ImatiSTL - Fast and Reliable Mesh Processing with a Hybrid Kernel

Marco Attene(✉)

IMATI CNR, Genova, Italy
marco.attene@ge.imati.cnr.it

Abstract. A novel approach is presented to deal with geometric computations while joining the efficiency of floating point representations with the robustness of exact arithmetic. Our approach is based on a hybrid geometric kernel where a floating point number is made fully interoperable with an exact rational number, so that the latter can be used only within critical parts of the program or within restricted portions of the input. The whole program can dynamically change the level of precision used to produce new values and to evaluate expressions. Around such a kernel, a mesh processing library has been implemented whose API functions can be classified depending on their precision as always exact, always approximated, or exact if the current level of precision is sufficient. Such a classification allows implementing algorithms with a full control of the robustness at an unprecedented level of granularity. Experiments show that this interoperability comes at a nearly negligible cost: on average, a test algorithm implemented on our hybrid kernel is just 8% slower than the same algorithm implemented on a *standard* floating point version of the same kernel while providing the possibility to be fully robust if necessary.

1 Introduction

Geometry processing involves a switch from the mathematical to the computational world where many developers simply approximate real numbers with floating point (FP) representations [1]. Most of the times this approach is accurate enough and efficient, and that is why numerous libraries and algorithms to perform geometric computations use FP numbers [2,3] to represent coordinates, distances, angles, etc. Unfortunately, FP operations are subject to roundoff error, and in some cases the result of a computation may become useless due to such a mismatch. Implementations that neglect this observation can be subject to failures, infinite loops, and crashes.

1.1 Exact and Multi-precision Arithmetic

To ensure that all the computations lead to exact results one may rely on exact or multi-precision arithmetic libraries [4–6]. This solution is extremely robust, and in most cases the available memory is the only limitation. Unfortunately, working

M.L. Gavrilova and C.J. Kenneth Tan (Eds.): Trans. on Comput. Sci. XXIX, LNCS 10220, pp. 86–96, 2017.
DOI: 10.1007/978-3-662-54563-8_5

with exact representations has a significant impact on the performances, and a program might become even twenty times slower than its corresponding FP-based version [7]. This makes this solution unpractical in many cases, especially when large datasets must be elaborated with guarantees.

1.2 Arithmetic Filtering

The correctness of some algorithms is solely based on the exact evaluation of a specific predicate that can assume one of a small set of values (e.g. true/false, $1/0/-1$, ...). If the arithmetic expression that leads to the predicate's value is subject to roundoff, one can rely on the so-called "filtered arithmetic" approach. Roughly speaking, the roundoff's potential magnitude is assessed, and only if it is sufficient to make the final result switch from one value to another (e.g. from true to false or vice versa), then the predicate is evaluated using a more accurate though slower approach. In several practical cases, FP numbers are accurate enough for the vast majority of the predicate evaluations, thus this approach combines the efficiency of FP computations with the robustness of exact arithmetic. A noticeable example of this technique is due to Shewchuk [2], who introduced fast and robust predicates to evaluate the incircle test needed to compute Delaunay triangulations. As a rule of thumb, the filtered arithmetic approach is appropriate when all the predicate expressions are direct functions of input values.

1.3 Lazy Evaluation

For some algorithms, however, a predicate might be necessarily a function of some intermediate values. If these values incorporate a roundoff error, the predicates's evaluation cannot be guaranteed to be correct even when using filtered arithmetic. To cope with these cases one can still revert to exact arithmetic and compute intermediate values without error but, once again, this easily leads to unacceptably slow implementations. Alternatively, instead of representing intermediate values explicitly, one may encode the symbolic expressions to be used for their evaluation. In other words, an intermediate value can be encoded within a Direct Acyclic Graph (DAG) representing a specific function of input values. When a predicate must be evaluated, such an expression is combined with the expression of the predicate itself, and filtered arithmetic can be used to exactly derive the result. Due to this inherent procrastination, this approach is usually called "lazy evaluation". Lazy evaluation is particularly useful to compute mesh booleans, where new points representing the intersection of edges and triangles are used to derive the result [8]. This approach is typically combined with filtered arithmetic: a DAG representing the number's exact history is coupled with an interval containing its exact value and an FP number within that interval. When a number is used in a predicate expression, its interval is exploited to perform the filtering: if the test passes, the predicate is evaluated using FP arithmetic, otherwise the evaluation is done based on the DAGs. The main drawback of this approach is due to the need of the DAGS: these structures, indeed, have

a relatively large memory footprint and their update and management have an inevitable impact on the performances.

