A Core Library For Robust Numeric and Geometric Computation

V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap
Courant Institute of Mathematical Sciences
New York University

Abstract

Nonrobustness is a well-known problem in many areas of computational science. Until now, robustness techniques and the construction of robust algorithms have been the province of experts in this field of research. We describe a new `C/C++` library (CORE) for robust numeric and geometric computation based on the principles of Exact Geometric Computation (EGC). Through our library, for the first time, any programmer can write robust and efficient algorithms. The Core Library is based on a novel numerical core that is powerful enough to support EGC for algebraic problems. This is coupled with a simple delivery mechanism which transparently extends conventional `C/C++` programs into robust codes. We are currently addressing efficiency issues in our library: (a) at the compiler and language level, (b) at the level of incorporating EGC techniques, as well as the (c) the system integration of both (a) and (b). Pilot experimental results are described. The basic library is available at `http://cs.nyu.edu/exact/core/` and the `C++-to-C` compiler is under development.

1 INTRODUCTION

Numerical non-robustness is well-known in many areas of computational sciences and engineering [7]. Non-robustness in this paper[1] refers to what is sometimes known as "catastrophic errors": errors that cause programs to enter unanticipated states and hence crash. In applications areas such as physical simulation and geometric modeling and design, the underlying geometry grows increasingly complicated, and non-linear models are increasingly used. Both these trends imply an acute need for solving non-robustness in a systematic and scientifically sound way. Although there have been many research efforts to create robust algorithms, such solutions are rarely used in practice: *ad hoc epsilon-tweaking rules remain the mainstay of practitioners.* Without going into the relative merits of the various proposed solutions, there is one overriding reason for this unfortunate state of affairs: most previous solutions apparently require programmers to change their programming behavior, sometimes in radical ways. It is one thing to demonstrate that a technique that can produce a robust algorithm for a particular problem. But it may be another problem when potential users need to (i) modify the technique for their particular requirements, or (ii) extend it to related problems. Robust solutions, especially those based[2] on "fixed-precision geometry", are particularly resistant to (i) and (ii). See [33] for a survey of robustness literature.

In this paper, we describe the **Core Library** (CORE for short), a new `C/C++` library for robust numeric and geometric computation. Our library API ("application programmer interface") model, as first proposed in [32], is able to deliver powerful robustness techniques in a relatively transparent manner. Thus, to construct a stand-alone robust algorithm, the CORE API allows the programmer to code the algorithm without explicitly worrying about robustness. We do require the programmer to design his or her algorithm assuming exact real arithmetic in the standard Euclidean geometry. But this is almost no requirement at all, as Euclidean geometry is the default model of most geometric concepts in practice. This avoids, for instance, the many highly unintuitive surprises when researchers use fixed-precision geometries. The programmer may pay only minimal attention to our library, only making sure that the final `C/C++` program contains a short (e.g., 2-line) preamble

[1] We are only interested in catastrophic errors that arise from numerical approximations. The computing literature often refers to preformance issues such as scalability of algorithms as "robustness issues". Such issues are also outside our scope.

[2] These are the "geometries" that researchers need to define to approximate the standard Euclidean geometry when they operate in fixed-precision arithmetic.

to invoke our library. If this program is now compiled with a C++ compiler and linked with our library, this code will be robust. In general, such a library API allows us to convert most stand-alone programs into robust programs with minimal effort.

The reader familiar with object-oriented programming languages may suspect that our library will replace standard number types (such as `int` or `double`) by some bigNumber class. This is correct as far as it goes, but as we shall see, something fundamentally deeper is going on. To give a hint of this, it may be noted that the "preambled" code is robust even if it involves non-rational operations such as square roots. No bigNumber package alone can ensure this behavior.

Our library supports the **Exact Geometric Computation** (EGC) approach [34] to robustness (although a user is free to ignore EGC and use the CORE API and features for other purposes). As the name of the library suggests, the heart of our library is indeed a new "numerical core". Just as the floating point package (hardware) constitutes the heart of contemporary scientific computation, our "numerical core" can serve as the basis for EGC computing. In the EGC approach to robustness, the key efficiency principle is **precision-sensitivity** (cf. [35, 29]). An algorithm is "precision-sensitive" if its running time scales with the actual precision needed for each input instance. Precision sensitive techniques take many forms. One is "floating-point filters" which several groups [14, 10, 2, 5] have shown to be very effective. Indeed, exploiting precision-sensitivity is what distinguishes the current EGC approaches [14, 6, 5, 3, 30, 21] from earlier attempts to use "exact arithmetic". These early attempts (e.g., [23, 36]) inevitably fare badly against worse-case scenarios. Many of these EGC techniques will be available through our library.

