SPHERICAL RAY-TRACING

BY

DAVID A. HANNASCH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Assistant Professor Matthew Turk

Abstract

Many scientific datasets (for example, simulations and reconstructions in astrophysics and geophysics) are spheres or sections of spheres and naturally fit into spherical coordinates.

When tracing rays through naturally spherical domains (for example, for visualization), converting to Cartesian coordinates introduces aberrations. Therefore, we seek an algorithm to natively cast rays through spherical coordinates, and without performing calculations that would amount to in-place conversion to Cartesian.

Three things make this work:

1. An alternative coordinate system which is isomorphic to spherical coordinates but has useful properties along straight lines.
2. Constantly referring back to the equation of the line a ray is a segment of, to control propagation of errors.
3. Using the shape of the cells to restrict where the ray might exit a cell so that we can (in some cases) select the next zone by process of elimination.

This method applies also to cylindrical coordinates (or even more simply, 2-d polar coordinates).

Acknowledgments

This research was supported by a fellowship from the National Physical Science Consortium and the Department of Defense.

Professor Turk provided the original topic as well as revisions to this document itself, and personally taught me a great deal about the broader context of how scientific visualization can allow us to ask deeper questions of data. Much helpful consultation came from the members of the Data Exploration Laboratory, especially Dr. Nathan Goldbaum. The National Center for Supercomputing Applications, in particular Amanda Lombardo and Tedra Tuttle, generously provided space right next to Professor Turk and the rest of the DXL, greatly aiding this research.

We would also like to thank Dr. Ben Holtzman of Columbia University, Dr. Martin James Pratt of Washington University, Mark Van Moer of the National Center for Supercomputing Applications, and Scott Noble of the University of Tulsa; without application scientists running simulations, postprocessing would be pointless.

Thanks are also due to Professor Alexandra Kolla for providing an extra set of eyes on short notice.

Table of Contents

Chapter 1

Introduction: the last-mile problem

Computational science provides us with many tools for simulating physical processes to high fidelity. But ultimately the purpose of these simulations is to support human decision-making. If the end result cannot be delivered in a human-comprehensible form, then the accuracy in the machine is wasted. Accuracy is even more important when a numerical analyst is trying to debug the computation itself. Seeing is believing; the most important judgement of whether the system is working comes when someone looks at the output. But whether an investigator is using the data to answer a scientific question or a numerical analyst is asking a question about validity, there is an implied warranty that *looking at the data* won't be misleading. That guarantee isn't always satisfied. Scientific simulations are always becoming more sophisticated, representing a greater variety of structures with a set of raw bytes. Visualization techniques must keep pace. Ray-marching is the particular step in the process that we concern ourselves with here — marching ray representing sightlines through a three-dimensional domain. We present a new method for marching rays through non-Cartesian domains without the need for the usual preprocessing of the data, as that preprocessing can introduce errors.

To visualize a three-dimensional domain, we associate with each pixel a ray pointing forward from the viewer's virtual location. Typically, we represent a ray as a starting point $\vec{p}$ where the pixel is and a direction vector $\vec{d}$ so that the points on the line are $\vec{p}+t\vec{d}$. The domain appears as if it were a physical, translucent object. If the screen is a flat plane (like a typical monitor), then we can precompute a couple quantities based on the direction vector $\vec{d}$, which is then the same for all rays. This will be useful, for example, when converting between the distance-from-$\vec{p}$ and the azimuth angle. However, the rays need not all point straight forward for the method to work. Certain kinds of virtual 'lenses' make use of this. For example, if a position is assumed for the human eye looking at the screen, then we can angle the rays so that the ray pointing from the pixel is parallel to the line from the human eye to the pixel on the screen instead of being parallel to all the other rays. This is called a perspective lens in the literature.

Definition 1. A ray is a continuous subset of a line extending infinitely in one direction only. That is, we are using the mathematics definition of a ray, not physical light rays that can bounce as in some other literature on ray-tracing.

Ray-marching refers to finding the intersection between a ray and some sets, usually cells in a discretization of a domain.

The literature also uses the terms ray-tracing and ray-casting interchangeably with ray-marching, but the reader should be warned that ray-tracing is also used to refer to techniques for bouncing rays off a series of surfaces. Of course, that includes ray-casting as a necessary subcomponent, so ray-tracing in that sense could be considered an application of ray-casting. Meanwhile, ray-casting has somewhat different connotations that ray-marching: ray-casting often refers to literally intersecting rays with surfaces. Those surfaces could be the boundaries of cells in a grid, so in that sense ray-marching is a special case of ray-casting. However, it

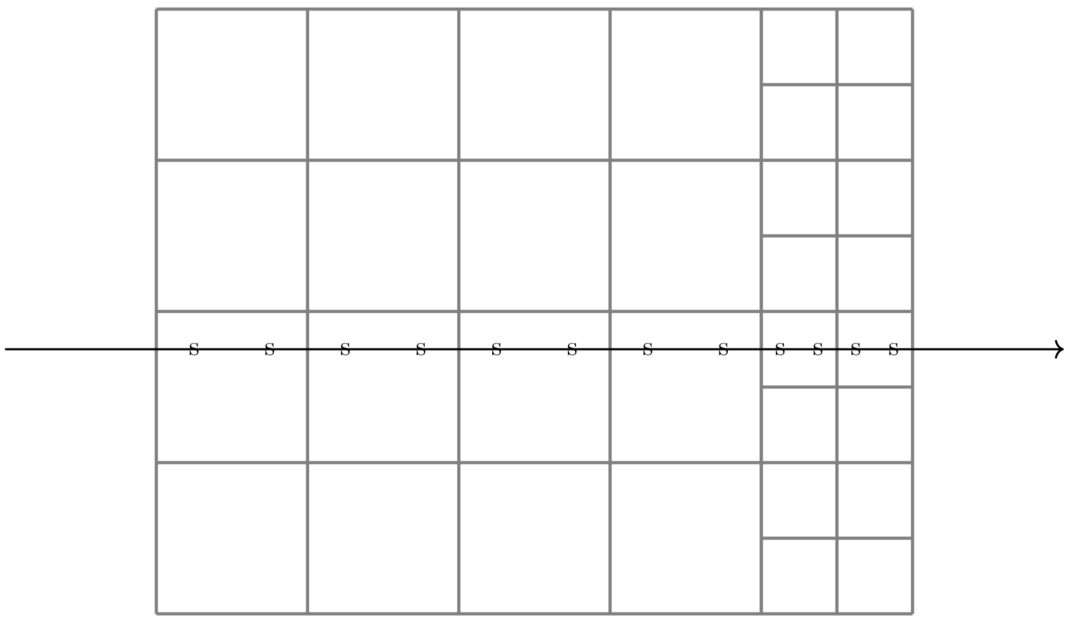


Figure 1.1: Ray-marching. s denotes a point at which the sampler function is called. Note that the density of sample points depends on the density of the simulation grid; here there are two samples per cell.

is important to note that for our purposes each pixel depends on every cell hit by its corresponding ray, not just the first nonzero cell, as is often the expectation with ray-casting.

The difference between ray-casting only to surfaces, and full ray-marching, is critical. Representing three-dimensional data in a two-dimensional image is a fundamental problem, due to occlusion. Virtual-reality interfaces partially mitigate this by generating two different images for two different eyes — but only partially. Asking a scientist to examine the data by walking around in it is like asking them to map out a city by walking around in it. We can do better than that. **Volume rendering** treats the entire field of data as translucent, allowing the entire domain to be seen at once. The algorithm we will present could be used simply for ray-casting to the first opaque surface, but volume-rendering is the application our algorithm is designed for. It is optimized around the assumption that any given ray will continue marching to the end of the domain, and if the ray encounters a hole in the domain, the ray will continue on the far side of the hole.

Mathematically, we can say that ray-marching takes a ray and a family of sets (the cells in our grid) and finds the length of the intersection of the ray with each of the sets. Of course, the intersection of the ray with most sets will be empty, with length zero. The pixel corresponding to that ray can depend on all of those lengths and on the data stored in the cells with nonzero intersections. In practice, we do not store all those lengths and pass them all off to an aggregation function; instead, a 'sampler function' ingests the lengths one at a time.

A sampler function $f_D(C, \vec{v}_{\text{enter}}, \vec{v}_{\text{exit}})$ on a domain D takes a cell C along the points at which the ray enters and exits the cell. If the underlying simulation data is cell-centered rather than vertex-centered, a sampler function may take only the length of the intersection of the ray with the cell; this is a special case of the above, but is allows the sampler function to do without an `FSQRT` if what it needs is the length.

