approach based on Dynamic Analyses

To address RQ1, the objectives of the first analyses are

- ensuring that the code can be instrumented with dynamic analysis libraries (that are generally simple to use from the instrumentation point of view),
- obtaining initial quantitative accuracy properties to be refined later with more complex analyses,
- evaluating the robustness of the implementation: a small perturbation of input data should generate a small deviation in the outputs. If the implementation is not robust, we have no chance of proving formal functional properties, such as “the results depend in a limited way on the size of the grid”.

Dynamic analysis with stochastic arithmetic meets these goals; that is why we use it as a first approach. To do this, we couple the instrumentation mechanism described in the previous section with stochastic analysis libraries in order to obtain accuracy and robustness results without any modification to the source code. Such analyses only require a test case and explore the impact of minor perturbations on the results after execution of the test scenario.

The Cadna[3] [7] library evaluates the accuracy of a code by propagating three executions in a single run (synchronous analysis with single control flow). This leads to maintaining three values (v_0, v_1, v_2 in

[3]See `https://www-pequan.lip6.fr/cadna`

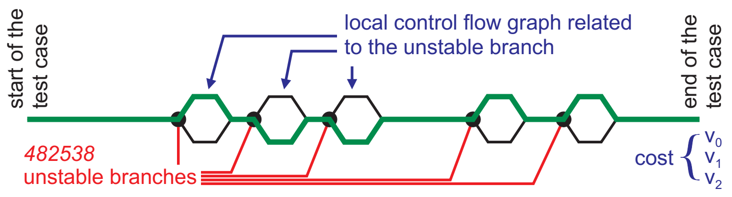

Figure 5: (Simplified) trace of the execution with Cadna (in green), where the cost of the path is evaluated by three values v_1, v_2, v_3.

Fig. 5) for each computed variable *var* instead of one. To evaluate potential impact of rounding errors, with this library each floating-point computation involving *var* is dispatched over v_0, v_1 and v_2. The ideal results are randomly rounded up or down — with a probability of 50% for up or 50% for down — instead of using the deterministic IEEE-754 rules implemented in the processor. The average of the three values provides the expectation of the computed result, while the standard deviation provides an estimate of the accumulation of round-off errors [13]. Such analysis is synchronous: the three executions are *forced* to follow the same control flow (shown in green in Fig. 5) and do not evaluate *unstable branches*, for which a possible imprecision can impact the result of a conditional test (and therefore the branch taken after it). Let us consider a comparison, say $d < d'$ between two `double` values d and d' instrumented by Cadna. It compares each of the three values v_0, v_1, v_2 obtained for the first value d with the corresponding value of the three values v'_0, v'_1, v'_2 obtained for the second value d'. Suppose that the first two comparisons return true and the last returns false. Cadna just reports an "`UNSTABLE BRANCHING`" and propagates the last execution into the then branch as well, even if it would naturally execute the else branch.

The instrumentation quickly succeeds on the target code with Cadna (thus fulfilling the first objective). The algorithm computes (in 0.556s) a path of 322 points as well as the cost of the path, stored in variable `cost`. Since `cost` is the value that the algorithm attempts to optimize, we expect it to be robust. The cost has an average of `0.392` and a relative error of 1.323×10^{-15} due to the accumulation of round-off errors. Cadna also reports the following warnings:

```
CRITICAL WARNING: the self-validation detects major problem(s). The results are NOT guaranteed.
There are 977194 numerical instabilities
1687 UNSTABLE MULTIPLICATION(S), ...
482538 UNSTABLE BRANCHING(S), 260343 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)
```

The execution of the test case takes 0.182s and generates a shorter path of 320 points with a cost of 0.393257, that is outside the error range computed by Cadna. That confirms — as suggested by the warnings — that the Cadna results are not conclusive: unstable branches (not evaluated by Cadna) probably have a major impact on the path computation and therefore on the robustness of the algorithm.

The second analysis uses the Verrou[4] [6] tool[5]. Verrou evaluates the accuracy of a code during multiple runs by randomly rounding up or down every floating-point computation (asynchronous analysis with single control flow). Unlike Cadna, Verrou does not need additional memory: since the execution of the program perturbed by Verrou is non-deterministic, multiple runs provide multiple output values (see Fig. 6, where we show only four traces for readability). The average and the standard deviation of the output respectively provide the expected stochastic result and an error that is representative of

[4]See `https://github.com/edf-hpc/verrou`

[5]The Verificarlo [5] tool (see `https://github.com/verificarlo/verificarlo`) can be expected to produce similar results, but it was not used in this study.

