Computational aspects of Gaussian beam migration

Dave Hale

ABSTRACT

The computational efficiency of Gaussian beam migration depends on the solution of two problems: (1) computation of complex-valued beam times and amplitudes in Cartesian (x, z) coordinates, and (2) limiting computations to only those (x, z) coordinates within a region where beam amplitudes are significant.

The first problem can be reduced to a particular instance of a class of closest-point problems in computational geometry, for which efficient solutions, such as the Delaunay triangulation, are well known. Delaunay triangulation of sampled points along a ray enables the efficient location of that point on the raypath that is closest to any point (x, z) at which beam times and amplitudes are required. Although Delaunay triangulation provides an efficient solution to this closest point problem, a simpler solution, also presented in this paper, may be sufficient and more easily extended for use in 3-D Gaussian beam migration.

The second problem is easily solved by decomposing the subsurface image into a coarse grid of square cells. Within each cell, simple and efficient loops over (x, z) coordinates may be used. Because the region in which beam amplitudes are significant may be difficult to represent with simple loops over (x, z) coordinates, I use recursion (a function that calls itself) to move from cell to cell, until the entire region defined by the beam has been covered.

Benchmark tests of a computer program implementing these solutions suggest that the cost of Gaussian beam migration is comparable to that of migration via explicit depth extrapolation in the frequency-space domain. For the data sizes and computer programs tested here, the explicit method was faster. However, as data size was increased, the computation time for Gaussian beam migration grew more slowly than that for the explicit method.

INTRODUCTION

Gaussian beam migration may be summarized concisely (Hale, 1992) by

$$g(x,z) = \sum_j \int dp_x \, A_j(p_x,x,z) \, b_j[\tau = \tau_j(p_x,x,z), p_x], \tag{1}$$

where $g(x,z)$ denotes the subsurface image, a function of horizontal distance x and depth z, and $b_j(\tau,p_x)$ denotes a local slant stack of seismic data recorded at the surface $z=0$, a function of time τ for each reflection slope p_x. The time and amplitude functions $\tau_j(p_x,x,z)$ and $A_j(p_x,x,z)$ are complex-valued, and they determine the mapping of each local slant stack (beam) onto the subsurface image. The sum over beam centers x_j and the integral over reflection slopes p_x simply accumulate the contributions of each beam.

Figure 1 illustrates the contribution of one Gaussian beam formed from synthetic seismic data. This beam corresponds to a beam center $x_j = 5\,\text{km}$ and a reflection slope p_x that is equivalent to an emergence angle of 36 degrees. The beam is refracted by a gradual increase in velocity with depth and by a low-velocity zone centered at $x = 4\,\text{km}$ and $z = 1.5\,\text{km}$.

The fundamental task in Gaussian beam migration is to efficiently compute the contribution, like that shown in Figure 1, of each beam to the subsurface image. The prerequisite task of computing $b_j(\tau,p_x)$ (beam forming) can be accomplished simply by slant stacking a Gaussian-tapered window of data, and this tapering and slant stacking can be performed efficiently via fast Fourier transforms. However, once the data have been beam-formed, one must then efficiently map each beam onto the subsurface image, and this mapping is complicated by two problems.

The first problem is due to the fact that the complex-valued functions $\tau_j(p_x,x,z)$ and $A_j(p_x,x,z)$ are difficult (although not impossible) to compute in the Cartesian coordinates x and z. They are more easily and typically computed as functions of ray-centered coordinates (s,n) (Červený, et al., 1982; Hill, 1990; Figure 2 below). Transformation from ray-centered coordinates to Cartesian coordinates requires that we determine the point on a ray [the ray that defines the coordinates (s,n)] that is nearest to a given point (x,z). As noted by Hill (1990), "the transformation from ray-centered (s,n) to Cartesion (x,z) coordinates and ... are time-consuming computations."

The second problem in efficiently mapping each beam is that computations should be performed for only those subsurface points (x,z) where the contribution of the beam is significant. As illustrated in Figure 1, not all subsurface points are affected by a given beam. In Gaussian beam migration of recorded seismic data, each beam may contribute significantly to only a small fraction of the sampled subsurface points (x,z). (See Hill, 1990, for some particularly good examples.) However, the subset of samples affected by a single beam forms a region that may be quite irregular and not easily translated into bounds for simple loops over x and z in computer programs.