Usually, the sampler function is called in the interior of a cell, and thus the sampler function must assume a structure (often piecewise linear) to the field data and interpolate from the field values at the vertices. Often, the sampler function will be called a constant number of times per cell, as shown in Fig. 1.1.

Regardless, the density of the underlying simulation grid is used as a guide — all the reasons why that region needed extra care in simulation are also reasons why that region needs extra care in visualization.

The pixel value is then determined by an aggregation function $A(s_0, s_1, \ldots)$ that takes the output from all calls to the sampler function. The sampler function should assign a value to any location outside the domain that is 'null' with respect to the aggregation function, so that 'empty space' is always perfectly transparent, since an investigator will typically look at the data first from the outside. As a running example, a sampler function $f_D(C, \vec{v}_{\text{enter}}, \vec{v}_{\text{exit}}) \in [0,1] \times \mathbb{R}^+$ can define an opacity (light-blocking) and an emissivity (light-generation, in this case only intensity, though this could have a color as well) for the ray's progress through the cell. The aggregation function in this case is a weighted sum of the emissivities of each of the sample points, with the weight of each point being the product of the opacities of all the points between a point and the pixel. This is in accordance with the radiative transfer equation:

$$\frac{dI}{dt} = s - \alpha I$$

where s is the new emission, α is the opacity, and t is the distance along the ray, *not* time. Outside the domain, the opacity and emissivity are both zero, so that the emission propagates unchanged.

As an example of an application in practice, a sampler function used in the software package yt[TSO$^+$11], `yt.utilities.lib.image_samplers.VolumeRenderSampler.sample`, takes `n_samples` (usually five or ten) evenly-spaced samples within each cell. It assumes values are vertex-centered and that the field is linear in each coordinate, so that the values at each of the n sample points are actually calculated by interpolating from the six surrounding vertices. Note that in Cartesian this means we could, if we so chose, run with the assumption of linearity and skip all of the sample points: if the field value is linear in Cartesian coordinates, then it's linear in t as we progress along $\vec{p} + t\vec{d}$, and we can trivially analytically integrate a linear function. If the field value is $f(\vec{v}_{\text{enter}})$ at $\vec{v}_{\text{enter}}$ and $f(\vec{v}_{\text{exit}})$ at $\vec{v}_{\text{exit}}$, the linear interpolant between them is

$$\begin{aligned} f(\vec{v}_{\text{enter}})\frac{t - T_2}{T_1 - T_2} + f(\vec{v}_{\text{exit}})\frac{t - T_1}{T_2 - T_1} &= \left(\frac{f(\vec{v}_{\text{enter}})}{T_1 - T_2} + \frac{f(\vec{v}_{\text{exit}})}{T_2 - T_1}\right) t - \frac{T_2 f(\vec{v}_{\text{enter}})}{T_1 - T_2} - \frac{f(\vec{v}_{\text{exit}}) T_1}{T_2 - T_1} \\ &= \frac{f(\vec{v}_{\text{exit}}) - f(\vec{v}_{\text{enter}})}{T_2 - T_1} t + \frac{T_2 f(\vec{v}_{\text{enter}}) - T_1 f(\vec{v}_{\text{exit}})}{T_2 - T_1}. \end{aligned}$$

Defining

$$a := \frac{f(\vec{v}_{\text{exit}}) - f(\vec{v}_{\text{enter}})}{T_2 - T_1} \qquad \text{and} \qquad b := \frac{T_2 f(\vec{v}_{\text{enter}}) - T_1 f(\vec{v}_{\text{exit}})}{T_2 - T_1}$$

we can analytically integrate across the cell: $\displaystyle\int_{T_1}^{T_2} (at + b)\,\mathrm{d}t = \left[t^2 a/2 + bt\right]_{T_1}^{T_2}$. If we assume that the field value is linear in each of the three *spherical* coordinates, however, this option is unavailable, because it does not follow that the field is linear along the path of a ray.

1.1 Domains: spherical and the others

The standard procedure for intersecting a line with a cell is Liang-Barsky serial clipping [Har16], which finds the t-value at which the line intersects each plane defining the boundary of the cell. Liang-Barsky serial clipping makes two assumptions. If any coordinate is not monotone along a line of interest, then the algorithm will simply fail. More subtly, it assumes that intersecting a line with a facet is not expensive —

which is true when facets are flat planes, and all facets of Cartesian grid cells are flat planes. Spherical grid cells, however, have some boundaries that are not planes, and we save time by minimizing the number of times we must calculate the intersection of a line with a nonplanar facet. Moreover, the zenith angle and radius are not monotone along lines, so the standard algorithm cannot work directly on spherical grids.

```
  function INTERSECT(line, facet)
      return $t$ at which the line intersects the facet
  end function
  procedure LIANG-BARSKY SERIAL CLIPPING($t_{\text{start}} = 0, t_{\text{stop}} = 1$)
      for coordinate in $x, y, z$ do
Require: coordinate is monotone along the line
          if coordinate is increasing along the line then
              first facet ← lower facet
              second facet ← upper facet
          else
              first facet ← upper facet
              second facet ← lower facet
          end if
          $t_{\text{start}} \leftarrow \max(\text{INTERSECT}(\text{line, first facet}), t_{\text{start}})$
          $t_{\text{stop}} \leftarrow \min(\text{INTERSECT}(\text{line, second facet}), t_{\text{stop}})$
      end for
  end procedure
```

So for a non-Cartesian discretization, the usual procedure is to convert the domain into Cartesian boxes and then march rays through the resulting Cartesian grid. There are two related problems. When converting the data, some error will be inevitably introduced, since the borders of the Cartesian boxes don't match the borders of the native-resolution cells. We will quantify this in Chapter 6. The tiling by Cartesian boxes must have *higher* resolution than the pre-existing mesh, and it's rarely clear a priori what resolution we will need. To know the necessary resolution to get this error down below the underlying discretization error of the problem, we would need a theoretical upper bound that does not currently exist. Worse, a non-Cartesian discretization usually has, by design, variation in cell sizes — spherical grids have smaller cells near the center and larger cells far from the center. So the Cartesian resampling has two options. It can use much higher resolution than it needs, which is expensive. Or it can implement an adaptive algorithm for refining the resampling mesh based on the sizes and shapes of the cells, and the values of the field data (How quickly is it varying?), and the preferred interpolation of that field (Do we expect this field to be linear or quadratic or exponential? If the simulation is modeling the field as varying with the radius in a certain way, how does that translate to how it varies with Cartesian coordinates?), which is possible but introduces complications and the potential for unexpected and potentially *undetected* failures on novel data.

An example may help. Fig. 1.2 shows a simulation of a magnetic field, which should, and does, show clearly defined elliptical bands. These bands are very close together near the center, but with angular coordinates they are still easily differentiable — which is important, because if the bands became garbled near the center, the larger banded structure would not appear.

But if we look directly at the center, the bands blend together. The vector graphics language is capable of rendering curves exactly, and the simulation code is capable of discretizing by angle so that the bands can at least be kept from curling around each other. But in between the simulation and the final image, the