1.4 Mixed Techniques

One of the most diffused forms of exact arithmetic uses rational numbers with arbitrarily large numerator and denominator [5]. Since this solution is only suitable to model problems where no irrational numbers are involved, in some existing libraries [6,7] values derived from irrational operations (e.g. square root) are stored in symbolic form as done in the lazy evaluation approach. Clearly, the evaluation of expressions that involve such "irrational" numbers might become slower. In the scope of this paper, however, we shall not deal with irrational expressions.

1.5 CGAL

The Computational Geometry Algorithm Library (CGAL) is one of the few existing tools which includes all the aforementioned techniques to deal with robustness, and can be considered as a representative of the state of the art in robust geometric computing. In particular, CGAL provides a special templated number type called Lazy_exact_nt<NT> to implement the aforementioned lazy evaluation on a basic number type NT [9]. In essence, approximated values are used instead of NT as long as possible, but the computational history is maintained within the DAG and evaluated if needed. Hence, when NT is a slow exact type this approach can significantly speed up the computation.

For more comprehensive overviews of geometric robustness and related issues, see Yap [10] and Goldberg [11].

1.6 Key Contribution

Existing libraries allow developers to choose among various number types and computational kernels. For example, if the program needs guarantees about the relative position of projected points, exact coordinates must be used; if the program must just visualize a mesh, FP numbers are sufficient. But what if the program needs to do both? Typically, the number type is chosen once for the entire program based on the maximum precision required. If such a maximum precision is required only by a small percentage of the operations, most of the program is unnecessarily slowed down and memory-intensive, even if lazily-evaluated types are used.

Conversely, in ImatiSTL the developer may freely switch from one kernel to another while being guaranteed that all the numbers remain compatible and interoperable with each other, with significant advantages in terms of both speed and memory consumption as shown in Sect. 4. This result could be achieved thanks to the definition and implementation of a novel *hybrid* number type, whose advantages with respect to the state of the art (i.e. CGAL) are described in the following Sect. 2.

2 Hybrid Number Type

CGAL's lazy evaluation and filtered arithmetic must determine an interval containing the exact result of an expression [9]. However, computing such an interval has its own cost, and this makes the approach really beneficial only when (1) there is an actual need of exact evaluations and (2) such a need is rare with respect to the total amount of the expressions to be evaluated. Based on these observations, a developer must determine whether his/her algorithm is worth to be implemented on a lazily-evaluated number type and, if so, the program must be configured accordingly. To do this, existing libraries such as CGAL require specifying a number type and a computational kernel, and dynamically changing these settings while the program executes is quite unpractical (i.e. all the numbers involved should be explicitly converted to the new type, which has an impact on both development effort and program efficiency). Therefore, though a developer might know exactly where the program requires exactness, both in terms of computational flow and in terms of input data, such an information can be hardly exploited. To overcome this limitation, in the remainder we define a novel polymorphic number type that has virtually the same performances of a standard double precision floating point, but can encode either an actual floating point or an exact number. Differently from CGAL's Lazy_exact_nt, the user can explicitly control the need of exactness when operating with our polymorphic numbers: this makes it possible to avoid unnecessary interval computations and checks to speed up the program when approximated results are known to be enough.

2.1 Terminology and Definition

In the remainder, a standard IEEE double precision floating point number is shortly called an "FP number". An FP number has an encoding and a value: the former is just a fixed-sized sequence of bits, while the latter is an element of the set of rational numbers Q. According to IEEE 754 standard specifications, any encoding corresponds to a unique value, though the vice-versa is not necessarily true (e.g. the encoding for the value 0.25 changes if the number is represented as $25 * 10^{-2}$ or as $250 * 10^{-3}$). FP numbers can encode a finite subset of the rational numbers. However, since rational numbers form an enumerable set, it is reasonable to look for a more comprehensive encoding. This observation led to the development of libraries such as GMP and LEDA, where any rational value is encoded as a pair of arbitrarily large integers representing the numerator and denominator.

