CS368: Geometric Algorithms Design and Analysis Handout #10
Stanford University Monday, 24 May 2004

The following is a set of notes prepared by last year's class TA, Daniel Russel, on addressing numerical issues in geometric computation.

Numerical Issues in Computational Geometry

Daniel Russel

May 14, 2003

1 Introduction

Numerical inaccuracy is one of the main issues in writing robust computational geometric code. There have been a number of attempts to address this problem, the two most popular of which have been exact evaluation and floating point filters (which use exact evaluation). There have also been attempts to write algorithms which are tolerant of inaccurate primitives, but the results have tended to be rather more complicated than would be desired. The canonical example of this is [9].

In most geometric algorithms, there is a separation between the *combinatorial structures* and the *numerical calculations*. For example, in incremental Delaunay triangulations, on the combinatorial side there is the maintenance of the triangulation data structure and point location hierarchy, which never touches point coordinates, while the numerical calculations are limited to a point/plane orientation test and an in-circle test. In the Delaunay case, all the numerical routines are what is known as *predicates*–functions that take the input data as arguments and produce a discrete set of values (in the Delaunay case Positive, Degenerate, Negative, depending on the point/plane or sphere/point orientation). Many algorithms spend much of their running time computing predicates.

The alternative to predicates are *constructions*, routines in which the computed numerical value is returned. It is much harder to handle the numerical issues involved with predicates, and when possible it is advantageous to transform algorithms to avoid constructions. An example of this would be a naive computation of an arrangement where the intersection points are explicitly computed and then sorted along the lines. Instead, the intersection points could be labeled by the pair of lines. When the order of two intersection points (l_0, l_1) and (l_0, l_2) must be computed, then the a predicate involving (l_0, l_1, l_2) can be evaluated and one of (before, after, equal) can be returned.

Numerical error can be deadly in computing geometric structures, possibly resulting in crashes or infinite loops in addition to large combinatorial errors. Some failure rates of floating point predicate evaluation, taken from a Delaunay triangulation computation in [5] are as follows:

	R5	*R20*	*E*	*M*	*B*	*D*
% wrong orientation	0	0	0	.0001	.002	.17
% wrong in_sphere	0	0	0	.005	.0005	.8

The models used here are

- R5: 500,000 uniform random points
- R20: 2,000,000 uniform random points
- E: 500,000 points on an ellipsoid
- M: ≈ 500,000 points on a molecular surface
- B: ≈ 500,000 point happy Buddha
- D: ≈ 50,000 point CAD model with lots of flat surfaces

The same models with be used for examples throughout.

The traditional approach involved making up epsilon values (a guess at the likely error) below which a computed value was considered zero and tweaking this until things worked most of the time. This obviously is not very satisfactory and is very painful to perform.

Many geometric predicates, such as those in computing Delaunay triangulations, convex hulls and arrangements, can be reduced to finding the *sign of an algebraic expression*. In these cases I will refer to the *algebraic value* meaning the correctly computed value of the algebraic expression, the *approximate algebraic value*, meaning such a value computed using an inexact number type and the *predicate value* meaning the value the predicate should return. In such predicates, degeneracy can be naturally defined: an input is degenerate if any of the predicates you evaluate are 0 over any subset of the input. In the second applied assignment, that of building a Kinetic Data Structure for the 2D delaunay triangulation, you encountered another operation, that of *root sorting*, determining which root of which polynomial occurs first. The root sorting operation also occurs in computing the arrangement of higher degree curves (you need to figure out whether one intersection occurs before or after another along a curve). Root sorting can be reduced to a sign comparison in some cases (if you compute an explicit expression for the roots of the polynomials and then do the work to compare them), but there are other techniques based on bounding intervals using either Descartes rule of sign or Sturm sequences which can be more effective. We still are looking for a really nice solution to this problem.

Some concepts that will be used in the discussion of methods are

- *relative error*: a bound on the error as a fraction of the value. Note, in predicates where the sign of the value is important, the relative error of the answer does not matter.
- *additive error*: the maximum difference between the value and computed bound. A sign computation is correct if the additive error is less than the value.
- *bit complexity*: the number of bits necessary to compute the answer exactly, i.e. if you computed the answer using bignums with a fixed number if bits, this is how many bits you would need to use. This may be significantly larger than the number of bits in the input.