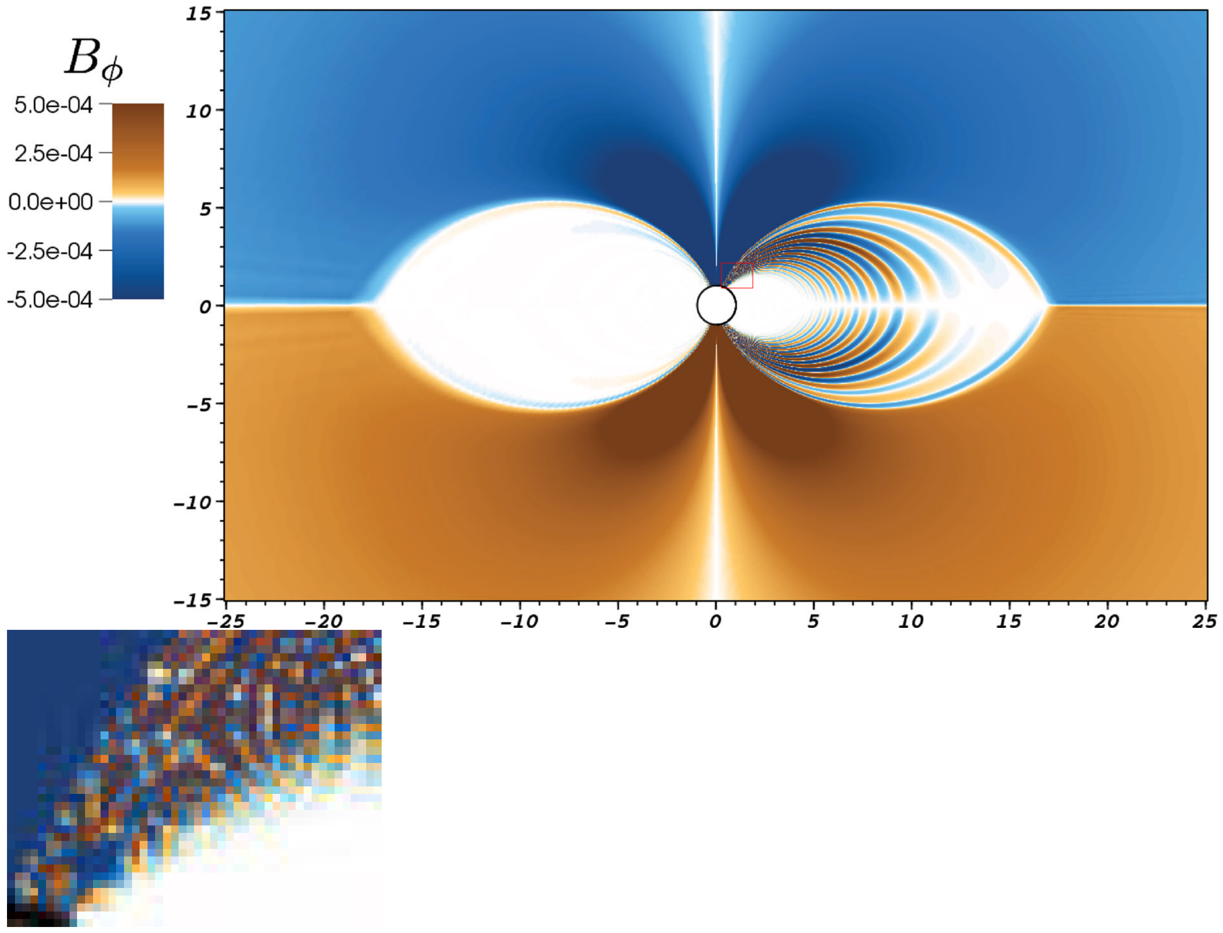


Figure 1.2: Comparing low resolution (256 × 155), left, and high resolution (768 × 507), right, simulations. The colour is toroidal magnetic field.[PBH12, page 1432]

$$\begin{tikzcd}\text{high-fidelity simulation} \arrow[r, "\text{Danger!}"] & \text{Cartesian resampling} \arrow[r, "\text{can preserve full accuracy}"] & \text{vector graphics language}\end{tikzcd}$$

domain has been converted to Cartesian coordinates, and rectangular boxes cannot be made to fit the shape of the true pattern.

Of course, in this case the overall structure is still clear thanks to the larger part of the picture, where the bands are wider. (Otherwise, this picture wouldn't have found its way into publication in the first place.) When the application scientist already knows the point they want to make, they can usually construct an image to make that point. We are more concerned about *data exploration*: When the application scientist *doesn't* know what they're going to find. Visualization is an excellent way to look for interesting patterns in three-dimensional data, but only if the visualization doesn't introduce subtle errors that obscure the truth. In data exploration, the scientist doesn't yet know what might be important; the algorithm certainly doesn't. And data exploration means generating *many* two-dimensional views into three-dimensional data, while the scientist is moving and rotating the view to look around.

Which brings us to the second problem: performance. We can use adaptive and high-resolution algorithms to tile domains with boxes and selectively refine as views are requested, but for reasons described later in Section 1.1.1, this has downsides.

There's a third concern that isn't so much a 'problem' with conversion as a reason why it just isn't done. Consider AxiSEM[NvS+14], short for Axially-symmetric Spectral Element Method. AxiSEM is a tool for calculating wave propagation through spherical domains. Despite the historical name, the field data need not be truly axially symmetric; rather, there is a perturbation function that describes the change as you proceed around the axis. Before we can even begin to cube that domain, we must decide how to discretize the perturbation function (a difficult problem with a different answer for each perturbation function), and turn what was essentially a two-dimensional domain into a fully three-dimensional domain, losing the benefit of the axial pseudo-symmetry by going from a quadratic number of elements to a cubic number of elements.

Moreover, the perturbation function is often highly oscillatory, so we run the risk that the 'cubing' process might sample the function at a nonuniform set of points, yielding misleading results. This is a problem that can also occur when discretizing the spherical domain for simulation in the first place, but it is a *difficult* problem, to deal with case-by-case, that the application scientists should not have to deal with twice.

The result of these complications is that many visualization frontends, such as yt, are unable to visualize AxiSEM data.

1.1.1 Performance

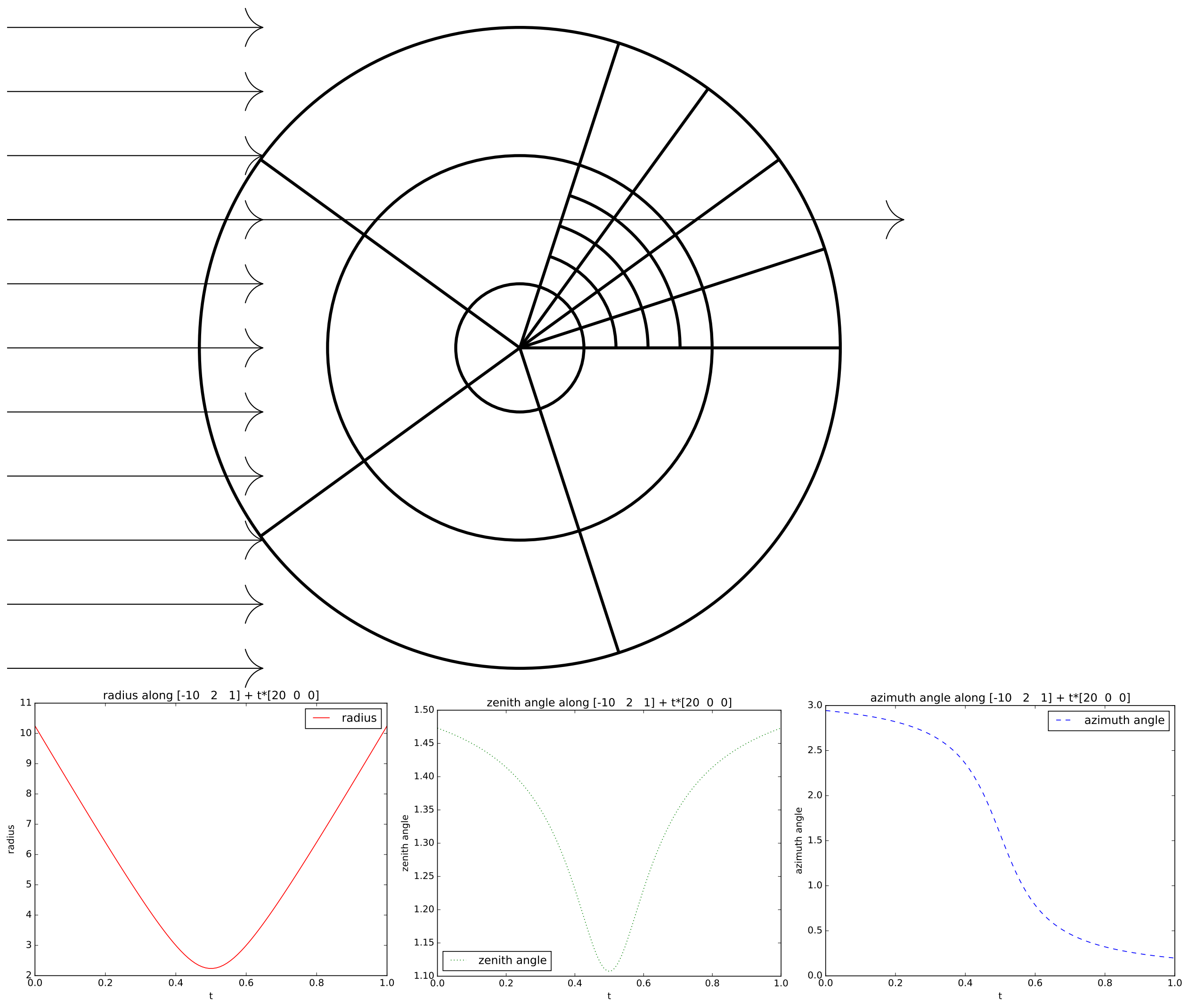
Our standard use-case is a scientist looking around a data set by rotating and shifting a view in real time.