Herewith a new number type called *PM_Rational* is introduced whose encoding is inherently polymorphic. Any rational number can be represented as a PM_Rational (up to memory limits), but its encoding might be either an FP number or a pair numerator/denominator. Independently of the actual encoding of the operands, arithmetic operations may be either exact or subject to roundoff, and the developer has the possibility to control this level of precision at any time. Essentially, the user acts on a global parameter that determines the precision

level of the PM_Rational operations. Three levels are available: *approximated*, *filtered*, *precise*. When the *approximated* mode is active, all the expressions on PM_Rationals are computed just as if they were FP numbers with virtually no performance degradation with respect to native IEEE double precision arithmetic: in this mode predicates might assume a wrong value. In *filtered* mode, PM_Rationals still behave as FP numbers, but predicates are evaluated using arithmetic filtering: in this mode predicates are guaranteed to assume the correct value, but expressions that produce other PM_Rationals might still lead to approximated evaluations: in other words, intermediate PM_Rational values are not guaranteed to be exact. Finally, in *precise* mode, all the predicates and rational expressions are guaranteed to be exactly evaluated.

Thanks to this paradigm, a program can load a geometric model such as a polygonal mesh and encode the vertex coordinates as PM_Rationals. This same mesh can be used for different processes with different precision levels without the need to perform explicit type conversions. For example, an *approximated* mode can be employed to render the mesh using backface culling. For this operation, indeed, it is reasonable to accept wrong orientations for triangles whose normal vector is nearly orthogonal to the line of sight. Then, the program can switch to *filtered* mode to perform *point_in_polyhedron* queries with a guaranteed correct result. A final switch to *precise* mode allows to exactly calculate and represent the intersection of the mesh with another model.

2.2 Implementation

Internally, the PM_Rational type has been implemented in C++ as a class containing one 64bit-sized generic *data* member, and one boolean *type* member that specifies what the *data* member encodes. In particular, the *data* member can encode either a standard IEEE double precision number (*type* = double) or a pointer to a pair numerator/denominator (*type* = rational). In its turn, a pair numerator/denominator is encoded as an *mpq_class* defined within the C++ interface to the GMP library. Note that the type of a PM_Rational number is independent of the global precision mode that the program employs at any time. The latter is encoded as a public static member called $Kernel\_mode$ that determines the current precision level to be used in the PM_Rational computations. $PM\_Rational :: Kernel\_mode$ is essentially a global variable that the user can change at any time.

All the arithmetic and comparison operators are defined on PM_Rationals. For arithmetic operators (i.e. $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$), the current $Kernel\_mode$ is used to produce the result, independently of the type of the operands. Hence, the operation $A + B$ returns a PM_Rational whose type is double if the current $Kernel\_mode$ is either *approximated* or *filtered*, whereas the resulting type is rational if the mode is *precise*. If necessary, the operands are transparently converted to the type of the result before calling the corresponding native operator. Conversely, for comparison operators (i.e. $==$, $<$, $>$, $<=$, $>=$) the *type* of the operands is used to determine the result. If both the operands have the same *type* the native comparison operator for such a *type* is

used. If they have different *type*, the operand having a double *type* is converted to rational and the native comparison for rationals is used.

3 ImatiSTL

A mesh processing library called ImatiSTL has been implemented to exploit the PM_Rational numbers. This library provides an API whose functions can be classified as follows:

- *Always exact* - the return value (or the processing result) is guaranteed to be reliable in any kernel mode. These functions include, e.g., coordinate comparison, vector inversion, operations on the connectivity graph.
- *Exact if kernel mode is filtered* - the return value (or the processing result) is guaranteed to be reliable only if kernel mode is at least set to $filtered$. Functions of this type normally involve orientation predicates (e.g. Delaunay triangulation of a 2D point set).
- *Exact if kernel mode is precise* - the return value (or the processing result) is guaranteed to be reliable only if kernel mode is set to *precise*. Functions of this type might involve intermediate values that influence the flow of the computation (e.g. relative position of projected points, intersections).
- *Always approximated* - the result of the function (or one of its intermediate values) does not necessarily belong to the set of rational numbers. Functions of this type normally involve Euclidean distances, angles, or other irrational quantities. Note that squared distances are implemented within the aforementioned class *Exact if kernel mode is precise*.