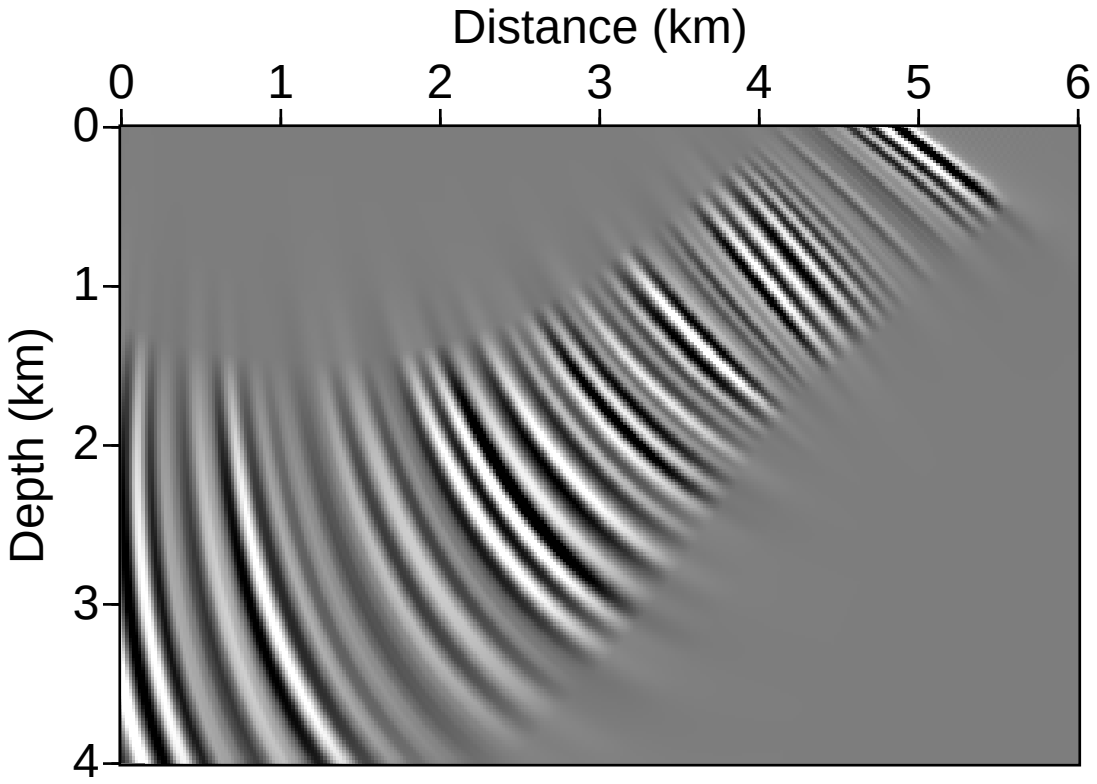

FIG. 1. Contribution of one Gaussian beam to a subsurface image. The fundamental task in Gaussian beam migration is to efficiently compute many such contributions, the sum of which yields the complete image.

This paper proposes solutions to these two computational problems: (1) finding the nearest point on a ray and (2) working only within a beam. Following the discussion below of these two problems and their solutions, the results of benchmark tests of a computer program implementing these solutions are presented.

NEAREST POINT ON A RAY

In this section, I assume that dynamic ray tracing (Červený, et al., 1982; Hill, 1990) has been performed to compute complex-valued time and amplitude functions in ray-centered coordinates (s, n). Equation (1) requires that these functions be evaluated in Cartesian coordinates (x, z). Therefore, we need a method to determine (s, n) coordinates from (x, z) coordinates.

Figure 2 illustrates the two different coordinate systems for the raypath used to compute the beam of Figure 1. This raypath was computed by integrating the dynamic ray tracing equations via a simple 4th-order Runge-Kutta method (e.g.,

Press, et al., 1986, 550–554). Solutions to these equations were thereby obtained at points uniformly sampled in time along the raypath.

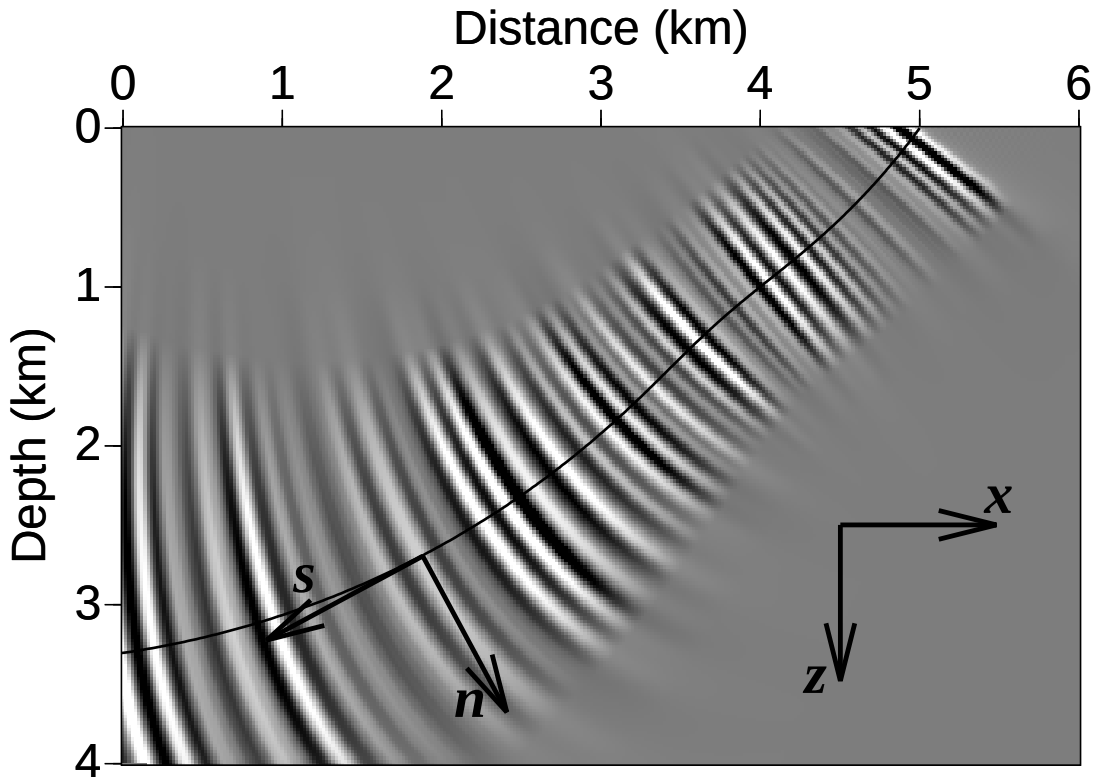

FIG. 2. Time and amplitude functions for this Gaussian beam were first computed in ray-centered coordinates (s, n) and then transformed to Cartesian coordinates (x, z). s denotes distance along the raypath and n denotes normal distance to the raypath.

Solution of the dynamic ray tracing equations requires evaluation of velocity and its derivatives along the raypath. Here, velocity was uniformly sampled in Cartesian coordinates x and z. Piecewise Hermite bi-cubic interpolation with continuous first (but not second) derivatives (e.g., Thomson and Gubbins, 1982) was used to interpolate velocity and its derivatives at points along the raypath. An attractive feature of bi-cubic interpolation in two dimensions is that it is easily extended to tri-cubic interpolation in three dimensions, as required for Gaussian beam migration of 3-D seismic data.

Although more sophisticated ray tracing methods might be used, the simple ray tracing method described above is efficient in the context of Gaussian beam migration. Accumulation of each beam requires evaluation of the seismic wavefield at many subsurface points (x, z). If N time steps are used in the solution of the dynamic ray tracing equations, then the computational cost of ray tracing is proportional to N. However, the cost of *using* the results of this ray tracing, the cost of accumulating a beam, is proportional to N times the nominal width N_w of the beam. As illustrated in

Figure 2, N_w varies along the raypath. However, for typical beam widths, the $O(N)$ cost of simple Runge-Kutta ray tracing is insignificant compared with the $O(N \times N_w)$ cost of accumulating the corresponding Gaussian beam.