The main theme in our current development of CORE is code efficiency. We attack this at the level of EGC-based techniques (e.g., floating point filters) as well as at the level of compiler optimization. The latter aims at applying aggressive optimization techniques to automatically produce robust code whose speed on "most inputs" is within a small constant factor of that achievable by hand-coded optimizations. As we will see (Section 4), this optimization research involves a rich interplay of EGC techniques and compiler optimizations (especially in the context of an object-oriented language such as C++).

The Core Library sources, with all the examples in this paper, are available at our website [19].

OVERVIEW OF PAPER. In the next section, we first give a user-viewpoint of the Core Library. In Section 3, we describe the internal view. In Section 4, we address compiler analysis and optimization issues. The preliminary experimental results are shown in Section 5. We conclude in Section 6.

2 A NOVEL NUMERICAL CORE

This section provides a user (API) view of CORE. The key ideas from [32] are (a) a novel and deceptively simple proposal for a "number core" based on 4 accuracy levels, and (b) a transparent "delivery mechanism" to bring this core capability to users. First we describe the four levels of **core (numerical) accuracies**:

- **LEVEL I: Machine Accuracy.** This may be identified with the IEEE-standard [26].
- **LEVEL II: Arbitrary Accuracy.** Users can specify any desired accuracy. E.g., "200 bits" means that numerical operations will not cause an overflow or underflow until 200 bits are exceeded. This feature is widely available in modern day computer algebra systems such as Maple or Mathematica.
- **LEVEL III: Guaranteed Accuracy.** This is the most interesting level: specifying "200 relative bits" means that the first 200 significant bits of a computed quantity are correct. The `Real/Expr` package [35, 24] is the first to achieve Level III accuracy for a non-rational family of expressions.
- **LEVEL IV: Mixed Accuracy.** The previous accuracy levels are intermixed and localized to individual variables, allowing finer accuracy control. This has not been implemented.

We next describe the mechanism for delivering these core accuracies to the user. A normal C/C++ program only have to be preceded by a suitable **preamble**. The simplest preamble is:

```
#define Level N       /* N=1,2,3 or 4 */
#include "CORE.h"
```

The program is then compiled in the usual way, although its behavior now depends on the chosen accuracy level. In other words, a single program can "simultaneously" access the different accuracy levels literally at the flip of a switch (viz., setting a variable). Fine tuning of accuracy via the setting of precision for individual variables is possible. Our library supports the setting of precision in both relative and absolute (or a mixture thereof [35]) terms. In the following, we explain what our library framework means for the user, and how it is can be used to achieve robustness.

ILLUSTRATIVE EXAMPLES. We currently have a prototype of CORE in which the first three accuracy levels can be accessed. This is basically achieved by constructing a wrapper around the `Real/Expr` package For brevity, we only discuss Levels I and III in the examples here.

- We wrote a basic geometry package. This allows us to construct a plane P with equation $x+y+z = 1$ and to intersect P with the lines L_{ij} $(i, j = 1, \ldots, 50)$ through the origin $(0, 0, 0)$ and the point $(i, j, 1)$. We then test if the intersection point $P_{ij} = L_{ij} \cap P$ lies on the plane P. When run at Level III, the answer is positive in all 2500 cases. At Level I, the answer is correct in 1538 cases (62.5%).
- Consider now the relationship between two lines in 3-space. We wrote three predicates `isSkew`, `isParallel` and `intersects`. Any pair of distinct lines ought to make *exactly one* of these predicates true. Let L_{ij} $(i, j = 1, \ldots, 50)$ be the line through the origin and $(i, j, 1)$ and let L'_{ij} be the line through $(1, 1, 0)$ and $(i+1, j+1, 1)$. At Level III, these (i, j) line-pairs are parallel in all 2500 cases, none skew or intersecting. At Level I, we get 2500 parallel pairs, but it also reported 1201 intersections and none skew. If we replace i and j by $sqrt(i)$ and $sqrt(j)$ in the same experiment, then Level III is again perfect while Level I reported 2116 pairs parallel, 378 skew and 1023 intersecting.
- We wrote a matrix package that included a straightforward Gaussian elimination algorithm (without pivoting) for computing determinants where the basic expression is

  ```
  A(j,k) -= A(j,i)*A(i,k)/A(i,i).
  ```

  This program will be further discussed in Section 4. Consider the following two matrices:

  ```
  double A[] = { 3.0,  0.0,  0.0,  1.0,
                 0.0,  3.0,  0.0,  1.0,
                 0.0,  0.0,  3.0,  1.0,
                 1.0,  1.0,  1.0,  1.0 };
  long B[]   = {  512,  512,  512, 1,
                  512, -512, -512, 1,
                 -512,  512, -512, 1,
                 -512, -512,  512, 1 };
  ```

  Level III correctly computes $\det(A) = 0$ and $\det(B) = 2^{31}$. However, at Level I, $\det(A)$ is non-zero because the Gaussian algorithm performs division by 3.0. Similarly, $\det(B)$ in Level I leads to an overflow (shows up as a negative number). Generally for determinants that vanish, no matter how convoluted the matrix entries (which may involve nested square roots), Level III never fails to detect 0. Similar behavior was also observed with Hilbert matrices.