A set of geometric predicates has been implemented in ImatiSTL to exploit the inherent type polymorphism provided by PM_Rational coordinates. Any such predicate is a $friend$ function of PM_Rational and proceeds to an appropriate computation depending both on the current $Kernel\_mode$ and on the *type* of the operands. Friendship is required because the predicates need to access the *type* member which is not part of the public interface. For example, the typical 2D orientation predicate is implemented as in Algorithm 1:

Note that all of this is transparent to the developer who is only required to change the kernel mode when necessary.

4 Results and Discussion

To test the actual behavior of the hybrid kernel, three different versions of the same algorithm have been implemented. The test algorithm creates a tetrahedron (Fig. 1(a)), performs five steps of Loop subdivision on it [12] (Fig. 1(b)), creates a copy of the so-subdivided tetrahedron and shifts it along the positive X axis (Fig. 1(c)), and calculates the outer hull of the resulting pair of intersecting models [8] (Fig. 1(d)). In the approximated version, the test algorithm uses an implementation of ImatiSTL where traditional double precision

Algorithm 1. Implementation of the planar orientation predicate. On lines 11–13 *Kernel_mode* is *filtered* but at least one operand is *rational*.

Require: Three 2D points represented as pairs of PM_Rationals $P = (p_x, p_y)$, $Q = (q_x, q_y)$, $R = (r_x, r_y)$
Ensure: CCW, ALIGNED, CW, depending on the relative orientation of R wrt P and Q

```
 1: PM_Rational O; // Temporary value to determine the predicate's output
 2: if Kernel_mode is approximated then
 3:    O = ((p_x − r_x) ∗ (q_y − r_y) − (p_y − r_y) ∗ (q_x − r_x))
 4: else
 5:    if all the operands are of type double then
 6:       Compute O using filtered arithmetic as done in [2]
 7:    else
 8:       if Kernel_mode is precise then
 9:          O = ((p_x − r_x) ∗ (q_y − r_y) − (p_y − r_y) ∗ (q_x − r_x))
10:       else
11:          Temporarily switch to precise Kernel_mode
12:          O = ((p_x − r_x) ∗ (q_y − r_y) − (p_y − r_y) ∗ (q_x − r_x))
13:          Swicth back to filtered Kernel_mode
14:       end if
15:    end if
16: end if
17: if O>0 then
18:    return CCW
19: else
20:    if O==0 then
21:       return ALIGNED
22:    else
23:       return CW
24:    end if
25: end if
```

numbers are used to represent the coordinates. Similarly, in the exact version $CGAL :: Lazy\_exact\_nt < CGAL :: Gmpq >$ was used to represent the coordinates. In the hybrid version, PM_Rational was used instead. The following three indicators were measured: number of source code lines used to implement the test program (ImatiSTL library not included in the count); elapsed time; memory footprint.

Not surprisingly, the last phase of the algorithm fails when using doubles: the outer hull computation, indeed, relies on the relative position of intersection points [13]. Hence, the time and memory evaluations are split in two parts, one regarding the algorithm without the outer hull computation, and one that includes this last phase. Quantitative results of this experiment are summarized in Table 1.

Table 1 reveals that the coding effort required to fully exploit the potential of the hybrid kernel is extremely limited. Indeed, with respect to the approximated version, the developer is just required to add an instruction at the beginning

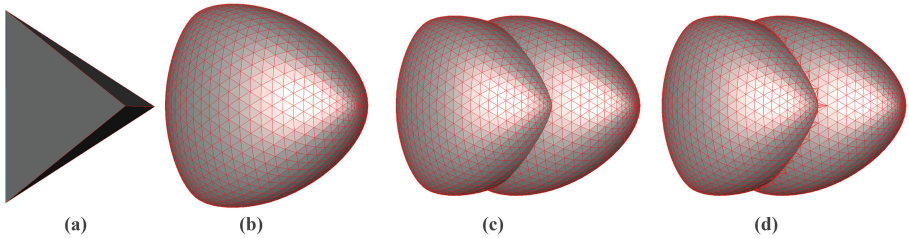

Fig. 1. The four phases of our test algorithm. An initial tetrahedron (a) is refined through five Loop subdivision steps (b). The resulting solid is duplicated, and one of the two copies is shifted along the X axis (c). The outer hull of the resulting two intersecting solids is computed (d).