For any given coordinates (x, z), corresponding coordinates (s, n) can be determined simply by finding that point on the raypath that is nearest to (x, z). s is then the distance along the ray to the nearest point, and n is the distance from (x, z) to that nearest point. For the Runge-Kutta ray tracing described above, which yields a raypath discretely sampled in time, we need only find the sample that is nearest to the specified (x, z). Once we know the nearest sampled point, the complex time and amplitude functions can be determined easily from quantities computed during dynamic ray tracing (Červený and Pšenčík, 1984).

As noted by Červený, et al. (1982), normals to a curved raypath will intersect at large distances from the raypath, so that a single set of (x, z) coordinates will correspond to more than one set of (s, n) coordinates. Fortunately, as illustrated in Figure 2, Gaussian beam amplitudes tend to be largest at points (x, z) that are nearest the raypath. Therefore, by choosing the smallest normal distance to the raypath, we choose that set of (s, n) coordinates that tends to yield the largest contribution to the subsurface image. In the special case that two or more points on the raypath are nearest and equidistant to (x, z), we will find only one of them, so that only one contribution of the Gaussian beam will be accumulated at that point (x, z).

The simplest way to find the nearest point on the raypath is to compute the distance from the specified (x, z) to all sampled points along the raypath, remembering the sample that corresponds to the minimum distance. However, this approach is inefficient, requiring a computational cost proportional to N for every point (x, z). This $O(N)$ cost is significant, because the number of points for which the nearest point problem must be solved is proportional to the area within the beam, $N \times N_w$, which implies an $O(N^2 \times N_w)$ cost per beam.

The reason that the simplest solution to the nearest point problem is so costly is that it fails to take advantage of the fact that the nearest point on the raypath for one (x, z) is likely to be close to the nearest point for a neighboring (x, z). As one iterates over the points (x, z) within a beam in some ordered fashion, the nearest point on the raypath is unlikely to jump wildly from end of the raypath to the other.

Unfortunately, it is difficult to guarantee that such wild behavior will never occur, particularly in the cases of turned rays and rapid velocity variations. The simplest method, while inefficient, is guaranteed to yield the nearest point.

In the remainder of this section, two more efficient solutions to the nearest point problem are presented. Both are guaranteed to yield the nearest point on the raypath. The first is highly efficient, but difficult to extend to 3-D ray tracing. The second is less efficient, but easily extended to 3-D ray tracing. Both solutions have a computational cost that is significantly less than the $O(N)$ cost of the simplest solution.

Triangles

Voronoi tessellation (e.g., Preparata and Shamos, 1985) provides a natural solution to the nearest point problem. The region bounded by the Voronoi polygon corresponding to any vertex in a Voronoi tessellation is, by definition, the locus of points that are nearer to that vertex than to any other vertex. If we let each sampled point on a raypath be a vertex in a Voronoi tessellation, then to find the point on the raypath nearest to an arbitrary point (x, z), we need only find that vertex with a Voronoi polygon containing the point (x, z). Preparata and Shamos (1985) show that the worst-case computational cost of constructing the Voronoi tessellation is $O(N \times \log N)$ and that the cost of finding the nearest vertex is $O(\log N)$.

Rather than constructing a Voronoi tessellation of the sampled points on a raypath, it is simpler to build its dual, a Delaunay triangulation of the Voronoi vertices (Watson, 1981). Figure 3 shows the Delaunay triangulation of sampled points along the raypath in Figure 2. For clarity in this figure, only one tenth of the sampled points that are actually computed during ray tracing are shown in the triangulation. The Delaunay triangulation, like the Voronoi tessellation, may be computed with a cost of $O(N \times \log N)$.

Given the Delaunay triangulation of a set of vertices, like that shown in Figure 3, one can easily find the vertex that is nearest to any point (x, z). Starting from any vertex, follow the edges of the triangulation, moving from vertex to vertex. When moving from a vertex to one of its neighbors, always choose the neighbor that is closest to the point (x, z). Eventually, a vertex will be reached that is nearer to (x, z) than any of its neighbors. That vertex is the nearest vertex in the triangulation.

This algorithm is efficient when iterating over numerous (x, z) coordinates. The nearest vertex (nearest point on the raypath) for one (x, z) is likely to be close to the nearest vertex for a nearby (x, z). However, this algorithm will successfully find the nearest point on the raypath, even when that nearest point jumps from one end of the raypath to the other as (x, z) changes. As illustrated in Figure 3, the edges of the Delaunay triangulation make such large jumps possible, although they will rarely be taken.

In the special case of a straight ray (as for constant velocity), triangulation of the collinear points along the raypath is impossible. However, the sampled points can still be connected with edges such that the vertex-to-vertex search can be performed as described above for a Delaunay triangulation. One way to avoid handling this special case is to always include in the triangulation three artificial vertices that form a large equilateral triangle containing the entire raypath. These fake vertices must be far enough away from the raypath that they will never be the closest vertex in the triangulation to any point (x, z) of interest.