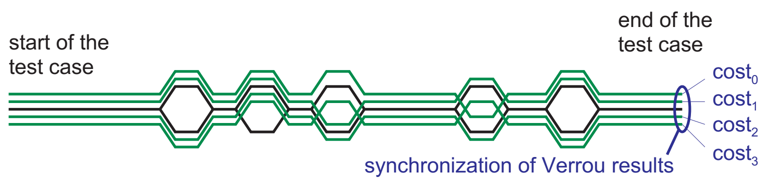

Figure 6: Four (simplified) traces (in green) from the 10 executions with Verrou. Each trace leads to computing a (possibly different) path and its cost

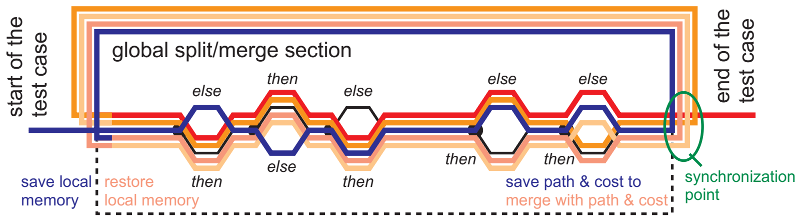

Figure 7: (Simplified) trace of the FLDLib analysis, in which the code of a SPLIT/MERGE section is executed several times for all executable control flows inside it and the results are consolidated at the end

the accumulation of round-off errors. With ten runs (performed in 16.123s), Verrou provides different lengths for the optimized path: 324, 307, 308, 307, 315, 315, 314, 317, 312, 300. The average of the ten values of `cost` is evaluated to 0.3922 and its standard deviation to 2.68×10^{-4}.

The error produced by Verrou seems to be more consistent than that of Cadna with respect to the original IEEE-754 floating-point execution. The multiple runs nevertheless do not contain the value of cost produced by the test case execution since $0.393257 \gg 0.3922 + 0.000268$. We relaunch the Verrou analysis several times and we systematically obtain an average and a standard deviation close to these values. That means that the floating-point execution takes a control path that is distinct from other control paths in terms of their impact on the `cost` value. At this point, the use of formal methods appears relevant to further investigate the relative instability of the floating-point execution.

3.3 Evaluating the Impact of Perturbations on the Control Flow and the Resulting Path

To further address RQ1 and investigate RQ2, the third analysis relies on the FLDLib[6] library [12] to provide a sound over-approximation of the accumulation of round-off errors by maintaining the ideal (in practice, a very precise machine) computation and the floating-point computation in parallel. FLDLib relies on SPLIT/MERGE sections (presented above) to analyze unstable branches by exploring each of the different control flows using abstract interpretation and by consolidating the observed results at the end. This analysis propagates affine forms for rounding errors and for the possible values on the test case. The mathematical representation of an affine form is $\alpha_0 + \sum_{i=1}^{n} \alpha_i \times \varepsilon_i$, where α_i are constant coefficients in $\mathbb{R}$ (approximated by floating-point values with a large mantissa) and ε_i are free variables in the interval $[-1.0, +1.0]$. The error symbols ε_i represent unknown values due to basic approximations of complex computations. The program variables can share some ε_i, which creates linear relationships between some of these variables. FLDLib also offers advanced features to reduce the size of the re-executed code with local synchronization annotations (see the FLDLib library documentation and [12] for more detail).

[6]See `https://github.com/fvedrine/fldlib`

```
double jm_res = (y - yMin) / dy;
unsigned int jm = (unsigned int) jm_res;
```

Figure 8: First unstable branch identified with FLDLib

In practice, we activate the FLDLib analysis after initialization of the mesh. Therefore, the grid is composed of points with floating-point coordinates considered exact, i.e. without any numerical error. With this assumption, the analysis only propagates affine forms over 4 execution paths. A simplified version of an execution trace of FLDLib is illustrated by Fig. 7. After several attempts, we have found the right settings: 319 bits for the internal mantissa of the coefficients of the affine forms and a limit of 30 shared symbols per expression. With an internal mantissa of 255 bits, the intervals representing the ideal computations are too wide at the end of the forward wave propagation (see Sect. 2.1). Therefore, the cost associated with the resulting path is too strongly over-approximated with the interval $[-\infty,+\infty]$: the algorithm performs at some moment a division by the distance between two points, and if the localization of these points is imprecise, a potential division by zero due to interval arithmetic gives this result. This first FLDLib experimentation (internal mantissa of 255 bits) is nevertheless interesting because, like with Cadna and Verrou before, it also produces a resulting path (with 278 points) that is different from the path produced by the floating-point execution of the test case (with 320 points).