Table 1. Quantitative comparison of three versions of the same algorithm showing the advantages of the hybrid approach proposed.

	Approximated	Exact	Hybrid
Num. code lines	2695	2701	2697
Time (no outer hull)	15 ms	81 ms	16 ms
Time (with outer hull)	n.a	2208 ms	413 ms
Memory (no outer hull)	9.2 MB	22.1 MB	9.9 MB
Memory (with outer hull)	n.a	58.4 MB	47.1 MB

(PM_Rational::$Kernel\_mode$ = *approximated*) and one before the outer hull computation routine (PM_Rational::$Kernel\_mode$ = *precise*). The additional code lines in the exact version of the test program are the various typedefs and kernel initializations needed by CGAL. Also, with respect to the approximated version, the increase in the elapsed time due to the use of PM_Rationals is negligible, whereas the need to check whether the approximated numbers would lead to unreliable results makes CGAL quite slower. Also, since the underlying *type* of PM_Rationals is double for the vast majority of the coordinates (i.e. all the vertices but those that represent intersection points), the memory footprint is not affected too much. Conversely, the use of CGAL's interval arithmetic has a much more significant impact in this sense due to the use of DAGs for all the points.

Similar experiments on more than a hundred mesh models demonstrate that, when there is no need to perform exact computations, using PM_Rationals instead of standard doubles slows the computation down of a factor of 8% on average. As far as the memory footprint is concerned, using PM_Rationals requires 12% more resources on average: this is mostly due to a typical behavior of compilers and of operating systems which allocate at least one byte for each class member. Thus, even if in principle 65 bits would be sufficient to encode a PM_Rational whose underlying *type* is double, 72 bits are used by the operating system due to such an alignment.

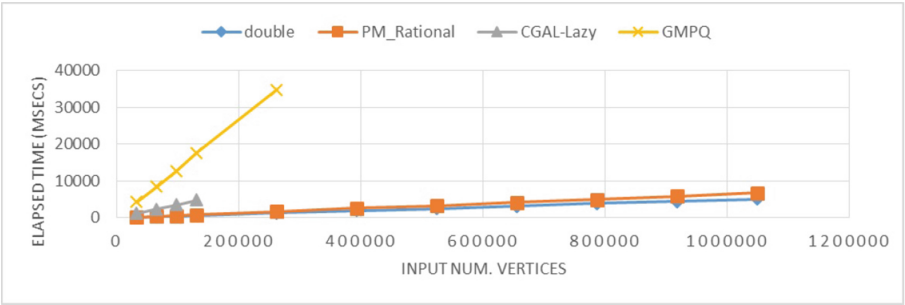

Fig. 2. Time necessary to perform a single Loop subdivision step. CGAL-based implementation crashes when the input exceeds 200 K vertices. This threshold is slightly higher for the GMPQ-based implementation.

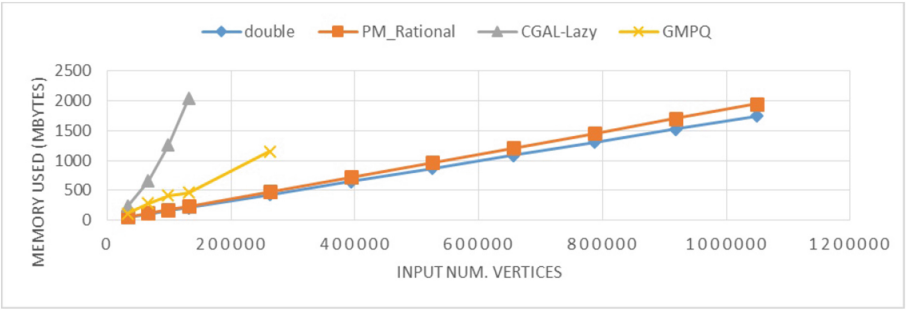

Fig. 3. Memory required to perform a single Loop subdivision step. CGAL-based implementation crashes when the input exceeds 200 K vertices. This threshold is slightly higher for the GMPQ-based implementation.