When a scientist is rotating a view using the mouse, we only have 10^{-2}–10^{-1} seconds to render each frame;[CMN83] (cf [LH14]) any longer and the unpredictable lag from user input to program output starts to interfere with the human brain's ability to grasp the full three-dimensional shape from the views. *Predictable* lag may be less of a problem, so an alternative might be to pad visualization time when it is low. For our purposes, real-time visualization needs to run *faster* than the original simulation on the same data, not slower, and usually needs to run on an ordinary computer rather than a supercomputer.

1.2 Last-mile error

There is a large volume of literature on reducing errors in simulations. To be sure, an error that occurs during a simulation may propagate forward and spoil every later step, while an error that occurs on the *last* step affects only one step.

Figure 1.3: Two-dimensional concept diagram of spherical ray-marching. Plots show the progress of radius, zenith angle and azimuth angle respectively for one of the rays.



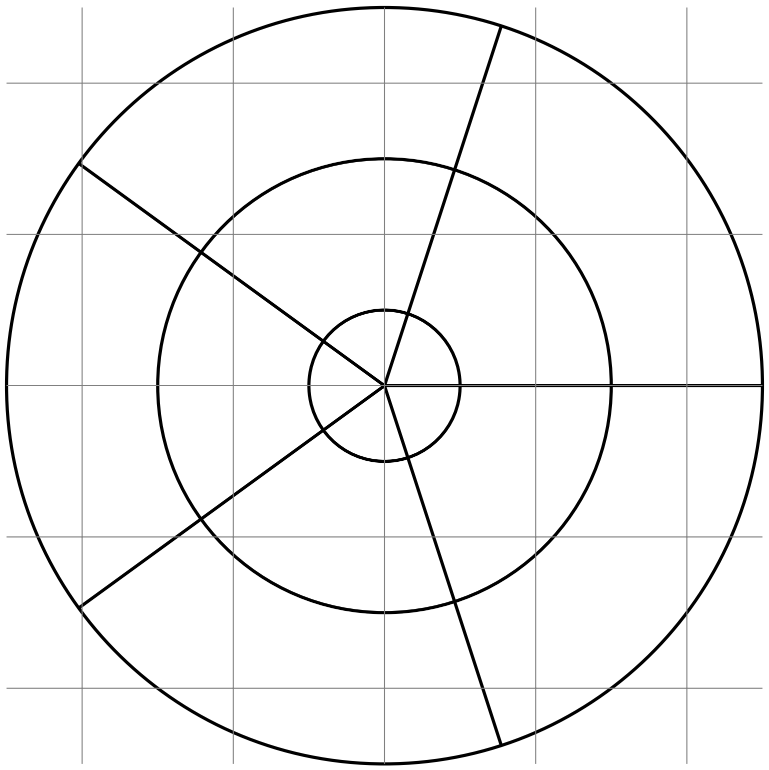


Figure 1.4: Cubing the sphere. If the field value represents, for example, density, then to preserve total mass we must set the field value at each of our new boxes by calculating its exact intersection with each cell — and if we're taking intersections with curved surfaces, then we might as well just work with the curved surfaces directly.

But that last step, when the data is delivered to an actual human, is the step that matters.[Tuf01]

To 'cube' a spherical grid to a Cartesian grid, we must cover each cell with boxes (though not necessarily cubes). Sampling at different points than the native data introduces error; worse, it's difficult to prove bounds on that error. Even merely changing the resolution can cause problems.[Jon99] Unless we use a complicated multiresolution cuber, the smaller inner cells will be undersampled and the larger outer cells will be oversampled relative to the true precision they represent. (When we have scores of small cubes repeating the same data from one outer spherical cell, we're just misleadingly multiplying data.)

1.3 Fineness of domain and screen

Traditional computer-graphics techniques are built around the idea that the screen is finer than the domain; that is, each triangle or polygon covers multiple pixels. When visualizing a high-performance computation on a 1024×768 screen, this is often not the case; there may indeed be cells that fall 'between rays' and thereby don't get touched at all. However, there are exceptions: a scientist may zoom in, or an adaptive algorithm may apply a coarse mesh over an 'unimportant' part of the domain.

Standard graphics would apply Gouraud interpolation[Har16, Chapter 14] to get a smooth transition for the pixels interior to a single cell of the domain. Even if there *are* pixels interior to each cell, this might turn out to be a mistake for scientific visualization. Most phenomena at most times might have smooth transitions, but if there *is* a sudden, sharp change, and a scientist zooms in on it, Gouraud interpolation would make it appear misleadingly smooth. Still, in general the actual simulation grid would likely be fine around any sharp swings, so we wouldn't have multple pixels per cell in the first place.

If we do have a very fine mesh and limited computational resources for visualization (for example, because we are updating the view in real time in response to user input), we can use techniques from computer graphics. The algorithm below already has an option to not quite compute the exact t-length in certain cells where doing so would be difficult. Rasterization does something more radical[Har16, Chapter 14, Bresenham line algorithm]: when advancing along a line, it advances in steps of constant length, and at each step, only asks which cell's center is nearest. (In their case the line is the object they want to visualize and the cells are the pixels, but the same priciples apply.)

It's worth noting that in case we *did* want to do something like that, the geometric center of a cell — the point of minimum maximum distance from any other point — is not the midpoint of each of the three spherical coordinates, as it is in Cartesian (see Section 6.1).

1.4 Previous work

Definition 2. A **facet** of a closed set is a maximal smooth (has differentiable tangent vector) subset of the boundary, so that the facet is delineated by a nondifferentiable sharp edge.

For example, a square in $\mathbb{R}^2$ has four facets, and more generally the facets of an n-dimensional polytope are its $(n-1)$-dimensional faces (the same as the definition of facets of a polytope in polyhedral combinatorics).

Hewett [Hew12, Chapter 4] used a generic programming paradigm, specifying a mesh-independent algorithm and using a compiler capable of taking the algorithm and a description of the mesh to generate the actual code. Their algorithm works by repeatedly finding the intersection of the ray with the facets of the current cell. In the case of nonconvex cells, it is necessary to sort those facets that intersect the ray so that we take the first exit.[Hew16] That is, just as with the Liang-Barsky algorithm we talked about in Section 1.1,

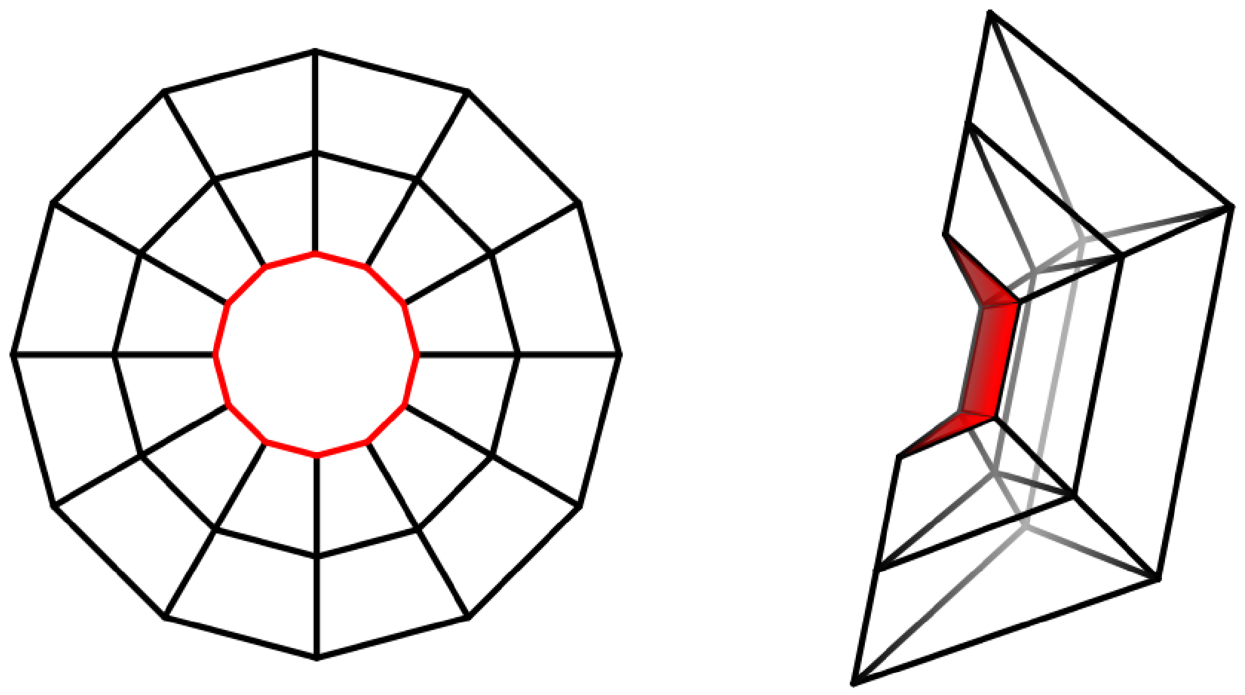


Figure 1.5: A ‘spherical’ mesh currently used for ray-tracing in solar tomography.[Hew12, page 109] With our new method, this can be replaced.

they still need to calculate the intersection, if any, of the ray with every facet of any cell it intersects, which is one reason they stick strictly to planar facets.