The main drawback of the Delaunay triangulation solution to the nearest point problem is that it is difficult to extend to three dimensions for use in Gaussian beam migration of 3-D seismic data. Although such an extension is possible (the triangles

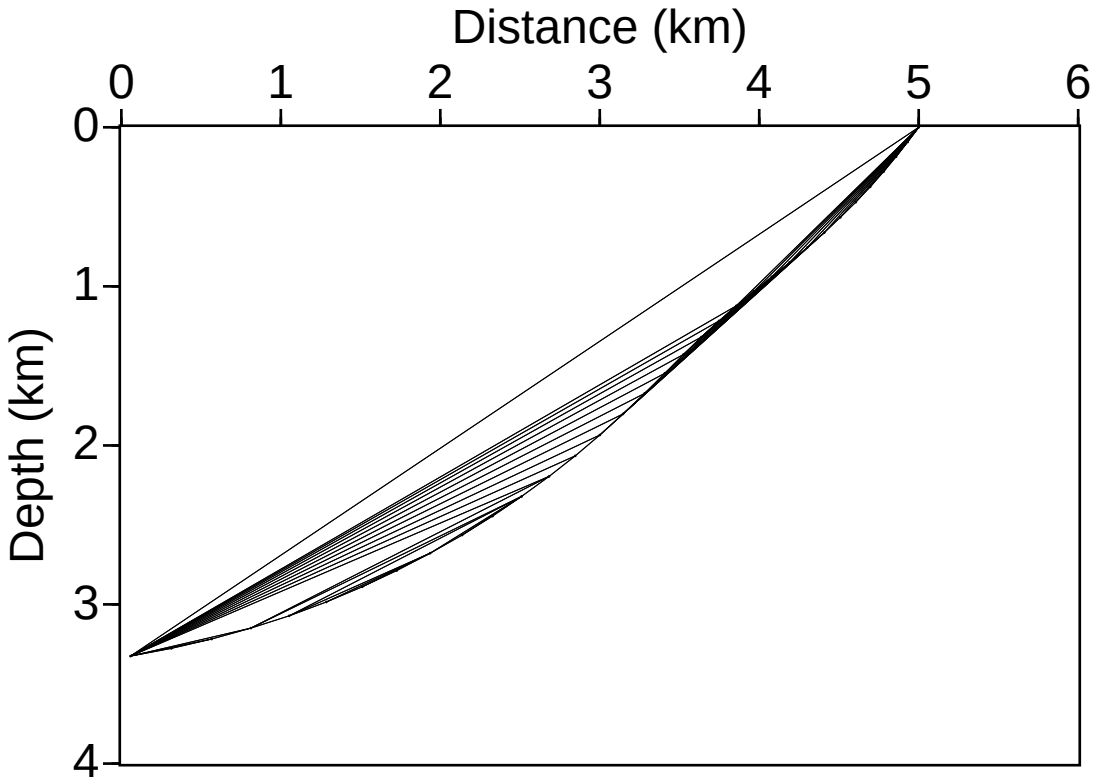

FIG. 3. Delaunay triangulation of sampled points along a raypath. By following the edges of this triangulation, one can efficiently determine that point on the raypath that is nearest to an arbitrary point (x, z).

become tetrahedra), it is not trivial. Watson's (1981) triangulation algorithm is perhaps the easiest to extend to three dimensions, but the cost of triangulation with Watson's method is $O\left(N^{3/2}\right)$ for two dimensions and $O\left(N^{5/3}\right)$ for three dimensions. Therefore, in keeping with the goal of using only computational methods that may be easily extended for use in 3-D imaging, an alternative solution to the nearest point problem is proposed below.

Circles

Figure 4 illustrates another solution to the nearest point problem. Here, the N sampled points along the raypath are grouped into N_c clusters bounded by circles. Each circle contains only a subset (roughly N/N_c) of the N sampled points. To find the point on the raypath that is nearest to a specified point (x, z), begin by choosing one of the circles. A good choice is the circle that contains a point previously found to be closest to a nearby (x, z). Interrogate (find the nearest point inside) this first circle, and remember the minimum distance. Then, for all remaining circles, interrogate the circle only if the circle's radius plus the minimum distance found so far is less than

the distance from the circle's center to the point (x, z). In other words, look inside only those circles that could possibly contain a point closer to (x, z) than the closest point found so far.

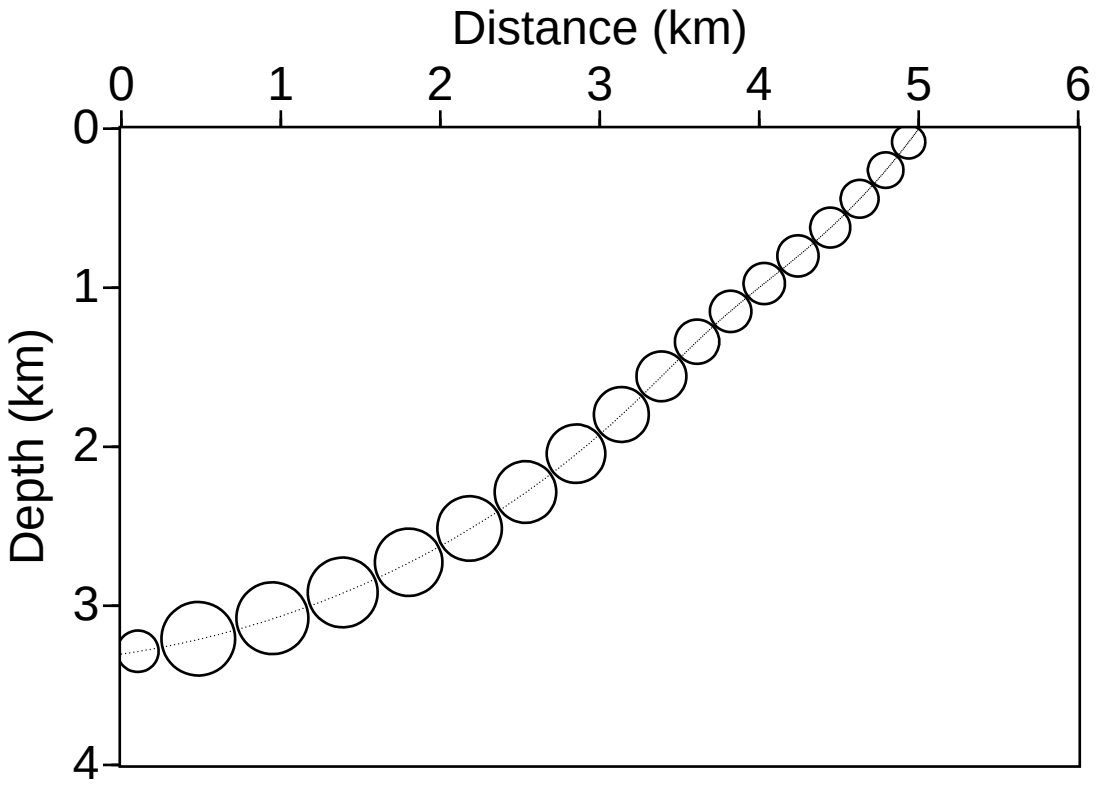

FIG. 4. Sampled points along a raypath are grouped into circular clusters to facilitate the search for that point on the raypath that is nearest to an arbitrary point (x, z). The points inside any given circle can be ignored during this search if the minimum distance found so far is less than the distance from (x, z) to the circle's boundary.