2 Exact Computation

The gold standard for predicate evaluation is exact computation. Most other numerical techniques need to have exact evaluation available as a last resort. In addition, any attempt to handle degeneracy, either explicitly, or by a perturbation technique such as *simulation of simplicity* [6]. The standardization of C++ has resulted in a number of number type classes which can be plugged into existing functions as a replacement for doubles to yield exact results. Exact number types have the additional advantage that they can be used for constructions without the concerns which accompany the use of doubles.

Many predicates, such as the determinant based ones we have seen in class, can be evaluated using only addition (and subtraction) and multiplication. These operations can be supported using (possibly extended) integers (by multiplying all the input by a large constant) or extended precision floating point numbers (generally a 32 bit exponent and extended or arbitrary precision mantissa). The simplest way to do this is to use an infinite precision integer package such as GMP which allocates space on the fly to store as many bits as need. For well structured predicates you can predict exactly how many bits are needed by computations at each stage, and perform the computations using only integers of that size–each addition increases the bit count by at most 1, each multiplication adds the two bit counts–thus saving on the memory allocation overhead. Most predicates do not have too high a bit complexity, so the memory usage is not too bad, but the extra overhead for memory management and multi-word arithmetic support is still quite high. The most recent estimate suggests that extended precision slows calculations down by a factor of 70 on Delaunay triangulations [5].

The next most complex operation to support is division (extending representation to rational numbers). This is generally supported by switching to a homogeneous or rational representation. A number X becomes a pair (x, y) where $X = x/y$. Now division and multiplication are equivalent (and easy), but storage is doubled and the number of machine operations necessary to perform an arithmetic operation have greatly increased. This approach is equivalent to the common computation geometry/graphics technique of representing a point (x, y, z) as a homogeneous vector (w, x, y, z) or (x, y, z, w). This representation of points allows certain dualities as well as projective maps to performed without division (or, in some cases, without any modification of the values at all), and as a result is advantageous without regard to the numerical considerations. CGAL, LEDA, CORE, and GMP all have rational number types that support these operations exactly. The cost of this support varies depending on exactly what is used but tends to slow things down by one to two orders of magnitude over floating point math.

The final operation generally needed and the most difficult to support, is the root operation (extending representation to real numbers). Since square roots can be irrational, there is not way to generally represent them as a bit sequence or a ratio, as before. Therefore most approaches represent a real number as an expression tree of some sort (as well as an interval which isolates the desired root). This make comparison a non-trivial operation involving repeatedly isolating and squaring roots until a root free expression is found. Since these numbers generally appear as the roots of polynomials, approaches which act directly on the polynomials (again, generally based on Sturm sequences or Descartes rule of sign) are also used, but tend to be much more specialized. Support for real numbers tends to slow predicates down by another order of magnitude over rationals.

Floating point hardware has become so heavily optimized that it is generally advantageous to use floating point rather than integer operations. A double gives 53 bits of exact storage (compared to the 32 bit for integers) and the floating point operations units are often more numerous and take fewer cycles. However, it is a bit trickier to get all the coding correct to perform exact operations using them. The widespread adoption of IEEE floating point has made implementation easier, but the necessary control of the rounding modes is still not standard, making portability an issue. Most big number packages use integers at this point (GMP, CGAL, LEDA), but most specialized packages written to exactly evaluate predicates tend to use doubles.

For example Jonathan Shewchuk [10] represents extended precision numbers as sums of floating point numbers in his Triangle and Pyramid packages. The numbers are chosen so that the leading non-zero bits of the mantissa do not overlap. For example, $.11111100 \times 2^6 + .11110000 \times 2^0$ is OK, but $.11111100 \times 2^5 + .11110000 \times 1$ is not since the last 1 on the first term occupies the same place as the first one of the second term. This allows him to efficiently represent $2^{300} - 2^{-300}$ using just two doubles instead of an array of 600 bits. In addition there is no conversion overhead from input doubles to the extended format.

