sum of all products containing i of the y variables, so that Φ_i has magnitude $\mathcal{O}(\epsilon^i)$.

One can obtain an approximation A_j with error no larger than $\mathcal{O}(\epsilon^j)$ by exactly summing the first j terms, Φ_0 through Φ_{j-1}. The sequence $A_1, A_2, \ldots$ of increasingly accurate approximations can be computed incrementally; A_j is the exact sum of A_{j-1} and Φ_{j-1}. Members of this sequence are generated and tested until one is sufficiently accurate.

An improvement is based on the observation that one can obtain an approximation with error no larger than $\mathcal{O}(\epsilon^j)$ by adding (exactly) to A_{j-1} a correctional term that approximates Φ_{j-1} with ordinary floating-point arithmetic, to form a new approximation C_j, as illustrated in Figure 3(c). The correctional term reduces the error from $\mathcal{O}(\epsilon^{j-1})$ to $\mathcal{O}(\epsilon^j)$, so C_j is nearly as accurate as A_j but takes much less work to compute. This scheme reuses the work done in computing members of A, but does not reuse the (much cheaper) correctional terms. Note that C_1, the first value computed by this method, is an approximation to Φ_0; if C_1 is sufficiently accurate, it is unnecessary to use any exact arithmetic techniques at all. This first test is identical to Fortune and Van Wyk's floating-point filter.

This method does more work during each stage of the computation than the first method, but typically terminates one stage earlier. Although the use of correctional terms is slower when the exact result must be computed, it can cause a surprising improvement in other cases; for instance, the robust Delaunay tetrahedralization of points arrayed in a tilted grid (see Section 5.4) takes twice as long if the estimate C_2 is skipped in the orientation and incircle tests, because A_2 is much more expensive to produce.

The decomposition of an expression into an adaptive sequence as described above could be automated by an LN-like expression compiler, but for the predicates described in the next section, I have done the decomposition and written the code manually. Note that these ideas are not exclusively applicable to the multiple-term approach to arbitrary precision arithmetic. They can work with multiple-digit formats as well, though the details differ.

5 Predicate Implementations

5.1 Orientation and Incircle

Let a, b, c, and d be four points in the plane whose coordinates are machine-precision floating-point numbers. Define a procedure ORIENT2D(a, b, c) that returns a positive value if the points a, b, and c are arranged in counterclockwise order, a negative value if they are in clockwise order, and zero if they are collinear. A more common (but less symmetric) interpretation is that ORIENT2D returns a positive value if c lies to the left of the directed line ab; for this purpose the orientation test is used by many geometric algorithms.

Define also a procedure INCIRCLE(a, b, c, d) that returns a positive value if d lies inside the oriented circle abc. By *oriented circle*, I mean the unique (and possibly degenerate) circle through a, b, and c, with these points occurring in counterclockwise order about the circle. (If the points occur in clockwise order, INCIRCLE will reverse the sign of its output, as if the circle's exterior were its interior.) INCIRCLE returns zero if and only if all four points lie on a common circle.

These definitions extend to arbitrary dimensions. For instance, ORIENT3D(a, b, c, d) returns a positive value if d lies below the oriented plane passing through a, b, and c. By *oriented plane*, I mean that a, b, and c appear in counterclockwise order when viewed from above the plane.

In any dimension, the orientation and incircle tests may be implemented as matrix determinants. For example:

$$\text{ORIENT3D}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{1}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \tag{2}$$

$$\text{INCIRCLE}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} \tag{3}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix} \tag{4}$$

These formulae generalize to other dimensions in the obvious way. Expressions 1 and 2 can be shown to be equivalent by simple algebraic transformations, as can Expressions 3 and 4. These equivalences are unsurprising because one expects the results of any orientation or incircle test not to change if all the points undergo an identical translation in the plane. Expression 2, for instance, follows from Expression 1 by translating each point by $-d$.

For exact computation, the choice between Expressions 1 and 2, or between 3 and 4, is not straightforward. Expression 2 takes roughly 25% more time to compute in exact arithmetic, and Expression 4 takes about 30% more time than Expression 3. The disparity likely increases in higher dimensions. Nevertheless, the mechanics of error estimation turn the tide in the other direction. Important as a fast exact test is, it is equally important to avoid exact tests whenever possible. Expressions 2 and 4 tend to have smaller errors (and correspondingly smaller error estimates) because their errors are a function of the relative coordinates of the points, whereas the errors of Expressions 1 and 3 are a function of the absolute coordinates of the points.

In most geometric applications, the points that serve as parameters to geometric tests tend to be close to each other. Commonly, their absolute coordinates are much larger than the distances between them. By translating the points so they lie near the origin, working precision is freed for the subsequent calculations. Hence, the errors and error bounds