The cost of this method depends on the number of circles and on the number of sampled points inside each circle. If we make one ($N_c = 1$) big circle containing all of the sample points, then the cost of finding the nearest point is $O(N)$, which is no more efficient than the simplest search. Alternatively, if we choose $N_c = N$ tiny circles, each containing only one sampled point, then the cost is again $O(N)$, because we must interrogate all N circles. The most efficient use of the circles lies between these two extreme choices.

In general, the cost of the circle-based solution to the nearest point problem is roughly $O(N_c + kN/N_c)$, where k is the number of circles that must be interrogated. If the first circle interrogated contains the desired nearest point, then perhaps none of the remaining circles need to be interrogated. If we suppose that this $k = 1$ ideal case occurs frequently as we iterate over (x, z) coordinates, then the cost of the circle-based search is minimized by choosing $N_c = \sqrt{N}$, and the cost becomes $O(\sqrt{N})$. For large

N (say, $N > 100$), this cost is significantly less than the $O(N)$ cost of the simplest search method.

The raypath in Figure 4 was sampled at $N = 316$ points. Therefore, there are $\sqrt{N} \approx 18$ circles in Figure 4, and each contains 18 samples of the raypath, except for the last circle at the end of the raypath, which contains only 10 samples. To find the sample nearest to a point (x, z), each of the 18 circles must be checked to determine whether or not it could possibly contain the nearest sample.

The most attractive feature of the circle-based search method is that it can be trivially extended to three dimensions, in which the circles become spheres.

For either two or three dimensions, the circle-based search method adds a factor $\sqrt{N}$ to the total cost of accumulating a beam. To keep this $\sqrt{N}$ factor from becoming significant for typical N, the nearest point problem should be solved for only a subset of the (x, z) samples within a Gaussian beam. As suggested by Hill (1990), the complex time and amplitude functions may be computed on a relatively coarse grid of (x, z) coordinates, and then bilinearly interpolated for use in equation (1). My experiments suggest that these functions can be sampled about 8 times more coarsely in each spatial dimension than the beam itself, which implies an $8^2 = 64$ fold reduction in the cost of time and amplitude calculations for 2-D grids, and an $8^3 = 512$ fold reduction for 3-D grids. These reductions make the cost of the circle-based search an insignificant portion of the total cost of accumulating the contribution of a Gaussian beam.

WORKING INSIDE THE BEAM

One of the most useful properties of Gaussian beams is their compactness. The amplitude of the beam in Figure 2 decays exponentially with distance-squared (n^2) from the central ray (Hill, 1990). For efficient Gaussian beam migration, we should exploit this compactness by ignoring points (x, z) where the exponential decays to less than, say, one percent of its peak value. In other words, we should solve the nearest point problem and compute each beam's contribution to the subsurface image for only those points (x, z) where that contribution is significant.

Figure 5 shows the superposition of a coarsely sampled grid on the Gaussian beam of Figure 1. Note that this grid does not include all points (x, z); rather, it includes only those points where the beam amplitude is significant. As discussed in the preceding section, complex times and amplitudes should be computed at the corners of each cell of such a grid and then bilinearly interpolated for use in equation (1). Computations should be restricted to only those cells having significant amplitudes at all four corners. In other words, we should work only inside the beam.

As Figure 5 illustrates, the grid of cells within the beam may be irregularly shaped, so that it may be difficult to determine the bounds of simple loops over these cells in computer programs. For this particular beam, one might determine bounds in depth

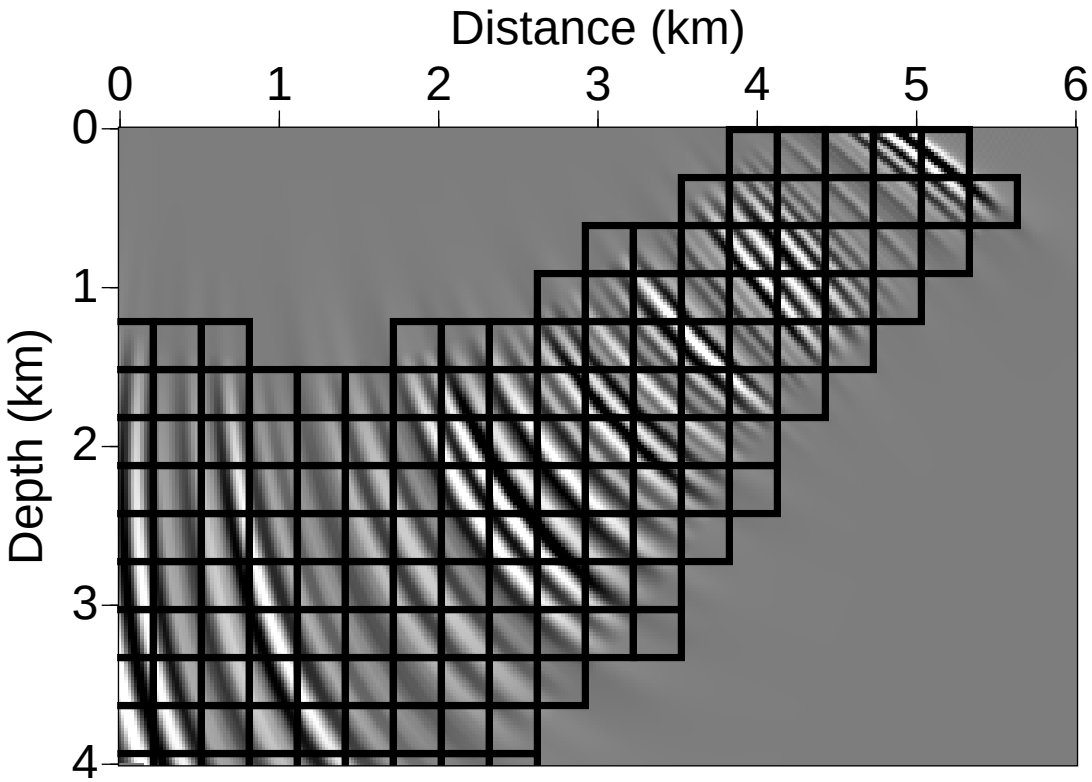

FIG. 5. Coarse grid of cells for which the amplitude of the Gaussian beam is significant. The irregular shape of this grid suggests that a recursive algorithm should be used to move from cell to cell, while accumulating the contribution of the beam within each cell.