FLDLib quickly identifies the location in the source code of a first unstable branching, for which it explores both branches in floating-point and ideal computation. It concerns the computation of the second instruction of Fig. 8 with the values `y=0.5`, `yMin=0`, `dy=0.005`.

In floating-point semantics, the value of `jm_res` is 100. In the ideal semantics, the value of `dy` is $5\times10^{-3}+1.0408\times10^{-18}$ since all the constants take the same floating-point value for both semantics; therefore, the analysis shows that the ideal value of `jm_res` belongs to the small interval $[100-2.082\times10^{-16}, 100-2.081\times10^{-16}]$. The value of `jm` is then 100 in floating-point semantics and 99 in ideal semantics, which creates an unstable branching. The analysis then separates the joined control flow of both semantics into two control flows and explores them separately. These control flows merge at the end of the source code after the computation of the path and its cost. The merge operation computes the numerical error from the substraction between the floating-point value and the ideal small interval of the cost, each value being inferred by the corresponding control flow. The result of this second FLDLib experimentation (internal mantissa of 319 bits) shows an interesting finding: the unstable branch has no impact on the cost and on the points of the path, even if the sorted priority queue (see the gray points of Fig. 2) is organized differently in the two control flows. Therefore, the resulting paths both have 320 points, like the floating-point execution and they return a relative error of 7.058×10^{-16} for the cost.

The duration of this second FLDLib analysis is 114 min after a limited exploration of only 4 execution flow paths. For one control flow path, the analysis encounters 351 296 unstable branches. Therefore, the estimated time for the complete analysis would be $2^{351\,296}\times 114\,\text{min}/4$. Nevertheless, these preliminary results allow us to identify the first unstable branches and to show their absence of impact on the computed path and its cost.

As another finding, this second FLDLib analysis also provides the complete list of locations in the source code of the unstable branches encountered. This list has only 5 locations (which are executed multiple times due to loops), each with a unique calling context. The calling contexts show that the first 4 locations (one of which is the unstable branch of Fig. 8) belong to the forward wave propagation (Sect. 2.1) and that the last location belongs to the backward propagation dedicated to the path generation (Sect. 2.2).

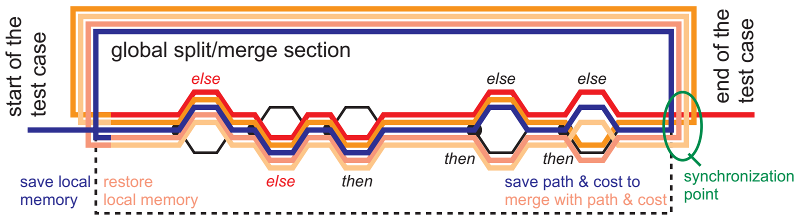

Figure 9: Trace of the FLDLib analysis forgetting unstable branches related to unsigned int conversion

To investigate the impact of other unstable branches, we modify the analysis with an analysis library that only targets the last unstable branch: the one that has a direct impact on the outcome of the path. The reason is that the first unstable branches listed potentially concern all the cells of the grid of Fig. 2 (as they are part of the forward wave propagation); therefore, they have little chance of having an impact on the final path, whereas the last unstable branch directly concerns the cells crossed by the resulting path (as it belongs to the path generation step). Concretely, the new analysis forces the ideal computation of the unstable branch created by the `unsigned int` conversion of Fig. 8 to be equal to the floating-point point computation. Indeed, we observe in Fig. 9 that for the first branches of the analysis paths, the control flow of ideal computations follows the floating-point control flow, which was not the case in Fig. 7. The duration of the new analysis increases significantly: 17 h 20 min instead of 114 min to cover four different branches[7].

The first iteration of this third FLDLib analysis follows a joined control flow for both semantics until the path and cost are computed, which was not the case in the previous analysis. The reported error for this iteration on cost is 4.71×10^{-13}, which is consistent with the Cadna results (1.323×10^{-15}) since FLDLib provides a guaranteed over-approximation while Cadna returns a stochastic estimation of the error. Then, as expected, the first unstable branch is found during the backward path generation. The second analysis iteration follows only the floating-point control flow and it gives the same result for cost as the reference execution. The third analysis iteration follows only the ideal control flow and the MERGE macro at the end of the `main` function creates an error of 4.66×10^{-13} as the maximum difference between the ideal cost and the floating-point cost. An important finding here is that this small error satisfies a sufficient stability criterion for our optimization algorithm, even if there are only two unstable branches evaluated.