An alternative approach to exact computation is taken by Clarkson in [3] where he presents an algorithm which can exactly evaluate the sign of a determinant of an $n \times n$ matrix using $1.5n + 2M$ bits where M is the bit complexity of the input. However, this algorithm is significantly slower than evaluation with doubles in practice, so it is not an all-purpose solution.

The high cost of exact computation relative to computation with floats has led to the invention of floating point filters which are covered in the next section.

3 Filters

Floating point filters are the most popular way of avoiding the high computing overhead of exact computation. In a filter, approximate algebraic value is first computed using an inexact number type (generally doubles as they are the fastest and most versatile). In addition, an additive error bound, called the *filter bound* is computed. If the approximate algebraic value is larger than the filter bound than the sign of the approximate value matches the sign of the true algebraic, value, and the predicate value can be determined. If not, the calculation must be repeated with an exact number type or a more expensive (and tighter) filter value must be computed.

The are several types of filters used and they can be categorized into 3 rough categories depending on how and when the filter bound is computed. The two important properties of a filter are the *overhead* (how expensive it is to compute the filter bound) and the *failure rate* (how often the magnitude of the algebraic value is smaller than the filter value). Clearly, if the input is degenerate any filter will fail, and the failure rate depends on the types of input data expected.

3.1 Static Filters

In a static filter, a filter bound for each predicate is computed at compile time or initialization time using any properties known about the data. Generally, the property used is the maximal bit length of the input. Static filters are the lowest overhead filter type since the only runtime computation is a single comparison and one of the first used. A good presentation of static filters can be found in [7]. It should be clear that this sort of bound can't support division.

Computing the maximum error given a maximal bit length is fairly straight forward. First, rules for updating the bit lengths of arguments are needed:

- $maxbitlen(a \pm b) = 1 + \max(maxbitlen(a), maxbitlen(b))$
- $maxbitlen(a \times b) = maxbitlen(a) + maxbitlen(b)$

Using this, the maximum error is calculated as follows:

- $maxerr(a \pm b) = maxerr(a) + maxerr(b) + 2^{maxbitlen(a+b)-53}$
- $maxerr(a \times b) = maxerr(a)maxbitlen(b) + maxerr(b)maxbitlen(a) + 2^{maxbitlen(a \times b)-53}$

The computed bounds are small until the bit length exceeds the number of bits that are being used to compute the answer (53 in the case of doubles) at which point it grows quite rapidly.

The error bounds can be easily computed by running the predicate with a specialized C++ number type. However, this approach is only as good as your predicate and won't be able to take advantage of any special arithmetic structure (for example terms which will generally cancel or are equal).

Failure rates for the previously mentioned models are below. The high failure rates for the in-circle test are presumably due to its higher algebraic degree resulting in the 53 exact bits being saturated early in the calculations.

	R5	*R20*	*E*	*M*	*B*	*D*
% failed orientation	0	0	.001	.015	.1	7.3
% failed in_sphere	7	37	39	16	74	17

In cases where the bit length limits are not know a-priori, the input data can be scanned and the filter bounds can be computed as part of the initialization.

3.2 Semi-static Filters

Semi-static filters are the least structured of the three category. They include any filter where a bound is computed for each call of the predicate but using a much simpler calculation than the actual predicate computation. One example of this would be to look at the bit length of the actual input to the predicate at each call and compute a bound using a calculation identical to that performed for the static filter. In determinate based predicates the computation is regular enough that the above error bounds can be calculated cheaply compared to the actual determinant. Since the predicates often only depend on the difference of point coordinates, the actual bit length of the input to most calls of the predicate may be a great deal smaller than the maximum possible. For example, in randomized incremental Delaunay, most of the in-circle tests are performed using 5 points which are close to one another, so the bit length of their difference is quite small, while the static filter has to compute a bound which holds for 5 points chosen from the extremes of the model.

In addition, tricks can be performed, such as translating a point parallel to the test plane in a point/plane orientation test, which further reduces the bit complexity [5].