HOW DO ACCURACY LEVELS SUPPORT ROBUST COMPUTING? Specifying "1 relative bit" in Level III amounts to guaranteeing the correct sign of computed values. From EGC theory, this ensures the exactness (and hence consistency) of our geometric structures. The more efficient Level II may suffice when we know *á priori* the needed precision, as in bounded-degree problems [34]. Even a speed-conscious user who cannot afford Level III in actual applications may use Level III accuracy to debug the program logic. *Note that this comes almost for free.* Conversely, as we have found, Level I is also useful for fast debugging of the non-numeric part of a program, even if we are ultimately interested in Level III.

WHAT IS THE DIFFERENCE BETWEEN LEVELS II AND III? Level III accuracy is the key innovation of CORE. Its distinction from Level II may not be obvious. Computer algebra systems deliver Level II accuracy. But specifying "500 bits of accuracy" does not mean that all the 500 bits in a quantity are significant – in fact, it is easy to lose every bit of significance. One of the authors (C.Y.) relates an experience with `Maple`'s Level II accuracy: as a particular computation is repeated with increased accuracy, the answers came back in a wildly unpredictable manner (including complex values when a real was expected). It was unclear whether this was a `Maple` bug or a programming error. On closer analysis, it turns out that the computation went through a singularity. This is a stroke of luck as it is usually tedious or infeasible to carry out such analysis. If the same computation could be carried out at Level III, this singularity would be automatically detected. Another qualitative difference is that Level III achieves **error-free comparisons** of any two reals, x and y. If $x \neq y$, then we could make this comparison in Level II, using a loop with increasing precision. But if $x = y$, then Level II is helpless: the loop is infinite.

HOW IS CORE DIFFERENT FROM OTHER ROBUST LIBRARY EFFORTS? The multi-institution European Community project `CGAL` [18, 25], and Max-Planck Institute of Computer Science project `LEDA` [22, 6] are two major libraries also committed to the EGC paradigm. Both aim at providing a comprehensive suite of efficient data structures and algorithms. The `CGAL` and `LEDA` efforts are very important and address real needs. Our approach is motivated by an orthogonal concern. We believe that no single library could possibly fulfill all the demands and complexities of potential applications. Rather, there is always a demand for small customized libraries in support of specialized applications. It is thus important to offer programmers the tools to achieve their robustness goals, which is precisely what CORE offers. Our work stresses the "small"[3] numerical core and supporting tools for constructing robust geometric software.

[3] Smallness is another quality that the name of our library is intended to evoke.

To support various applications, we prefer to define **core library extensions** (COREX) that embed domain specific knowledge. With our partners, we are currently building several such COREXs, including a COREX for mesh generation. An additional difference is our use of aggressive compiler techniques to minimize the amount of hand-tuning required for efficient implementation. NOTE: a simultaneous submission [4] by the `CGAL/LEDA` group to this conference describes an effort called `leda_real` which has many similarities to our work.

BENEFITS OF OUR APPROACH. Several benefits accrue to users of our library. (1) We already mentioned the advantages of working in an environment with access to different accuracy levels for debugging programming logic. (2) Another advantage is automatic technology transfer. As EGC technology improves, these improvement will be reflected in our library. Hence users of our library automatically enjoy the fruits of such advances. (3) Many applications must choose a trade-off between robustness and speed. Our accuracy levels greatly facilitate making or changing such choices.

3 HOW CORE WORKS

Since CORE is directly derived from the `Real/Expr` package they share many features. In particular, both use "precision-driven mechanisms" [35, 24] for expression evaluation, critical because in general, the worst case bounds in EGC are not sustainable. Both packages currently support the operators $+, -, \times, \div, \sqrt{\cdot}$, but in principle can be extended to any algebraic operation. The main difference between `Real/Expr` and CORE is in their semantics of assignments. In `Real/Expr`, the assignment "$a = b + c$" is really asserting a *permanent* relation (i.e., constraint) between three variables. This results in highly unintuitive behavior (e.g., *subsequent* assignments to b and c change the value of a). CORE removes all such surprises.