Analogous results were achieved during a further experiment where the program had to just perform a single Loop subdivision step on an input mesh made of N vertices. For this experiment, the same program was implemented using four different types to represent the coordinates: standard IEEE double precision FP numbers, PM_Rational, GMPQ, CGAL::Lazy_exact_nt<CGAL::Gmpq>. Figures 2 and 3 depict a comparison of the time and memory performances respectively as the number of input vertices increases. Note that the two versions based on GMPQ and CGAL fail much earlier as the input size grows.

4.1 Limitation

Non-rational numbers are not representable as PM_Rationals. Competing libraries such as, e.g., LEDA or CGAL, provide tools to represent a subset of these numbers: for example, algebraic numbers are somewhat useful in geometric computation and can be represented "symbolically" within LEDA. Unfortunately, this limitation for PM_Rationals is intrinsic and apparently can be solved

only through an integration with a symbolic calculus module which would probably spoil the gain in performances. Hence, this is an open problem that represents an interesting direction for future research.

5 Summary and Conclusions

In this paper, the need for robust arithmetic in mesh processing algorithms has been re-casted to a novel paradigm where the user has a full control of the precision, in any part of the program and for any portion of the input dataset. Experiments demonstrated that this approach is more efficient than state-of-the-art solutions, in terms of both memory consumption and speed of the computation.

Clearly, the developer must know where and how to deal with robustness to fully exploit the potential of this new approach, but a specific algorithmic design is necessary in any case to guarantee robustness even with existing libraries (e.g. CGAL). Further research is still necessary to make development as easy as for traditional floating point arithmetic. A naive approach is to use the precise kernel for PM_Rationals everywhere, but this would make the program too slow in general. As an alternative, the developer may use as much precision as necessary just to make sure that ImatiSTL API functions return a correct result. In principle, the switch to the necessary precision level can be made automatically, but this makes sense only as long as the user is forced to use the provided API functions only. Unfortunately, preventing the user to insert custom PM_Rational operations would be probably a too strict constraint for a flexible development. A really effective solution to the problem would probably require an automated analysis of the source code.

Acknowledgements. This work has been partly supported by the international joint project on Mesh Repairing for 3D Printing Applications funded by Software Architects Inc (WA, USA). Thanks are due to the SMG members at IMATI for helpful discussions.

References

1. Shewchuk, J.R.: Lecture notes on geometric robustness. University of California at Berkeley (2013)
2. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discr. Comput. Geometry **18**, 305–363 (1997)
3. Visual Computing Lab: Vcglib: visualization and computer graphics library. http://vcg.sourceforge.net
4. RWTH: Openmesh: visualization and computer graphics library. http://www.openmesh.org/
5. Granlund, T.: The GNU multiple precision arithmetic library. TMG Datakonsult, Boston, MA, USA (1996)
6. Karamcheti, V., Li, C., Pechtchanski, I., Yap, C.: A core library for robust numeric and geometric computation. In: Proceedings of the Fifteenth Annual Symposium on Computational Geometry, pp. 351–359. ACM (1999)

7. Mehlhorn, K., Naher, S.: Leda: a platform for combinatorial and geometric computing. Commun. ACM **38**, 96–103 (1995)
8. Attene, M.: Direct repair of self-intersecting meshes. Graph. Models **76**, 658–668 (2014)
9. Pion, S., Fabri, A.: A generic lazy evaluation scheme for exact geometric computations. Sci. Comput. Program. **76**, 307–323 (2011)
10. Yap, C.: Robust geometric computation. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn. CRC Press, LLC, Boca Raton (2004)
11. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**, 5–48 (1991)
12. Loop, C.: Smooth subdivision surfaces based on triangles. Department of Mathematics, The University of Utah, Master Thesis (1987)
13. Campen, M., Attene, M., Kobbelt, L.: A practical guide to polygon mesh repairing. In: EUROGRAPHICS Tutorials, Eurographics, May 2012