z for each horizontal distance x within the beam. For other beams, however, multiple loops over disjoint sets of z coordinates might be required for a single x coordinate. Such loops over cells within an irregularly shaped beam become even more difficult to organize in three dimensions.

A simple solution to the problem of working within the beam is to use recursion instead of loops over cells. Begin with the corner of the cell that is nearest to the point where the raypath intersects the surface $z = 0$. This corresponds to the cell corner at a distance of about 5 km in Figure 5. For any reasonable beam width, the Gaussian beam amplitude at this first cell corner is guaranteed to be significant. Compute the complex time and amplitude for this first cell corner. Then, compute the times and amplitudes for the neighboring cell corners left, right, and below the first one. If the amplitude at a neighbor is significant, then compute times and amplitudes for that neighbor's neighbors. Repeat this recursive process until times and amplitudes for all cell corners with significant amplitudes have been computed.

The computer software that I use to set the complex time and amplitude at each cell corner is written in the C programming language as follows:

```
void setCell (Cells *cells, int jx, int jz)
/*****************************************************************************
Set a cell by computing the Gaussian beam complex time and amplitude of
its upper left corner.  If the amplitude is non-zero, set neighboring
cells recursively.
*****************************************************************************
Input:
cells           pointer to cells
jx              x index of the cell to set
jz              z index of the cell to set
*****************************************************************************
Notes:
To reduce the amount of memory required for recursion, the actual
computation of complex time and amplitude is performed by the function
cellTimeAmp(), so that no local variables are required in this
function, except for the input arguments themselves.
*****************************************************************************/
{
        /* If cell is out of bounds, return. */
        if (jx<0 || jx>=cells->mx || jz<0 || jz>=cells->mz) return;

        /* If cell is live, return. */
        if (cells->cell[jx][jz].live==cells->live) return;

        /* Make cell live. */
        cells->cell[jx][jz].live = cells->live;

        /* Compute complex time and amplitude.  If amplitude is
         * big enough, recursively set neighboring cell. */
        if (cellTimeAmp(cells,jx,jz)) {
                setCell(cells,jx+1,jz);
                setCell(cells,jx-1,jz);
                setCell(cells,jx,jz+1);
                setCell(cells,jx,jz-1);
        }
}
```

This `setCell` function "sets a cell" by computing the complex time and amplitude of its upper left corner. The first statement in this function ensures that the cell lies within an array of cells that spans the entire subsurface image. The second statement checks to see whether or not the cell has already been set. The third statement marks the cell as being "live". A live cell is a cell that has been (or, more precisely, soon will be) set. The actual work of computing the complex time and amplitude is done by the function `cellTimeAmp`. If the amplitude is significant, `cellTimeAmp` returns `True` and the function `setCell` calls itself four times, once for each of its neighbors left, right, below, and above.

The type `Cells` has been defined elsewhere to be a C structure containing all of the information required to compute times and amplitudes, including the results of dynamic ray tracing sampled at points along the raypath, the circular clusters used to find the nearest point on the raypath, as well as the grid of cells itself. The variable `cells` is a pointer to (i.e., address of) this structure.

The `live` flag is used to avoid redundant computations and, even more importantly, to ensure that the recursion ends. Note that at least one of the four calls to the function `setCell` is unnecessary, in principle, because the only way to set any cell (except for the first cell) is to have already set one of its four neighbors. However,

rather than testing to see which of the four neighbors is live, it is simpler to have the function `setCell` return if the cell has already been set.

For convenience, the grid of cells is represented by a two-dimensional array that spans the entire subsurface image, not just the region within the beam. To avoid having to initially turn off the live flag in every cell, which would require working *outside* the beam, a pseudo-random integer is used to denote a live cell. For each beam, a unique pseudo-random integer is generated. That integer is then used as the live flag for every cell within the beam. A new and different integer is used for each beam so that the live flags do not need to be turned off before processing the next beam.

In my computer program that performs Gaussian beam migration, the function `setCell` is called explicitly only once for each beam to start the recursion. All other calls to `setCell` are made by the function `setCell` itself.

After complex times and amplitudes have been computed at the corners of each cell within the beam, we may then accumulate the contribution of the beam to the subsurface image. Again, recursion is used to visit all of the cells that contribute significantly, starting with the cell nearest the raypath at the surface. Only cells with four live corners are used. Within each cell, simple loops over finely sampled x and z coordinates are used to accumulate the cell's contribution. Cells that have been processed in this way are marked with another psuedo-random flag (different from the live flag) that is unique for each beam.

BENCHMARKS

To analyze the cost of Gaussian beam depth migration, a series of benchmark tests were conducted using a computer program based on the computational methods described above. All tests were conducted for equal numbers of time, horizontal distance, and depth samples, so that computation time could be measured as a function of a single size N. In this way, the time required for Gaussian beam migration was measured for sizes $N = 100$, 200, 400, and 800. All tests were performed on an IBM POWERstation 520 workstation.

For comparison, the same sizes N were used to measure the cost of depth migration via explicit depth extrapolation filters applied in the frequency domain, as described by Hale (1991). This explicit method has a highly optimized inner loop, which in these tests consisted of 20 complex multiplies and 38 complex adds. This inner loop must be executed for all frequencies, horizontal distances, and depths. Therefore, the number of times that this innermost loop is executed is proportional to N^3.