HOW DO THE VARIOUS LEVELS INTERACT? We provide more details on the delivery mechanism [32]. In the previous section, we have assumed the simplest situation, where the program is a standard `C/C++` program. We call this a **Level I program**. Thus, its primitive number types are basically `int`, `long`, `float`, `double`. These are the **Level I number types**. CORE defines the new[4] number types `Real` and `Expr`. These are (respectively) Levels II and III number types. Actually, Level II number types include `bigInt`, `bigRat`, or `bigFloat`. The number type `Real` is not a particular representation of numbers, but a superclass of all the number representations found in the system. There is a natural partial ordering $\prec$ among these types:

$$\texttt{int} \prec \texttt{long} \prec \texttt{bigInt} \prec \texttt{bigRat} \prec \texttt{Real},$$
$$\texttt{float} \prec \texttt{double} \prec \texttt{bigFloat} \prec \texttt{bigRat}$$

The automatic promotion or demotion of number types may occur during assignments, as in conventional languages. E.g., if we assign a `bigFloat` to an `int`, the value must be demoted before assignment. However, promotion and demotion of number types also occur when we set CORE accuracy levels. The most interesting case is when we run a Level I program at Level III: then `long` and `double` both promote to `Expr`; however, `int` and `float` remain unchanged. The motivation is that, even at Level III, it is desirable to have machine-precision variables for efficiency. As another example, if we run a Level III program at level II, then `Expr` demotes to `Real`. Here are the general principles:

1. A program is said to be **Level** ℓ (ℓ=I,II,III) if it explicitly declares number types of Level ℓ, but has no number types of Levels $> \ell$. This definition of ℓ is "static", independent of the preamble which sets the "run-time" accuracy Level. Note that there is no static Level IV.
2. Features accessible at run-time level ℓ are also available at run-time Level j ($\ell < j$).
3. Variables and features of Level ℓ will be demoted to corresponding variables and features at Level $j (j < \ell)$ when the program is run at Level j.
4. At run-time level IV, only assignments force promotion or demotion of variables.

WHAT IS IN LEVEL II ACCURACY. Consider the program fragment in figure 1(a). At Level II accuracy, variables m, p, q are promoted to type `Real` (internally, m would be `bigInt` while p, q are `bigFloat`). For efficiency, we may prefer to keep p, q as `double` for as long as possible. If so, we need a runtime check for overflows (and convert p or q to type `bigFloat` when it happens).

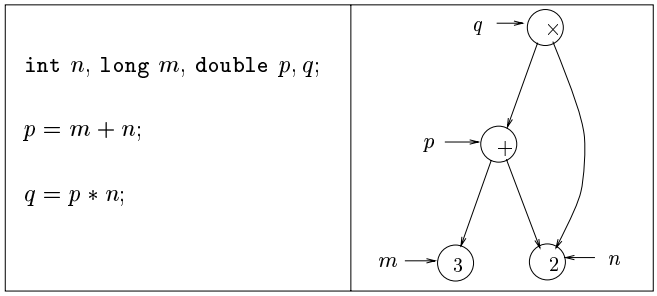

Figure 1: (a) Program Fragment. (b) Expression dags for p, q.

[4] Another Level II number type is `Complex` which we ignore in this paper for simplicity.

WHAT REALLY HAPPENS IN LEVEL III. If the above program fragment ($p = m+n; q = p*n$) were compiled with Level III accuracy, something completely different happens. As in `Real/Expr` [24], p is actually made to point to an expression corresponding to $m+n$ and similarly for q. See figure 1(b) where expressions are represented as directed acyclic graphs (dag's). That is, the dependence of p upon the values of m and n is remembered. In turn, the values of m, n may depend on other values. The reason for constructing such expressions is that we need to propagate the precision of p into precisions for m and n. There is a downward propagation of precision, upward propagation of errors, and this may be iterated until the error is less than the requested precision. Note the technical difference between *precision* and *error* in this mechanism. This is the **precision-driven mechanism** of [35]. We also need to maintain bounds on the (algebraic) heights and degrees of the algebraic quantities at each node of the dag.

LIBRARY STRUCTURE AND USAGE MODELS. The CORE library is written in `C++` using an object-oriented programming style which encapsulates underlying numerics from higher-level library and application routines and additionally supports extensibility. This library and its compilation infrastructure can be used in several modes. In Section 1, we described the mode of writing stand-alone CORE programs. One requirement was that exact real arithmetic must be assumed – it implies the programmer should avoid the temptation to manipulate bits in their numbers. Because `double` and `long` are promoted in Level III, they should also be careful in using standard `C` functions such as `scanf`. Note that some of these restrictions could be removed by providing CORE substitutions. In practice, the stand-alone mode is ill-suited for meeting the needs of existing application systems. For instance, we are constructing a Meshing COREX for `Card3d` [17], a meshing system for Computational Fluid Dynamics (CFD). This system contains both `FORTRAN` and `C` code. For minimal changes to such systems, we need rewrite only a handful of robustness-critical primitive functions. Since the CORE library is in `C++`, it is interoperable with languages such as `FORTRAN`. However, to take full advantage of the optimizations described in Section 4, the portion of the program that utilizes CORE must be recompiled using a special, but portable `C++`-to-`C` compiler. The compiler analyzes application usage patterns to improve the implementation of internal CORE library structures, so its benefits are proportional to the program portion that is recompiled.