The robustness of this conservative analysis, if it is confirmed on all paths despite the unstable branchings encountered by Cadna and qualified by Verrou, suggests that certain computations are redone in different parts of the algorithm, notably between the forward wave propagation (Sect. 2.1) and the backward propagation for path generation (Sect. 2.2). This would be another interesting finding for such kinds of algorithms, where some small steps are recomputed several times. Indeed, the stochastic analysis does not ensure the introduction of exactly the same perturbation if exactly the same computation is performed several times. Suppose such a redundant computation evaluates to a value *res* the first time, the stochastic analysis evaluates it to $res + \delta$ the second time, since the pertubations introduced by the analysis are not

[7]Here is an explanation for this time difference. The first analysis very early encounters an unstable branch that separates the floating-point control flow from the ideal control flow. The analysis of the floating-point control flow then propagates only constant values and the analysis of the ideal flow stops propagating affine forms related to the difference between the float and the ideal value since the floats are no longer present. Conversely, the new analysis has to propagate constants for floating-point values and affine forms for ideal values and for errors during longer execution fragments, which is costly, including the reduced product between the inferred error and the subtraction of ideal value and floating-point value.

	Cadna	Verrou	FLDLib: 4 paths over $2^{351\,296}$
instrumentation time	10 min	0 s	3 h
analysis time	0.556 s	16.123 s	17 h 20 min
cost error	1.323e-15	2.68e-4	4.71e-13
error of mean cost value wrt. reference execution	1.25e-3	1.27e-3	4.71e-13
indicative confidence in results (/10)	4	6	7
reason of error inconsistency	unstable branching not evaluated	original float execution not reached	no inconsistency for 4 paths

Figure 10: Analysis summary (the floating-point reference execution time without analysis being 0.182 s)

the same. Hence, the branch taken after the first computation may be different from the one taken after the second, whereas the deterministic IEEE-754 computation guarantees that it will be the same branch. This issue also occurs for FLDLib analysis in case of over-approximations. In this case, the evaluation result is $res + [a,b]$, but the analysis cannot guarantee that it is the same branch because the value chosen in $[a,b]$ the first time may be different from the value chosen in $[a,b]$ the second time. Since the intervals for the ideal values are very very small (319 bits of mantissa is equivalent to a precision of 4.68×10^{-97}) and since the main linear relationships are preserved between the variables, the analysis is likely to avoid certain over-approximations that would consider unreachable branches and generate false negatives.

Figure 10 shows a summary of our first experiments, which required little investment in annotations of the source code, but a lot of effort in the definition and configuration of the analyses. It gives an indicative (and subjective, based on our experience) level of confidence for the results of each tool. The instrumentation time corresponds to the time that was required for the authors to instrument the code. The analysis time shows the tool execution without the compilation steps. The cost error is the error output directly produced by the tools for the `cost` variable. It concerns the standard deviation of the cost values for the stochastic tools (Cadna and Verrou) and the conservative error for the formally guaranteed tool (FLDLib). The error of mean cost value is the difference between the mean of the cost values computed by the tools and the original floating-point evaluation of the variable `cost` computed by the code without any instrumentation. The indicative confidence in the corresponding analysis increases when both errors become closer. The reason of error inconsistency is given in the last line.

3.4 Ongoing Work on Formal Verification for Thin Numerical Scenarios

The aforementioned results are promising and make us believe that a complete formal robustness analysis for this case study is possible. But we need to cover all of the 351 296 unstable branches identified by the FLDLib analysis to know if the accuracy of the cost is rather close to 1.27×10^{-3} or 4.71×10^{-13} for this test case (cf. Fig. 10). This section presents our ongoing work in this direction.