Failure rates for the previously mentioned models are using both of the above semi-static techniques are:

	R5	*R20*	*E*	*M*	*B*	*D*
% failed orientation	0	0	.0002	.001	.1	7.3
% failed in_sphere	.6	4.6	20	4	37	7.5
% failure for translation	0	0	0	.007	.003	.8

A more sophisticated sort of semi-static filter is presented in by [1] for computing the sign of a determinant. There they perform LU decomposition on the predicate matrix using doubles, invert L and U and then use the difference between the numerical inverses and the original matrix to compute the filter bound.

3.3 Dynamic Filters

The final type of filter commonly used is a dynamic filter. This means that a running estimate of the error is computed along with the value in any computations of the predicate. The most dominant approach currently is interval arithmetic, mostly due to its inclusion in CGAL. In this method, a pair of doubles is maintained as the calculation proceeds, one of which is the lower bound and the other is the upper bound for the true algebraic value, and this pair is updated for each arithmetic operation. When the sign is desired, a check is made to see if the interval includes 0, if it does not, then the sign can be returned, if it does, a more accurate computation must be performed. The approach is very appealing as it simply requires changing the number type and can support any sort of flow control.

To further explain, let $[x], [y]$ be interval values $[\underline{x}, \overline{x}], [\underline{y}, \overline{y}]$. The interval update rules are as follows:

- $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
- $[x] - [y] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$
- $[x] \times [y] = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})]$
- $[x]/[y] = \begin{cases} [x] \times [1/\overline{y}, 1/\underline{y}] & \text{if } 0 \notin [y] \\ \mathbb{R} & \text{otherwise} \end{cases}$
- $[x]^{1/2} = \begin{cases} [\underline{x}^{1/2}, \overline{x}^{1/2}] & \text{if } 0 \leq \underline{x} \\ \mathbb{R} & \text{otherwise} \end{cases}$

These rules depend on the floating point calculations producing the closest representable value to the correct one (as is guaranteed by the IEEE specification) and rounding properly (down for lower bound, up for upper bound). In order to avoid constantly changing the rounding mode, in the CGAL implementation the intervals are represented as $[-\underline{x}, \overline{x}]$ instead. Knuth has a good description of computing error bounds for floating point calculations in volume 2, section 4.2.

The failure rates for the standard examples are:

	R5	*R20*	*E*	*M*	*B*	*D*
% failed orientation	0	0	0	.0005	.005	7.4
% failed in_sphere	0	0	0	.007	.001	1.35

The error can also be tracked by computing the best floating point estimate and then the magnitude of the error. LEDA takes this approach.

4 Available Software

There are a number of packages available which include support for or tools for supporting exact predicate evaluation and floating point filters. Most of them make strong use of various features of C++ in order to simplify usage. Compilers are now mostly standardized so compiler support is fairly broad for must packages, but some of them are difficult to use with compilers other than gcc (especially GMP). Packages which depend on setting the floating point rounding modes (any extended precision floating point code or interval arithmetic) are also less portable as there is no platform independent to the floating point rounding modes.

4.1 CGAL

The more recent versions of CGAL [2] (2.3 and higher) include a filtered version of the Cartesian kernel. The kernel first evaluates the predicate using Sylvain Pion's interval arithmetic number type and an exception is thrown if the sign cannot be determined. In that case the calculation is repeated with an exact number type. The C++ exception mechanism is fairly expensive (compared to a rewritten predicate which checks the bounds explicitly), but this approach makes implementation very simple and very general. The predicate can be implemented once as a functor templated by the number type, and the predicate is passed as a template argument to a filter helper class (which acts as the actual predicate). When the predicate is called, the helper class runs it using an interval type, which throws an exception if the sign is asked for on an interval which includes 0. If so, the predicate is run again using an exact type. This means that any sort of expression (using operations supported by the interval type) and any sort of flow control is supported with no extra work on the part of the programmer. Since (hopefully) the filter rarely fails, the expense of exception handling is not too much of a penalty. There are also plans to put more specific predicates into various algorithms, especially Delaunay triangulations.