We are attempting to realize gains from specialization, without specializing too much. The broad class of meshes we are concerned with are meshes such that each cell is an intersection of intervals in *some* coordinate system. We call such a mesh a **grid**. Note that a grid need not be *regular*; we can easily have the grid be denser around critical regions. Of course, any mesh can be described as a ‘grid’ by defining coordinates appropriately, but usually those ‘coordinates’ would be discontinuous functions. In practice we are interested in coordinate systems that are easy to work with in some sense and that are, at minimum, continuous and reasonably smooth as we move about in space. This paper deals with spherical (or cylindrical) coordinates. Our spherical grids are permitted to have nonuniform divisions in each of the three coordinates. Although our method does not directly address more-refined local patches of grids (such as the upper-right corner of Fig. 1.3), the fact that we support nonuniform coordinate-interval sizes means that together with a rendering framework such as yt[TSO$^+$11], we can support arbitrarily refined grids by having the framework recursively call the algorithm on the patches of denser grid.

Almost all literature on ray-casting deals with planar facets, because they are easier to intersect. For example, Miranda and Celes[MC12] deal with unstructured meshes, but all cells must be hexahedral. Fuchs and Hjelmervik[FH16] recently provided a ray-casting algorithm for isogeometric models, where both the geometry and the scalar field are given by spline functions. The ray-surface intersections are solutions of nonlinear equations.

Ertl et alia[UFE10] cast rays onto curved surfaces, in fact surfaces described by higher-than-quadratic polynomials. Putting together such facets correctly could produce very general meshes, including our spherical grids. But this generality cost them dearly: they had to use an iterative Newton-Raphson solver to find ray-surface intersections. Although the radius and zenith angle can be intersected with rays using only quadratics, they approximated a sphere with a mesh consisting of 1290 curved cubic triangles[UFE10, page 344]. More recently Ertl et alia have been moving in a different direction, accelerating sampling along rays by analyzing the transfer function which maps field values to (emissivity, opacity) pairs and taking fewer samples in those cells which they determine will not have perceptible details.[BUS$^+$15]

Haimes et alia[NKH11] are working on a problem that is spiritually similar to ours, though focused on a different point of approximation: they are concerned about resampling high-order basis polynomials to linear when the visualization does only linear interpolation. ([BUS$^+$15] touched on this as well.)

> Linear approximations of high-order data are created by sampling the data at a specified set of points. If the sampling is performed too coarsely, then the approximation will be unable to resolve details in the underlying data, resulting in visualization errors. Conversely, if the sampling is too fine, while the details will be preserved and the visualization may not contain error, it will have used more processing time and other resources than necessary.[NKH11, page 1803]

In both cases, the problem is the same: the visualization can't handle the data directly, and approximations introduce errors, so we need to find a way to reroute around those resampling steps. Of course, their work in bringing forward the true basis elements is orthogonal to our work in bringing forward the true grid; both are necessary for the whole picture. The ElVis authors demonstrated the importance of allowing interpolation functions that fit the actual computation[NLKH12][NKH14]. Our algorithm leaves the precise interpolation unspecified as part of the sampler functions we call out to. For example, [LCTD14] used the equivalent of sampler functions with quadratic interpolation, though they were ray-casting in tetrahedral meshes.

Schollmeyer and Froehlich[SF14] generated a series of intervals that might contain ray-surface intersections, similar to the Las Vegas algorithm we will mention in Section 7.3.1, which we have not yet tested against our approach. They hypothesize (page 1238) that their speedup over traditional algorithms is due to processing all rays (meaning, sorting their intersections) in parallel via GPU. Our algorithm is certainly suited to processing all rays in parallel, though we have not yet implemented it using GPUs so we cannot say whether we will see similar results.

One of the recent accelerations of ray-tracing (in the sense of bouncing rays, though the part they accelerated was the ray-casting) was Nery et alia[NNFJ13], which exploits a special way to do ray-triangle intersections. That obviously won't work when trying to intersect rays with nonplanar surfaces such as spherical cells.

AVIS[MCZ$^+$16] for meteorological uses spherical volume-rendering and is likely another potential application of spherical ray-casting, although unfortunately we have not been able to obtain the full text of its description.

1.5 Ordering cells

There is one last alternative that should be addressed before we describe our new approach. We could simply calculate many points on the ray and find which cells those points lie in. If the grid is *regular*, then we can turn a set of coordinates into a cell address in $O(1)$ time, and of course we can turn a cell address into a field value in $O(1)$ time. However, this method is by nature imprecise; instead of finding the t-value when the ray enters and exits a cell to full machine precision, it requires another point-calculation for each additional bit of precision. On very fine grids, this may be worth trying, just to compare, although we already have options to compute faster by sacrificing precision — by simply reducing the resolution, for one. Of course, on non-regular grids, with cells of varying sizes, we often cannot transfrom a set of coordinates into a cell address in $O(1)$ time — the fact that sequential cells must be neighbors in the grid helps us there, but that tends toward restricting the sampling to be sequential rather than random-access. Moreover, many sampler functions require the cells in the proper order (for example, for transfer functions). Because of these complications, the most efficient approach — an adaptive algorithm that sampled random points on the ray and refined as needed according to how quickly the field was varying or some other criterion — probably would not work, unless we retained all the cells in memory and had an additional step to sort them, which would be its own cost. That leaves only the simplicity of marching forward in tiny sequential t-steps, and without knowing the precise t-values at which the ray enters and leaves cells, even if we could always determine the exact coordinates given a t-value (which we will be able to, for the alternate coordinate system given in Chapter 2), we need the cell address to get the bounds of the cell so we can turn the coordinates into normalized reference coordinates for the sampler function's interpolation. As noted in Section 1.4, a great deal of current work on reducing errors is centered on customizing the interpolation to the underlying data. This *could* work for a regular grid, though remember that a regular spherical grid does not have all cells the same size, even though the coordinate intervals are all the same — the cells nearer the origin are smaller, so that the grid is denser there and we might want to sample more finely there. But insisting on regular spherical grids would fall under the heading of 'specializing too much'. Indeed, many scientific datasets such as the ones we'll discuss in Chapter 7 are *not* regular, not even close. The available data can mix gaps of one kilometer with gaps of a hundred kilometers. We want to be able to handle that.

Conclusion There is a need for an inexpensive algorithm to march rays through spherical domains with high precision. Before we can give it to you, however, we must reorient the way we think about the spherical grid. We will transform the spherical grid into an alternate coordinate system in Chapter 2.

Chapter 2

Alternate coordinate system

We are looking at data in the form of spheres and sections of spheres. Our method also applies to cylindrical coordinates, as we'll note in Section 3.3, but the cylindrical case amounts to a blending of the spherical case and the Cartesian case, and the Cartesian case is already well-handled by other means, so the spherical case contains all the interesting differences from existing practices. Often, the data we are given comes from scientific simulations, but it can also come from reconstructions created from observations — for example, a reconstruction of the interior of the Earth from multiple readings around the globe. Regardless of what the data represents, they are divided into grids according to spherical coordinates.

Definition 3. Throughout this paper, θ will refer to the zenith angle in $[0, \pi]$ and ϕ will refer to the azimuth angle in $[0, 2\pi)$ (note the order; this is the convention most commonly used in physics).

r refers to the radius $\sqrt{x^2 + y^2 + z^2}$. See Fig. 2.1.