4 EFFICIENT EXECUTION OF EGC PROGRAMS

The CORE library effectively addresses the robustness concerns in geometric computations. However, at higher accuracy levels (III and IV), this robustness often comes at the cost of efficiency. For example, anecdotal evidence [12, 20] shows that geometric primitives become slower by factors of up to 150 and 10,000, respectively, when exact integer computations and exact rational number computations are used instead of machine floating-point. While improved EGC techniques such as floating-point filters and precision-sensitive computation are capable of reducing these slowdowns, careful hand-tuning of implementations is required to get good performance. A novel aspect of our library is that it automates this tuning process, relying on an optimizing compiler to customize the implementation of internal CORE library structures based on an analysis of application usage patterns. In the rest of this section, we first identify the primary reasons for implementation inefficiency, and then describe our analysis and optimization approach. The next section describes pilot studies showing the performance advantages of this approach.

4.1 Sources of Overhead

To understand the various sources of overhead, let us examine the run-time object structures that are created for the expression `A(j,k) -= A(j,i)*A(i,k)/A(i,i)` in the Gaussian elimination algorithm. In Figure 2, the titles of boxes (e.g., `Expr, ExprRep`) correspond to classes in the CORE library: the container objects and multiple levels of indirection encapsulate concrete implementations of expression tree nodes (such as a binary subtraction operation). This encapsulation is conveniently expressed using class inheritance and virtual functions in an object-oriented language such as `C++`.

Expression evaluation involves a recursive traversal of the expression tree, and iterated traversals may be necessary because of the precision-driven nature of Level III evaluation. Maintaining the expression tree guarantees robustness, but it reduces execution efficiency on current-day systems with deep memory hierarchies. Level III evaluation suffers from three primary sources of overhead absent in Level I:

- *Function-call overhead*: The object-oriented programming style used in CORE encourages programs with small function (method) bodies that are dynamically dispatched using virtual functions. This increases the relative contribution of function call costs to overall execution time, and additionally reduces the effectiveness of sequential compiler optimizations (such as constant propagation).
- *Memory management overhead*: Expression trees in Level III are dynamic pointer-based structures

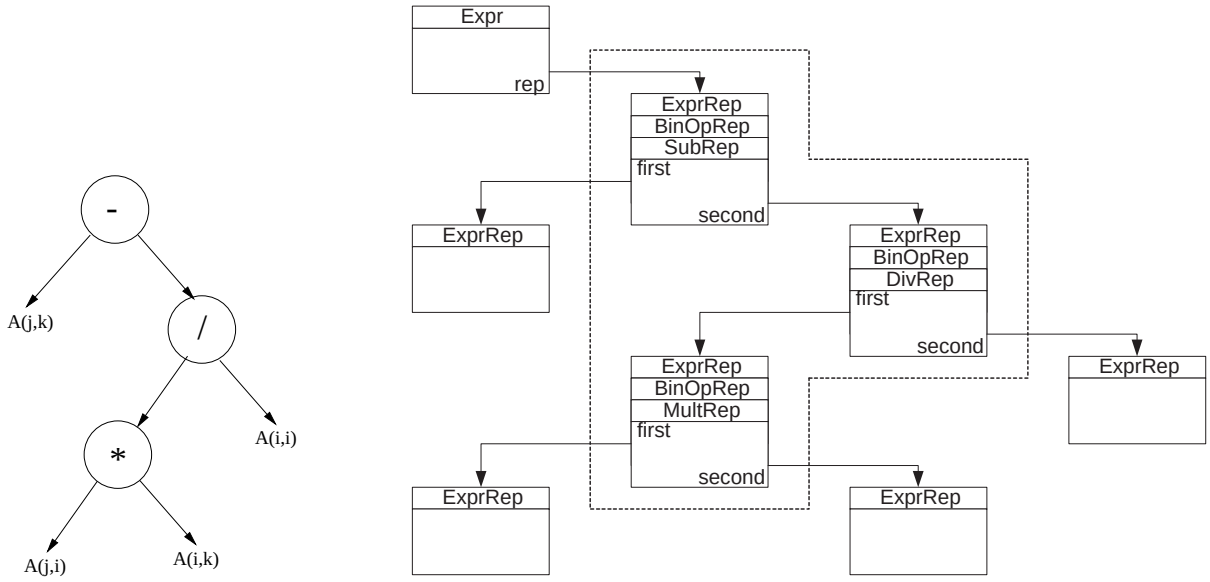

Figure 2: Expression tree created in the innermost loop of the Gaussian elimination algorithm. The boxes represent classes in the CORE library and cascaded boxes show the inheritance hierarchy.

consisting of simple small objects. These structures incur overheads for memory allocation and deallocation, and do not effectively utilize the memory hierarchies in current-day machines. Pointer structures suffer from cache-line fragmentation, as well as poor spatial locality.