For this purpose, we add local synchronization annotations around the detected unstable branches (see the resulting trace in Fig. 11). That means that the `unsigned int` variable receiving the conversion of the floating-point computation in Fig. 8 is conditionally defined. For instance, the evaluation of `jm_res` with the values `y=0.5`, `yMin=0`, `dy=0.005` can be seen as producing an integer defined as

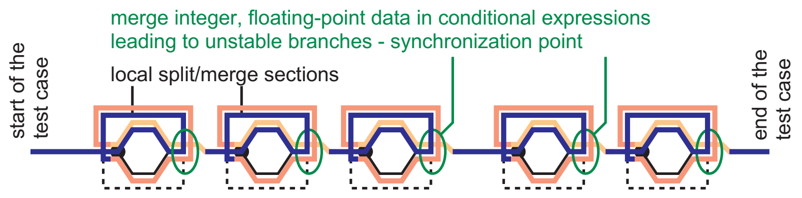
merge integer, floating-point data in conditional expressions
leading to unstable branches - synchronization point
local split/merge sections
start of the test case
end of the test case

Figure 11: (Simplified) trace of the FLDLib analysis with local synchronization points

if b_0 *then* 100 *else* 99, where b_0 is a fresh and free logical variable in $\{\text{true}, \text{false}\}$. For this unstable branch, b_0 evaluates to true in the floating-point semantics and false in the ideal semantics. Further unstable branches have a more complex evaluation in ideal semantics.

For this propagation, we define a new conditional domain[8] in FLDLib that contains cascading conditional expressions or a simple integer value. This domain is implemented as an instantiation of `EnhancedInteger` template mentioned in Sect. 3.1. Therefore, it can represent domains like

$$\textit{if}\; b_0 \;\textit{then}\; (\textit{if}\; b_1 \;\textit{then}\; \ldots \;\textit{else}\; \ldots) \;\textit{else}\; ((\textit{if}\; b_2 \;\textit{then}\; \ldots \;\textit{else}\; \ldots))$$

The conditional domain also propagates to floating-point computations.

The initial code contains integer and floating-point values. Our automatic instrumentation (see Sect. 3.1) preserves floating-point constants but replaces floating-point variables with the default floating-point domain containing an affine form for the ideal computation and the accumulation of round-off errors, and an interval for the floating-point computations. Thus, each new domain potentially interacts with 3 different domains (conditional integer, conditional floating-point, affine forms) and the concepts of C++-20 are very useful for handling these interactions — adding the conditional domains to FLDLib required 12 kloc of C++ code.

A finalization of these new domains and their application for the robustness analysis of the case study is still ongoing. It will require a manual instrumentation of the code (that will probably take more than 8 hours) but can be expected to help analyze the target code.

Robustness analysis is a mandatory requirement before attempting to verify functional properties, such as the relative independence of the results with respect to the size of the grid (cf. RQ3). Checking these properties follows the same methodology as checking robustness. We proceed first with simple tests, then with formal analysis. We start by using the same test case, before attempting later a modular verification approach based on deductive methods.

4 Conclusion

Numerical analysis of software is important for critical programs, in particular related to trajectory computation used in autonomous systems. It is also a very challenging and time-consuming task. Indeed, precision and robustness of the algorithm can be impacted by many instructions, especially for programs with long execution paths and/or simulating a continuous real space by a discrete grid, for which a small perturbation of data can naturally lead to another behavior.

[8]This domain is not yet available in the public repository of FLDLib, but we plan to integrate it into the open-source repository of the tool in the near future.

This case study paper describes an industrial application of several modern numerical analysis tools to a real-life path computation algorithm with a realistic test case. We present the applied methodology and results. An important first step of the study is to ensure code instrumentability and to compare various analysis results to qualify the impact of unstable branches with stochastic methods (with tools like Cadna and Verrou). Next, we investigate unstable branches and formally ensure robustness with a formal analysis (using a tool like FLDLib). The results we obtained seem very promising: we managed to identify the unstable branches and the corresponding locations on the code that constitute important attention points for numerical analysis. Dynamic analysis tools (Cadna and Verrou) show that the relative error in the path computation is sufficiently small, and the algorithm is sufficiently robust. This conclusion should be confirmed by a formal analysis. A representative subset of unstable branches coming from different parts of the algorithm has been formally shown (with FLDLib) to ensure expected robustness properties, while the study for other branches is still in progress. So far, the analysis confirmed that the algorithm meets the user expectations in terms of accuracy and robustness.

Future Work. This case study suggests numerous future work perspectives. One perspective is to finalize the investigation of unstable branches. We plan to use the new conditional domains that were recently integrated into FLDLib and will be evaluated on this case study. Considering other realistic test cases and replaying the analyses for them is another work direction. Applying the described methodology on other industrial use cases is another perspective.