4.2 LEDA

LEDA includes a number number types to aid implementation. These include extended precision integers, reals and floats, and a real type which uses a variety of techniques to support arbitrary algebraic operations. I don't know so much about their packages and they are commercial.

4.3 GMP

GMP [8] is the standard implementation for extended precision number types. It is very heavily optimized and represents a large coding effort over the years. The types supported include arbitrary precision integers and rationals and extended precision floating point numbers. The main source code is in C which makes usage somewhat uglier than a C++ library. The package now includes a primitive, but improving, C++ wrapper for several of the types withing GMP. CGAL also has a C++ wrapper for GMP, as does another package called CLN [4].

4.4 CORE

The CORE library [11] by Chee Yap is designed to allow people to write code which can be run at a variety of different tolerances with no extra programming effort. It has 4 different modes of operation, chosen by the value of a preprocessor variable:

- LEVEL I: Machine accuracy. Do computations with doubles.
- LEVEL II: Arbitrary accuracy. Do the computations using a certain number of bits for all computations. This provides no guarantees about the correctness of the results.
- LEVEL III: Guaranteed accuracy. Guarantee that a certain number of bits in the solution will be correct. Level III run with 1 bit means that the sign of the answer is guaranteed to be correct.
- LEVEL IV: Mixed accuracy. Mix levels I,II,III based on local need to allow finer control. This is not currently implemented.

The library is designed to allow you to run almost any code unmodified, and simply changes the underlying number type used to support the desired accuracy. Levels I and II aren't particularly interesting (just being doubles and fixed precision big integers), although CORE's support is rather nice. The implementation of Level III is more interesting and is rather similar to the LEDA Real. Performing a calculation results in a variable (the result) which contains an expression tree for the whole calculation. When the sign is asked for, this expression tree is then evaluated with different bit lengths (precisions) as needed and the error computed. If the error is too large, then the precision of all or some of the tree is increased and the computations are repeated. This is iterated until the desired error bound is achieved.

They have plans for a compiler based on these ideas which would allow much more aggressive optimizations. So far that has not appeared.

5 Putting it all Together

For Delaunay triangulations Olivier Devillers and Sylvian Pion ran a bunch of timing tests with various sorts of filters and computation types [5]. Timings in seconds, for the same models as before, are:

	R5	*R20*	*E*	*M*	*B*	*D*
double	40.6	176.5	41.0	44	50.3	∞
MP_Float	3,063	12,524	2,777	3,195	3,472	214
Interval	137	574	133	144	165	16
semi-static + interval	52	234	61	59	93	9
static + semi-static + interval	44	210	55	52	87	8
static + translation + interval	43	199	52	49	64	8
static + semi-static + translation + interval	43	196	49	49	64	8
Shewchuk's predicates	58	249	58	63	72	7
CORE Expr	570	2431	3520	1355	9600	173
LEDA real	682	2784	640	742	850	125
Lazy_exact_nt<MP_Float>	705	2750	631	726	820	67

MP_Float is the GMP arbitrary precision floating point type (it is implicitly the last stage in all the filtered runs). Lazy_exact_nt<MP_Float> is a CGAL wrapper to support reals similar to CORE's Expr or the LEDA real. As shown in the data, many layers of filters, applied from coarse (and cheap) to fine can aid calculation speed, although the gains after 4 layers are fairly minimal.

References

[1] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[2] CGAL. http://www.cgal.org.

[3] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395, Pittsburgh, PA, October 1992.

[4] Class library for numbers. http://www.ginac.de/CLN.

[5] O. Devillers and S. Pion. Efficient exact geometric predicates for delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44, 2003.

[6] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

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

[8] T. Granlunch. GMP. http://www.swox.com/gmp.

[9] D. Salesin, J. Stolfi, and L. Guibas. Epsilon geometry: building robust algorithms from imprecise computations. In *Proceedings of the fifth annual symposium on Computational geometry*, pages 208–217. ACM Press, 1989.

[10] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, Oct. 1997.

[11] C. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998.