- *Operation overhead*: Level III evaluation requires several iterations of the downward propagation of precision and upward propagation of errors. To preserve encapsulation, both these steps are performed at the granularity of individual operation nodes in the expression tree (e.g., a subtraction operation). The cost of iterations can be significantly reduced by exploiting knowledge of a global structure such as the subtraction-division-multiplication tree in the Gaussian algorithm.

One source of the high overheads at Level III arise from global program structures being constructed in an object-oriented style from smaller component objects. This fact has been recognized by several other researchers leading to the development of packages supporting large expressions: `LN` [13], `LEDA` [6], and `Real/Expr` [35].

4.2 Analysis and Optimization Approach

Our compiler-based approach reduces abstraction overheads using expression-level optimizations that break down the encapsulation introduced by the object-oriented style. The approach comprises three steps:

1. Identification of expression structures used by the program.
2. Propagation of precision requirements of a value to expressions that produce and consume it.
3. Customization of expression structure based on component operations and its leaf arguments.

The unifying analysis for the first two steps is a context-sensitive global interprocedural flow analysis [27, 11] which propagates the type information about a value to all the places that produce or consume the value. In this analysis, *type* more generally refers to both implementation type (e.g., `int`, `long`, or `double`) as well as the precision requirements of the value [14]. To prevent information loss, the analysis creates contexts (representing program environments) for differing uses of classes (for example, polymorphic containers), and methods (for example, differing types for a given argument at given call sites). Since expression trees are built up from individual expression tree nodes, the same analysis also detects expression "skeletons" even when these are built across procedure boundaries. Similar to analyses in object-oriented languages that resolve concrete types of method invocation targets [27, 9], our analysis helps identify the expression structures that are created at run time (and precision constraints on values consumed and produced by them). Identifying the expression structure permits its optimization using three broad categories of specialization—code, data, and operational:

CODE SPECIALIZATION. Information about the expression skeleton (its structure, the types of input and output values, and associated precision) enables several static and dynamic optimizations that reduce object-orientation overheads The most significant benefit is the elimination [27, 9] or optimization [16] of dynamic method dispatches, which in turns enable other optimizations such as inlining and cloning [28] that increase the effectiveness of traditional sequential optimizations.

DATA SPECIALIZATION. Knowledge of the expression structure also enables memory-efficient layout employing optimizations such as object inlining [11] and, in general, grouping of linked data structures. These transformations flatten the pointer-based data structures resulting in better cache-line utilization, improved spatial locality behavior, and more effective prefetching. These benefits are particularly important given the increasing latency (in processor cycles) of accessing off-chip memory. Additionally, the cost of dynamic memory management can be significantly reduced by customizing memory allocators for expression structures instead of relying on a generic memory allocator [15].

OPERATION SPECIALIZATION. The expression "skeleton" also provides a natural granularity at which to perform domain-specific optimizations. As discussed earlier, Level III precision and error propagation may be more effective if performed with respect to the global expression structure. Additional optimizations include partial-evaluation of fixed inputs (e.g., loop-invariant values), replacing a number representation with a more efficient one (e.g., automatically demoting Level II numbers to Level I numbers when the analysis detects no loss in precision), and in general, customizing the implementation for the concrete expression structure.

Figure 3 demonstrates the cumulative effects of these different specializations for the expression tree in Figure 2. The analysis yields the concrete types of objects making up the expression tree: `SubRep`, `DivRep`, and `MultRep`. This information enables code specialization where function invocations between these objects are statically resolved and optionally inlined. Data specialization inline-allocates the storage for the `DivRep` and `MultRep` objects in the `SubRep` object, reducing both object allocation and method invocation costs. Operation specialization constructs a composite `sub-div-mult` operator with an arity of four and customizes evaluation functions to take advantage of the global expression structure.

We are currently incorporating the techniques described here into a `C++`-to-`C` compiler built on top of the SUIF/OSUIF National Compiler Infrastructure [1]. The choice of `C` as the output language provides portable optimization of application packages built using the CORE library. End users can then compile the library using platform-specific native C compilers.