Working directly with spherical coordinates would slow us down too much, so we define an alternate coordinate system. We can make an analogy to using Euler roll-pitch-yaw coordinates for rotations versus using quaternions. Euler roll-pitch-yaw coordinates can certainly describe any single orientation. A single orientation as a point on the unit sphere does not completely specify a rotation even if we attempt to disambiguate by always taking the shortest rotation, because antipodal points on the sphere are connected by multiple geodesics, but we can disambiguate those arbitrarily since we don't care which geodesic we take when rotating. The real issue is that *computing* the rotation corresponding to an orientation becomes unnecessarily complicated, so roll-pitch-yaw coordinates are a poor choice when interpolating between two orientations. For interpolating one orientation with another orientation, the best coordinate system is usually quaternions, a three-dimensional manifold embedded in four-dimensional space. But quaternions might not be the best coordinate system for doing other things with orientations. Which coordinate system we use depends on what we want to do. Our purpose is marching rays, and so we want a coordinate system that facilitates that.

Definition 4. Define $\operatorname{sgnSqr} \alpha$ to be the signed square $|\alpha|\, \alpha$.

Definition 5. Define $\operatorname{sgnSqrCos} \alpha$ to be the signed square cosine $|\cos \alpha| \cos \alpha$.

Define $\operatorname{sgnSqrCos} \theta$ to be the signed square cosine of θ, $|\cos\theta| \cos\theta = \dfrac{|z|\, z}{r^2}$.

Definition 6. The **pseudospherical quartet** of a point in three-dimensional space with physics spherical coordinates (r, θ, ϕ) consists of the squared distance from the origin r^2, the zenith fraction $\operatorname{sgnSqrCos} \theta = \dfrac{\operatorname{sgnSqr} z}{r^2}$, and a pair $(\lambda \cos \phi, \lambda \sin \phi)$ where $\lambda \in \mathbb{R}^+$ is some arbitrary positive number, so that the pair represents some two-dimensional point with the same azimuth angle.

One obvious candidate for the pair $(\lambda \cos \phi, \lambda \sin \phi)$ when we are converting from Cartesian is the pair (x, y) of Cartesian coordinates; to normalize to find the actual cosine and sine from Cartesian coordinates

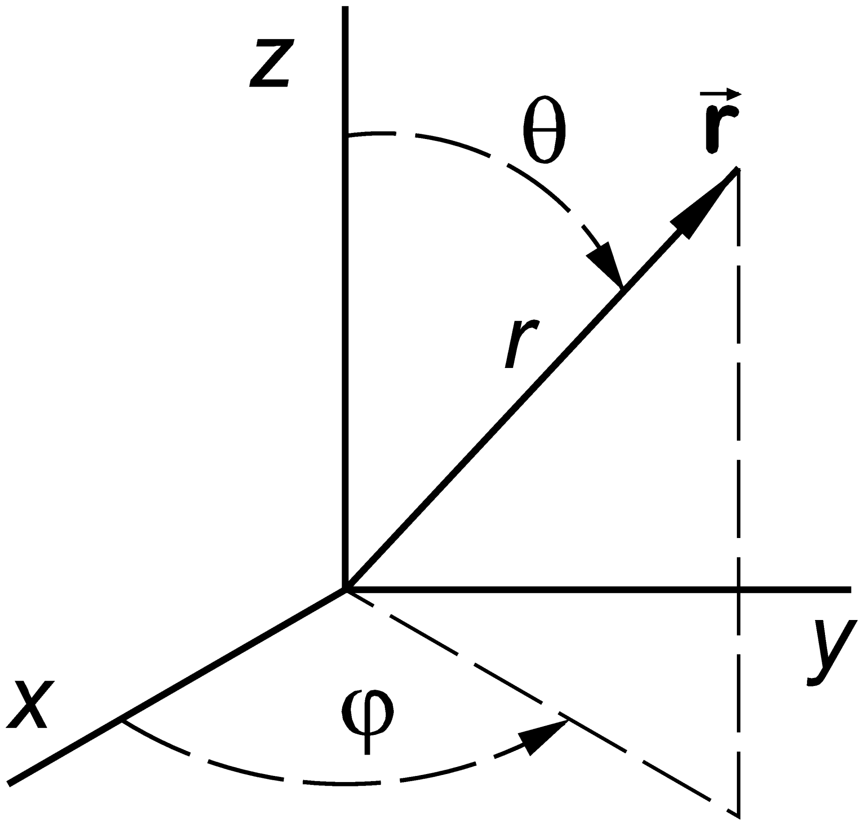


Figure 2.1: In physics, spherical coordinates consist of the distance r from the origin, the angle of declination θ from the zenith, and the azimuth angle ϕ.[Wor08]

$x = r\cos\phi\sin\theta$ and $y = r\sin\phi\sin\theta$, we would have to find the radius $r = \sqrt{x^2+y^2+z^2}$, but we can avoid that by leaving them un-normalized.

We use the square of the radius (which is naturally monotone in the non-negative radius) and the signed square $\operatorname{sgnSqrCos}\theta$ of the cosine of the zenith angle (which is monotone decreasing over the range $\theta \in [0,\pi]$). The sine-cosine pair for the azimuth angle obviously looks different from the other two; we'll discuss an alternative in Section 2.1. For now, we'll discuss how we can use the sine-cosine pair.

The standard procedure for intersecting a line with a cell is Liang-Barsky serial clipping [Har16], which finds the t-value at which the line intersects each plane defining the boundary of the cell. This works for Cartesian cells because finding the intersection between a line and a plane is fast; spherical cells, by contrast, have some boundaries that are not planar.

However, the azimuth-angle boundaries *are* planar. A ϕ-wedge of length $\leq \pi$ is the intersection of two half-spaces. A ϕ-wedge of length $\geq \pi$ is the union of two half-spaces. As such, we can calculate the intersection of a line with an azimuth-boundary just as we could for Cartesian facets: by intersecting the ray with the equation $\sin\phi x = \cos\phi y$, for which $\lambda\sin\phi x = \lambda\cos\phi y$ is just as good.

Strictly speaking, you could skip from here straight to the algorithm in Chapter 3. The remainder of Chapter 2 will be discussion of why we chose this coordinate system, complete with a few mathematical proofs of our claims.

2.1 Augmented tangent versus augmented sine versus sine-cosine pair

Given a spherical grid, with grid points in spherical coordinates, it is convenient to have coordinates that are monotone in corresponding spherical coordinates, so that we can quickly and easily translate a coordinate to the corresponding grid index without needing to search. We ultimately compromised on this point for implementation reasons, using the sine and cosine to represent the azimuth angle. However, we *can* have a monotone function of the azimuth angle as well.

Definition 7. Given an azimuth angle ϕ, the **augmented tangent** $\operatorname{mtan} : \left[-\frac{\pi}{2}, \frac{3}{2}\pi\right) \to \mathbb{F}_2 \times (\mathbb{R} \cup \{\infty\})$ is defined by $\operatorname{mtan}\phi := \left(\frac{\pi}{2} \leq \phi < \frac{3}{2}\pi, \tan\phi\right)$.

The canonical domain is chosen simply so that the function starts at $(0,\infty)$ and proceeds through the real line before reaching $(1,\infty)$ and proceeding through the real line again. We can then define a cyclic ordering $(0,\infty) < (0,x) < (1,\infty) < (1,x) < (0,\infty)$ and order within the real line normally. With this ordering, the augmented tangent is monotonically increasing over $\left[-\frac{\pi}{2}, \frac{3}{2}\pi\right)$.

Definition 8. The **pseudospherical triple** of a point in three-dimensional space with physics spherical coordinates (r,θ,ϕ) are the squared distance from the origin r^2, the zenith fraction $\operatorname{sgnSqrCos}\theta = \frac{\operatorname{sgnSqr} z}{r^2}$, and the augmented tangent $\operatorname{mtan}\phi$.

Our main task is to calculate a lot of intersection points between rays and facets. And there, the augmented tangent is cumbersome to deal with, and causes problems in floating-point calculations where it approaches infinity at $\frac{\pi}{2}$ and $\frac{3}{2}\pi$. So in practice we end up using the pair $(\cos\phi, \sin\phi)$ instead.

A hybrid approach is possible if we store both the augmented tangent and $(\cos\phi, \sin\phi)$ for all the grid points — we can use the augmented tangent to quickly find a range given the azimuth, but use the sine and cosine to calculate intersections with rays. This increases memory requirements, since we are storing three arrays instead of two.