As a more ambitious long-term research objective, proving that the result does not depend on the size of the grid (RQ3) is a much more complex problem. Our plan is to apply a component-based divide-and-conquer approach on the source code. For each component, this requires formal instrumentation in order to propagate logical formulas instead of abstract domains. The starting point is the previous instrumentation of the code with its annotations for the synchronization of unstable branches. The methodology is inspired by the approach used in deductive verification, by first replacing the data structures of the code with classes representing formal properties. C++ operator overloading will propagate these properties across components using carefully designed verification unit scenarios. The engineer's objective will be to design unit scenarios (such as the postcondition/output invariant of a method/class is formally contained in the precondition/input invariant of the method/class that takes its results).

Acknowledgment. Part of this research (tooling improvement) was supported by the ANR InterFLOP project (grant ANR-20-CE46-0009). Part of this work was also supported by the ANR EMASS project (grant ANR-22-CE39-0014). We thank the designers and developers of numeric analysis tools of the InterFLOP community.

References

[1] R. Bellman & R. Kalaba (1965): *Dynamic Programming and Modern Control Theory*. Academic paperbacks, Elsevier Science.

[2] Elisabetta Carlini, Maurizio Falcone, Nicolas Forcadel & Régis Monneau (2008): *Convergence of a generalized fast marching method for a non-convex eikonal equation*. SIAM Journal on Numerical Analysis 46, pp. 2920–2952, doi:10.1137/06067403X.

[3] Da Chen, Jian Zhu, Xinxin Zhang, Minglei Shu & Laurent D. Cohen (2021): *Geodesic Paths for Image Segmentation With Implicit Region-Based Homogeneity Enhancement*. IEEE Transactions on Image Processing 30, pp. 5138–5153, doi:10.1109/TIP.2021.3078106.

[4] Michael G. Crandall, Hitoshi Ishii & Pierre-Louis Lions (1992): *user's guide to viscosity solutions of second order partial differential equations*. Available at `https://arxiv.org/abs/math/9207212`.

[5] Christophe Denis, Pablo de Oliveira Castro & Eric Petit (2016): *Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic*. In: *Symposium on Computer Arithmetic (ARITH)*, doi:10.1109/ARITH.2016.31.

[6] François Févotte & Bruno Lathuilière (2019): *Debugging and Optimization of HPC Programs with the Verrou Tool*. In Ignacio Laguna & Cindy Rubio-González, editors: *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, IEEE, pp. 1–10, doi:10.1109/CORRECTNESS49594.2019.00006.

[7] Fabienne Jézéquel & Jean Marie Chesneaux (2008): *CADNA: a library for estimating round-off error propagation*. *Comput. Phys. Commun.* 178(12), pp. 933–955, doi:10.1016/J.CPC.2008.02.003.

[8] R. Kimmel & J. A. Sethian (1998): *Computing geodesic paths on manifolds*. *Proceedings of the National Academy of Sciences of the United States of America* 95(15), pp. 8431–8435, doi:10.1073/pnas.95.15.8431.

[9] Jean-Marie Mirebeau, Lionel Gayraud, Rémi Barrère, Da Chen & François Desquilbet (2023): *Massively parallel computation of globally optimal shortest paths with curvature penalization*. *Concurrency and Computation: Practice and Experience* 35(2), p. e7472, doi:10.1002/cpe.7472.

[10] J. A. Sethian (1996): *A Fast Marching Level Set Method for Monotonically Advancing Fronts*. *Proceedings of the National Academy of Sciences of the United States of America* 93(4), pp. 1591–1595, doi:10.1073/pnas.93.4.1591.

[11] J.A. Sethian (2001): *Evolution, Implementation, and Application of Level Set and Fast Marching Methods for Advancing Fronts*. *Journal of Computational Physics* 169(2), pp. 503–555, doi:10.1006/jcph.2000.6657.

[12] Franck Védrine, Maxime Jacquemin, Nikolai Kosmatov & Julien Signoles (2021): *Runtime Abstract Interpretation for Numerical Accuracy and Robustness*. In Fritz Henglein, Sharon Shoham & Yakir Vizel, editors: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, *Lecture Notes in Computer Science* 12597, Springer, pp. 243–266, doi:10.1007/978-3-030-67067-2_12.

[13] J. Vignes (1993): *A stochastic arithmetic for reliable scientific computation*. *Mathematics and Computers in Simulation* 35(3), pp. 233–261, doi:10.1016/0378-4754(93)90003-D.

[14] Wikipedia: *Substitution failure is not an error*. Available at `https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error`.