5 PILOT STUDIES

We conducted some basic experiments using Gaussian elimination and 2D Delauney triangulation algorithms to first quantify the current performance of CORE EGC techniques at different accuracy levels, and then verify the performance advantages of the compilation techniques described in Section 4. For the latter, since our compiler is still under development, we manually applied the analyses and transformations described in Section 4 to Level III techniques. Specifically, we identified a handful of expression tree "skeletons" and optimized their implementation by (1) replacing virtual function calls to intermediate nodes with statically bound inlined function calls, and by (2) providing custom memory allocators (customized to the size of the specialized structures). Our prototype implementation of the CORE library is layered on top of the `Real/Expr` package which in turn relies on a big-integer package supplied by GNU libg++. The implementation of the GNU big-integer package was not optimized. All experiments reported in this section were conducted on a SUN UltraSparc.

GAUSSIAN ELIMINATION. Computing determinants or the sign of determinants is perhaps the single most important primitive in geometric computation. We use the algorithm described in Section 2, modifying the 4×4 matrix B so that its entries have type `double` instead of `long`. Table 1 reports the execution times for 1000 determinant evaluations of matrix B at various levels of accuracy. The column labeled **Level III (opt)** corresponds to the optimization of the expression `A(j,k) -= A(j,i)*A(i,k)/A(i,i)`.

The results in Table 1 show clearly that there is a performance penalty associated with using higher levels of accuracy, and in particular Level III EGC which runs up to 150 times slower than Level I. Of course, this penalty must be balanced against the fact that EGC is robust, and eliminates all qualitative errors in the computation. What is very encouraging is that despite applying the transformations described in Section 4 at only a few places, performance of Level III evaluation improves by as much as a factor of two (for higher precision). Since the rest of the CORE library as well as the GNU big-integer package lend themselves to similar optimizations, significant additional performance improvements are likely.

2D DELAUNEY TRIANGULATION. We conducted some basic experiments by "preambling" an $O(n^4)$ code from Joe O'Rourke's book. While more efficient algorithms exist, our interest here is in understanding the relative efficiency of different accuracy levels. Ta-

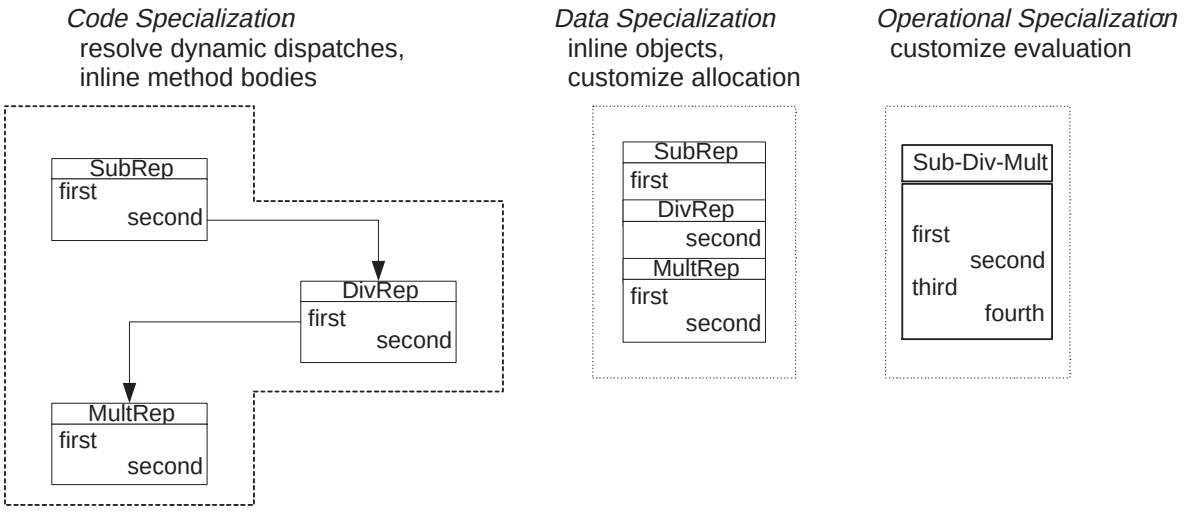

Figure 3: Code, data, and operation specializations for the Gaussian primitive.

Precision	Level I	Level II	Level III	Level III (opt)
1		3.29	3.94	2.32
10		3.42	3.94	2.36
IEEE double	0.03	3.46	3.99	2.38
100		3.48	4.15	2.39

Table 1: Execution times (in seconds) for 1000 determinant evaluations on a SUN UltraSparc at various accuracy levels for different amounts of precision (in bits).

Input Size	Level I	Level II	Level III	Level III (opt)
12	0.001	0.08	0.99	0.58
20	0.006	0.49	7.58	3.55
28	0.022	1.79	30.38	15.69
36	0.060	9.38	88.40	44.41

Table 2: Execution times (in seconds) for Delauney triangulation on a SUN UltraSparc at various accuracy levels for different numbers of cocircular points.

ble 2 reports the execution times for four input sizes, corresponding to different numbers of (exactly) cocircular points. Note that cocircular points are the worst case for Level III. **Level III (opt)** corresponds to the optimization of expressions in the body of the triply nested loop.