The results of these tests are plotted in Figure 6. As indicated in Figure 6a, for all sizes N tested, the explicit extrapolation method requires less computation time than the Gaussian beam method. However, Figure 6b shows that the cost of the explicit method grows at a faster rate than that of the Gaussian beam method. The slopes of

the almost linear curves in the log-log plots of Figure 6b indicate that the cost of the explicit method is roughly proportional to $N^{2.7}$, while the cost of the Gaussian beam method is approximately proportional to $N^{2.2}$, for the range of N used in these tests.

To highlight the relative costs of the two computer programs, the ratios of the times required for the explicit program to the times required for the Gaussian beam program are plotted in Figure 7. For the largest size tested here, $N = 800$, the Gaussian beam program is about 20 percent slower than the explicit program.

We must be careful not to conclude from these benchmark tests that Gaussian beam migration is inherently slower than migration via explicit extrapolation filters. Rather, we should only conclude that, for the sizes tested, my current computer program for Gaussian beam migration is slower than my current program for the explicit method.

The most interesting results of these tests are the slopes of the almost linear curves in Figure 6b, not their intercepts. Optimization of the innermost loops of either computer program may significantly change the intercepts, but is unlikely to change the slopes of these curves significantly.

CONCLUSION

Two of the most difficult (or at least awkward) computational problems in Gaussian beam migration are (1) computation of complex-valued beam times and amplitudes in Cartesian (x, z) coordinates, and (2) performing calculations for only those (x, z) coordinates within the region where beam amplitudes are significant.

One way to simplify these problems might be to restrict the use of Gaussian beam migration to special velocity functions. For example, subsurface models with layers in which velocity or the velocity gradient is constant might facilitate the mapping from ray-centered (s, n) coordinates to Cartesian (x, z) coordinates, because the raypath within each such layer could be expressed analytically.

The algorithms proposed in this paper were selected, in part, to make such restrictions unnecessary. For example, an efficient solution to the first problem requires only that the results of dynamic ray tracing be well-sampled along rays traced from

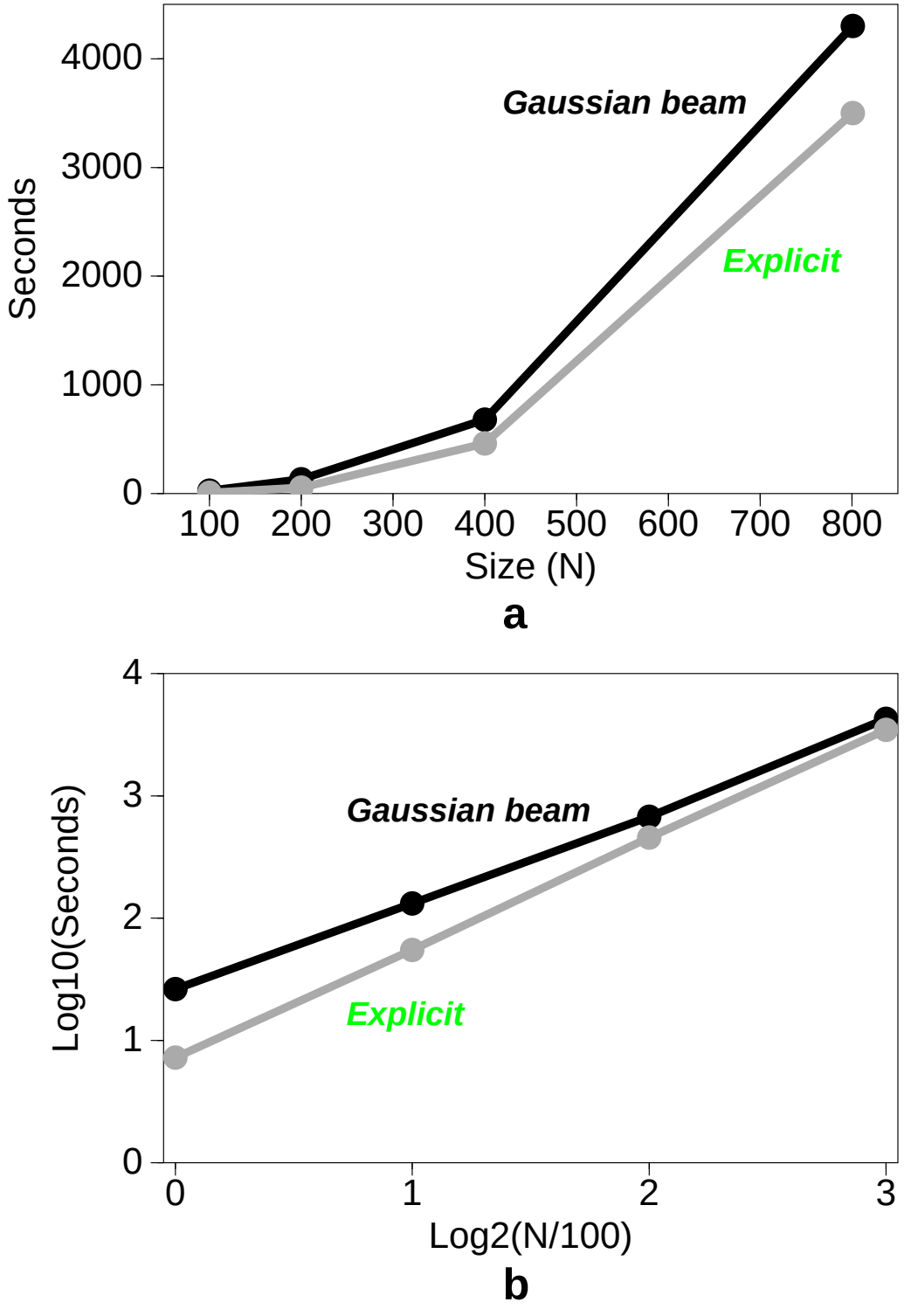

FIG. 6. Benchmark tests of two computer programs for depth migration. Computation times for the Gaussian beam program are (a) greater than, but (b) increase at a slower rate than those for the explicit extrapolation program. Log10 and Log2 denote logarithms for bases 10 and 2, respectively.