There is another alternative as well. If we were to go to the trouble of separately tracking whether the azimuth angle is in the left half-space or right half-space, we could use $\sin\phi \in [-1, +1]$ instead of $\tan\phi \in (-\infty, +\infty)$ and still have something monotone in the azimuth angle, but floating-point arithmetic has problems with $\sin\phi$ when ϕ is very near $\pm\frac{\pi}{2}$, whereas $\tan\phi$ is usually easier to handle precisely because of its spreading effect.

Remember that the point of our alternate coordinate system is that we will use these functions to seek a cell in the grid. $\sin\phi$ is nearly flat near $\phi \approx \frac{\pi}{2}$, where $\sin\phi \approx 1$. As a result, a small error in $\sin\phi$ corresponds to a large difference in ϕ, meaning a small error in $\sin\phi$ could cause us to select a cell several cells away from the cell we want. Indeed, this is a chronic problem we need to be aware of with $\operatorname{sgnSqrCos}\theta$. By contrast, the slope of $\tan\phi$ is never less than 1, so a small error in $\tan\phi$ corresponds to a small error in ϕ.

sinpi As an aside, when we do deal with the angles directly, it should be noted that we can make our numerical work a little easier simply by storing θ and ϕ as multiples of π (when the breakpoints are rational fractions of π, which they usually are). IEEE 754-2008 defines sinPi(x) = sin(pi∗x) and cosPi = cos(pi∗x).

sinpi is implemented in C++ Accelerated Massive Parallelism library (`https://msdn.microsoft.com/en-us/library/hh290990.aspx`) and can be reached via CUDA (`https://github.com/Microsoft/clang/blob/master/lib/Headers/__clang_cuda_runtime_wrapper.h`).

2.2 Ditonicity

Definition 9. A **one-peak function** has exactly one local maximum. A **one-trough function** has exactly one local minimum. (Either of these is sometimes called unimodal, but at this point the word 'unimodal' has been attached to so many *subtly different* definitions that it seems best to avoid it.)

We call a function **ditone** if it is monotone or it has exactly one local maximum and no local minima or it has exactly one local minimum and no local maxima. The **unique extremum** or **hinge** of a ditone function is its unique extremal point. A monotone function, as a ditone function, may be referred to as 'hinged at infinity'.

The easiest way to think about ditonicity is that a ditone function is monotone twice, on either side of a maximum or minimum. We showed two example ditone functions in Fig. 1.3: the radius and the zenith angle. That's no coincidence.

On a line, r is ditone in t, hinged at the point where $\vec{p} + t\vec{d}$ is orthogonal to $\vec{d}$, as discussed in Lemma 8. What is less obvious is that the zenith angle is also ditone.

Lemma 1. $\operatorname{sgnSqr}\alpha$ *has continuous first derivative* $2|\alpha|$.

Proof. Everywhere except $\alpha = 0$, this is immediate. For $\alpha = 0$, $\lim_{h\searrow 0} \frac{\operatorname{sgnSqr}(0+h)}{h} = \lim_{h\searrow 0} \frac{|h|\,h}{h} = \lim_{h\searrow 0} |h| = 0$. □

Lemma 2. *On any line* $\vec{p} + t\vec{d}$*, the zenith angle* $\operatorname{sgnSqrCos}\theta$ *has derivative along the line* $\frac{\partial}{\partial t}\operatorname{sgnSqrCos}\theta = 2|z|\frac{\left\langle \vec{p}+t\vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle}{r^4}$*, continuous everywhere except (if the line contains the origin) the origin.*

Since $\operatorname{sgnSqrCos}\theta$ *is never noninstantaneously constant in* θ *by Lemma 3, this also tells us that the zenith angle is never noninstantaneously constant along the line unless* $\vec{p}$ *is parallel to* $\vec{d}$ *or the line is contained in the plane* $z = 0$*. (On any double-cone other than the degenerate double-cone that is the xy-plane, at any point in the double-cone, the only direction vector contained in the double-cone is the one on the line to the origin.)*

Proof. $\frac{\partial}{\partial t} r^2 = \frac{\partial}{\partial t}(x^2+y^2+z^2) = 2xd_x + 2yd_y + 2zd_z = 2\left\langle \vec{p}+t\vec{d} \,\middle|\, \vec{d}\right\rangle$; indeed, r^2 increases or decreases depending on whether we have yet passed the point on the line nearest the origin, which is the point orthogonal to $\vec{d}$.

$\frac{\partial}{\partial t}\operatorname{sgnSqr} z = \frac{\partial z}{\partial t}\frac{\partial \operatorname{sgnSqr} z}{\partial z}$. Of course, $\frac{\partial z}{\partial t} = \frac{\partial (p_z + td_z)}{\partial t} = d_z$. Meanwhile by Lemma 1, $\frac{\partial \operatorname{sgnSqr} z}{\partial z} = 2|z|$. Thus $\frac{\partial}{\partial t}\operatorname{sgnSqr} z = 2d_z|z|$.

Putting those together,

$$\begin{aligned}
\frac{\partial}{\partial t}\frac{\operatorname{sgnSqr} z}{r^2} &= \frac{r^2\frac{\partial}{\partial t}\operatorname{sgnSqr} z - \operatorname{sgnSqr} z \frac{\partial}{\partial t} r^2}{r^4} = \frac{r^2 2d_z|z| - z|z|\,2\left\langle \vec{p}+t\vec{d}\,\middle|\,\vec{d}\right\rangle}{r^4} \\
&= 2|z|\frac{r^2 d_z - z\left\langle \vec{p}+t\vec{d}\,\middle|\,\vec{d}\right\rangle}{r^4} \\
&= 2|z|\frac{d_z\left\langle \vec{p}+t\vec{d}\,\middle|\,\vec{p}+t\vec{d}\right\rangle - (p_z+td_z)\left\langle \vec{p}+t\vec{d}\,\middle|\,\vec{d}\right\rangle}{r^4} \\
&= 2|z|\frac{\left\langle \vec{p}+t\vec{d}\,\middle|\,d_z\vec{p}+td_z\vec{d} - (p_z+td_z)\vec{d}\right\rangle}{r^4} \\
&= 2|z|\frac{\left\langle \vec{p}+t\vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle}{r^4}
\end{aligned}$$

□

Lemma 3. $\frac{\partial}{\partial\theta}\operatorname{sgnSqrCos}\theta = -2\sin\theta|\cos\theta|$. *In particular,* $\operatorname{sgnSqrCos}\theta$ *is monotone decreasing in* θ *for* $\theta\in[0,\pi]$ *and is one-to-one (never noninstantaneously constant).*

Proof. $\frac{\partial}{\partial\theta}|\cos\theta|\cos\theta = \frac{\partial\cos\theta}{\partial\theta}\frac{\partial\operatorname{sgnSqr}(\cos\theta)}{\partial\cos\theta}$. Of course, $\frac{\partial\cos\theta}{\partial\theta} = -\sin\theta$. Meanwhile by Lemma 1, $\frac{\partial\operatorname{sgnSqr}(\cos\theta)}{\partial\cos\theta} = 2|\cos\theta|$. □

Note that $\left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle$ depends only on the line, id est is invariant to translation of $\vec{p}$ by multiples of $\vec{d}$: $\vec{p}^{(alt)} := \vec{p}+\vec{d}$ so that $\left\langle \vec{d}\,\middle|\,d_z\vec{p}^{(alt)} - p_z^{(alt)}\vec{d}\right\rangle = \left\langle \vec{d}\,\middle|\,d_z(\vec{p}+\vec{d}) - (p_z+d_z)\vec{d}\right\rangle = \left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle$.

Lemma 4. *If* $\left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle \neq 0$, *then* $(\operatorname{sign} z)\frac{z^2}{r^2} = \operatorname{sgnSqrCos}\theta$ *is ditone in t, hinged at:*

$$t = \frac{p_z\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle - d_z\langle\vec{p}\,|\,\vec{p}\rangle}{d_z\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle - p_z\left\langle \vec{d}\,\middle|\,\vec{d}\right\rangle} = \frac{\left\langle \vec{p}\,\middle|\,p_z\vec{d} - d_z\vec{p}\right\rangle}{\left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle} = -\frac{\left\langle \vec{p}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle}{\left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle}.$$

That is, we can tell whether we are in the increasing range or the decreasing range by examining the sign of the linear function $\left\langle \vec{p}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle + t\left\langle \vec{d}\,\middle|\,d_z\vec{p} - p_z\vec{d}\right\rangle$.