These results show the same trends as the Gaussian example: Level II evaluation is up to 150 times slower than Level I, and Level III contributes a further slowdown of up to 10 times because of its use of expression trees. As before, the transformations described in Section 4 are very effective, improving performance of Level III by as much as a factor of two despite being applied on a very small portion of the overall program. Similar techniques applied over the complete program have the potential of approaching the performance of hand-coded optimizations. As anecdotal evidence, researchers have shown that pure object-oriented languages such as Smalltalk [31], SELF [8], Cecil [9], and ICC++ [28], which share the same computation structure as EGC computations (and whose execution times are often two to three orders of magnitude worse than `C`) can, with aggressive compiler technology, achieve performance within a factor of 2 to 5 of a comparable `C` program.

6 CONCLUSION

Our Core Library represents a novel API for robust numeric and geometric computation. Its most striking feature is a nearly transparent integration with conventional `C/C++` programming. We believe such ease of use

is a necessary prerequisite for robustness research to impact programs in the real world. This paper has also demonstrated the efficiency gains possible using several automatic optimization techniques. In general, research is just beginning in the area of optimizing Exact Geometric Computation (EGC) techniques in a context that must balance the demands of object-oriented design with the need for code efficiency.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[2] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for Computation Geometry. *ACM Symp. on Computational Geometry*, 14:165–174, 1998.

[3] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. *ACM Symp. on Computational Geometry*, 13:174–182, 1997.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, 1999. To Appear.

[5] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. *ACM Symp. on Computational Geometry*, 14:175–183, 1998.

[6] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th ACM Symp. Comp. Geom.*, pages C18–C19, 1995.

[7] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[8] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI'89*, pages 146–60, 1989.

[9] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA'96*, 1996.

[10] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report 2971, INRIA, 1996.

[11] J. Dolby and A. Chien. An evaluation of object inline allocation techniques. In *OOPSLA'98 Proceedings*, 1998.

[12] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *ACM Symposium on Computational Geometry*, pages 163–172, 1993.

[13] S. J. Fortune and C. J. van Wyk. LN User Manual, 1993. AT&T Bell Laboratories.

[14] S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[15] D. Grunwald and B. Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software: Practice and Experience*, 23(8), 1993.

[16] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of ECOOP'91*, 1991.

[17] Cart3D Homepage, 1998. Homepage for adaptive Cartesian mesh generation. URL `http://george.arc.nasa.gov/~aftosmis/cart3d/cart3d.html`.

[18] CGAL Homepage, 1998. Computational Geometry Algorithms Library (CGAL) Project. A 7-institution European Community effort. See URL `http://www.cs.ruu.nl/`.

[19] CORE Homepage, 1998. Core Library Project: URL `http://cs.nyu.edu/exact/core/`.

[20] M. Karasick, D. Lieber, and L. R. Nackman. Efficient delauney triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10:71–91, 1991.

[21] G. Liotta, F. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. *ACM Symp. on Computational Geometry*, 13:156–165, 1997.

[22] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995.

[23] S. Ocken, J. T. Schwartz, and M. Sharir. Precise implementation of CAD primitives using rational parameterization of standard surfaces. In J. Hopcroft, J. Schwartz, and M. Sharir, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 245–266. Ablex Pub. Corp., Norwood, NJ, 1987.

[24] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[25] M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 53–58, Berlin, 1996. Springer. Lecture Notes in Comp. Sci. No. 1148.

[26] D. A. Patterson and J. L. Hennessy. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. (with an appendix on Computer Arithmetic by David Goldberg).

[27] J. Plevyak and A. A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94*, pages 324–340, 1994.

[28] J. Plevyak, X. Zhang, and A. A. Chien. Optimizing sequential efficiency for concurrent object-oriented languages. In *Proceedings of POPL'95*, 1995.

[29] J. Sellen, J. Choi, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *SIAM Journal Computing*, 1999 (accepted). Also: 11th ACM Symp. on Comp. Geom., (1995)350–359.

[30] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annual Symp. on Computational Geometry*, pages 141–150. Association for Computing Machinery, May 1996.

[31] D. M. Ungar. *the Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[32] C. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998. Invited Talk. Brown University, Oct 11–12, 1998. Abstracts in `http://www.cs.brown.edu/cgc/cgc98/home.html`.

[33] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, 1997.

[34] C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997. Invited talk, Proceed. 5th Canadian Conference on Comp. Geometry, Waterloo, Aug 5–9, 1993.

[35] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, 1995. 2nd edition.

[36] J. Yu. *Exact arithmetic solid modeling*. Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992. Technical Report No. CSD-TR-92-037.