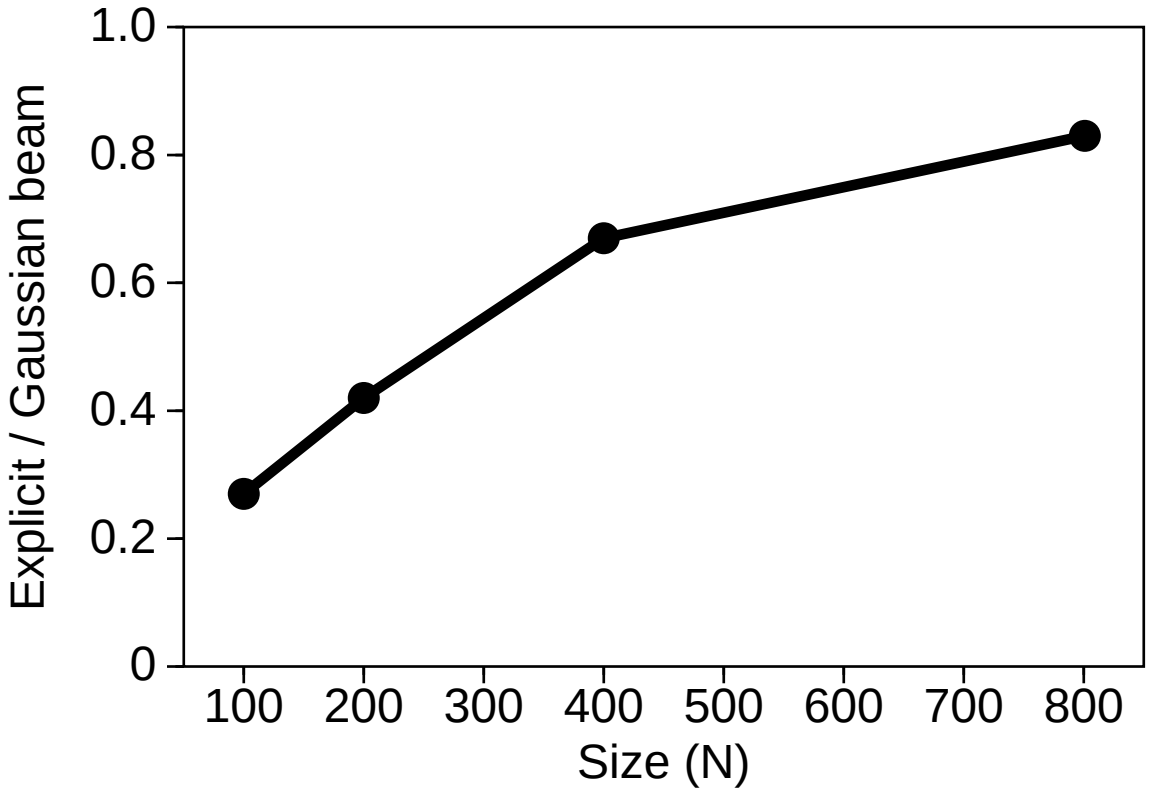

FIG. 7. Ratios of the computation times plotted in Figure 6a.

the surface. The algorithms proposed here do not depend on a specific representation of velocity or a particular method for ray tracing.

The algorithms were also chosen to facilitate their extension to 3-D migration and 2-D prestack migration of non-zero-offset sections. The square cells used in 2-D migration simply become cubic cells in 3-D migration. For non-zero-offset data, beam times and amplitudes must be computed for both source and receiver beams (Hill, et al., 1991). By computing the required times and amplitudes on a coarse grid of cells, as suggested by Hill (1990), we can reduce the cost of these computations. Furthermore, by using recursion to move from cell to cell, we can limit subsequent computations to only those cells where both source and receiver beam amplitudes are significant. Extensions to 3-D poststack depth migration and 2-D prestack depth migration, while not trivial, are straightforward.

The results of benchmark tests suggest that Gaussian beam migration is competitive with the most efficient depth migration methods. Although my current implementation of Gaussian beam migration is slower than a highly optimized depth migration based on explicit extrapolation filters, benchmark timings suggest that Gaussian beam migration may be most efficient for large data sizes (requiring more

than one hour on a typical computer workstation). As data dimensions increase, the computational cost of Gaussian beam migration grows more slowly than that of the explicit method.

ACKNOWLEDGMENTS

Support for this work was provided by the members of the Consortium Project on Seismic Inverse Methods for Complex Structures at the Center for Wave Phenomena, Colorado School of Mines, Support was also provided by the United States Department of Energy, Grant Number DE-FG02-89ER14079. (This support does not constitute an endorsement by DOE of the views expressed in this paper.)

REFERENCES

Červený, V., Popov, M. M., and Pšenčík, I., 1982, Computation of wave fields in inhomogeneous media — Gaussian beam approach: Geophys. J. R. astr. Soc., **70**, 109–128.

Červený, V., and Pšenčík, I., 1984, Gaussian beams in elastic 2-D laterally varying layered structures: Geophys. J. R. astr. Soc., **78**, 65–91.

Hale, D., 1991, Stable explicit depth extrapolation of seismic wavefields: Geophysics, **56**, 1770–1777.

Hale, D., 1992, Migration by the Kirchhoff, slant stack and Gaussian beam methods: this report.

Hill, N. R., 1990, Gaussian beam migration: Geophysics, **55**, 1416–1428.

Hill, N. R., Watson, T. H., Hassler, M. H., and Sisemore, L. K., 1991, Salt-flank imaging using Gaussian beam migration: Presented at the 61st Ann. Internat. Mtg. Soc. Expl. Geophys., Expanded Abstracts, 1178–1180.

Preparata, F. P., and Shamos, M. I., 1985, Computational geometry — an introduction: Springer-Verlag.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., 1986, Numerical recipes — the art of scientific computing: Cambridge University Press.

Thomson, C. J., and Gubbins, D., 1982, Three-dimensional lithospheric modelling at NORSAR: linearity of the method and amplitude variations from the anomalies: Geophys. J. R. astr. Soc., **71**, 1–36.

Watson, D. F., 1981, Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes: The Computer Journal, **24**, no. 2, 167–172.