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$*, then* $(\operatorname{sign} z)\dfrac{z^2}{r^2} = \operatorname{sgnSqrCos}\theta$ *is monotone in* t*, increasing or decreasing depending on the sign of* $\left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$*. In this case, since the zenith angle is never noninstantaneously constant unless it is always constant, the range of the zenith angle must be open at both ends, never achieving its supremum nor infimum.*

Note that when $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$ *is* nearly *zero — if we rotate a line to* approach *the monotone case — the hinge approaches infinity.*

Furthermore, since by Lemma 3 $|\cos\theta|\cos\theta$ *is monotone decreasing in* θ *for* $\theta \in [0, \pi]$*, this also proves that* θ *is ditone in* t *with the same hinge.*

If the line does intersect the origin, then the zenith angle is constant except for a single jump discontinuity at the origin from $\dfrac{d_z^2}{\left|\vec{d}\right|^2}$ *to* $-\dfrac{d_z^2}{\left|\vec{d}\right|^2}$*.*

Proof. It suffices to show that the sign of $\dfrac{\partial}{\partial t}\operatorname{sgnSqrCos}\theta = \dfrac{\partial}{\partial t}\dfrac{z\sqrt{z^2}}{r^2}$ is always either zero or the same as the sign of $\left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle + t\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$.

By Lemma 2, $\dfrac{\partial}{\partial t}\operatorname{sgnSqrCos}\theta = 2\,|z|\,\dfrac{\left\langle \vec{p}+t\vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle}{r^4}$.

$2r^{-4}\sqrt{z^2}$ is always non-negative, so for any line that does not pass through the origin, the sign of $\dfrac{\partial}{\partial t}\operatorname{sgnSqrCos}\theta$ is the same as the sign of $\left\langle \vec{p}+t\vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = \left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle + t\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$, which is a linear function of t. □

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle > 0$, then $(\operatorname{sign} z)\dfrac{z^2}{r^2}$ starts out decreasing (negative derivative) and reverses at a minimum. If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle < 0$, then $(\operatorname{sign} z)\dfrac{z^2}{r^2}$ starts out increasing (positive derivative) and reverses at a maximum.

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$, then $(\operatorname{sign} z)\dfrac{z^2}{r^2}$ is either always increasing or always decreasing (approaching some limit), depending on the sign of $\left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$.

$\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$ does have a geometric meaning, but for the algorithm we don't need to care about that, since $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$ is the calculation that the computer will do.

Remark. For calculation purposes, we may note that

$$\begin{aligned} &\left\langle \vec{p}+t\vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle \\ &= d_z\,\|p\|^2 - p_z\,\langle p \,|\, d\rangle + t\,\langle d \,|\, p\rangle - tp_z\,\|d\|^2 \\ &= x * (d_z p_x - p_z d_x) + y * (d_z p_y - p_z d_y) \end{aligned}$$

However, since $\left|\vec{d}\right|^2$ and $\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle$ and $|\vec{p}|^2$ are useful for many things, we chose to store those and calculate the zenith hinge as:

```
double positionDirectionAndPrecomputed::thetaHinge() {
  return (posVec[2]*componentOfDirAwayFrom0
            - dirVec[2]*posSize2)
        / (dirVec[2]*componentOfDirAwayFrom0
            - posVec[2]*dirVecPrecomp.dirSize2);
```

}

Lemma 5. *If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle \neq 0$ (which by Lemma 4 is the condition for the zenith angle to have a hinge), then at the zenith-hinge, the extremum is* $\operatorname{sgnSqrCos}\theta = \operatorname{sign}\left(\left\langle \vec{d} \,\middle|\, p_z\vec{d} - d_z\vec{p} \right\rangle\right) \dfrac{\left\| p_z\vec{d} - d_z\vec{p} \right\|^2}{\|p\|^2 \|d\|^2 - \langle p \mid d\rangle^2}$.

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$ and $p_z = 0$, then either $d_z = 0$ or d is orthogonal to p, and in either case $\dfrac{\left\| p_z\vec{d} - d_z\vec{p} \right\|^2}{\|p\|^2 \|d\|^2 - \langle p \mid d\rangle^2} = \dfrac{|d_z|^2}{\|d\|^2}$ but of course $\operatorname{sign}\left(\left\langle \vec{d} \,\middle|\, p_z\vec{d} - d_z\vec{p} \right\rangle\right) = 0$*. Since the hinge can be taken to be either $\pm\infty$ in this case, we can simply take a floating-point signbit which will be arbitrary in the monotone case.*

Proof. Plugging

$$t = \frac{p_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - d_z \langle \vec{p} \mid \vec{p} \rangle}{d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - p_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle} = \frac{\left\langle \vec{p} \,\middle|\, p_z\vec{d} - d_z\vec{p} \right\rangle}{\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle} = -\frac{\left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle}{\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle}.$$

into $\dfrac{(p_z + td_z)^2}{\left\langle \vec{p} + \vec{d}t \,\middle|\, \vec{p} + \vec{d}t \right\rangle} = \dfrac{(p_z + td_z)^2}{\|p\|^2 + 2\langle p \mid d\rangle t + t^2 \|d\|^2}$,

The numerator becomes

$$\begin{aligned}
&\left(d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - p_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle\right)^2 (p_z + td_z)^2 \\
&= p_z^2 \left(d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - p_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle\right)^2 + 2p_z d_z \left(d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - p_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle\right)\left(p_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - d_z \langle \vec{p} \mid \vec{p} \rangle\right) \\
&\quad + d_z^2 \left(p_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle - d_z \langle \vec{p} \mid \vec{p} \rangle\right)^2 \\
&= p_z^2 \left(d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 2p_z d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle + p_z^2 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle^2\right) \\
&\quad + 2p_z d_z \left(p_z d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle - p_z^2 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle + p_z d_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle\right) \\
&\quad + d_z^2 \left(p_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 2p_z d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle + d_z^2 \langle \vec{p} \mid \vec{p} \rangle^2\right) \\
&= p_z^2 d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 2p_z^3 d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle + p_z^4 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle^2 \\
&\quad + 2p_z^2 d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 2p_z d_z^3 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle - 2p_z^3 d_z \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle + 2p_z^2 d_z^2 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle \\
&\quad + p_z^2 d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 2p_z d_z^3 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle + d_z^4 \langle \vec{p} \mid \vec{p} \rangle^2 \\
&= 4p_z^2 d_z^2 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - 4p_z^3 d_z \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle - 4p_z d_z^3 \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle \\
&\quad + p_z^4 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle^2 + 2p_z^2 d_z^2 \left\langle \vec{d} \,\middle|\, \vec{d} \right\rangle \langle \vec{p} \mid \vec{p} \rangle + d_z^4 \langle \vec{p} \mid \vec{p} \rangle^2 \\
&= 4 \left\langle d_z\vec{p} \,\middle|\, p_z\vec{d} \right\rangle^2 - 4 \left\langle d_z\vec{p} \,\middle|\, p_z\vec{d} \right\rangle \left\langle p_z\vec{d} \,\middle|\, p_z\vec{d} \right\rangle - 4 \left\langle d_z\vec{p} \,\middle|\, p_z\vec{d} \right\rangle \langle d_z\vec{p} \mid d_z\vec{p} \rangle \\
&\quad + \left\langle p_z\vec{d} \,\middle|\, p_z\vec{d} \right\rangle^2 + 2 \left\langle p_z\vec{d} \,\middle|\, p_z\vec{d} \right\rangle \langle d_z\vec{p} \mid d_z\vec{p} \rangle + \langle d_z\vec{p} \mid d_z\vec{p} \rangle^2 \\
&= \left(\left\langle p_z\vec{d} \,\middle|\, p_z\vec{d} \right\rangle - 2 \left\langle d_z\vec{p} \,\middle|\, p_z\vec{d} \right\rangle + \langle d_z\vec{p} \mid d_z\vec{p} \rangle\right)^2 \\
&= \left\langle p_z\vec{d} - d_z\vec{p} \,\middle|\, p_z\vec{d} - d_z\vec{p} \right\rangle^2
\end{aligned}$$

while the denominator becomes

$$\begin{aligned}
&\left(d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle - p_z \left\langle \vec{d} \middle| \vec{d} \right\rangle\right)^2 \langle \vec{p} | \vec{p} \rangle \\
&\quad + 2 \left\langle \vec{p} \middle| \vec{d} \right\rangle \left(d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle - p_z \left\langle \vec{d} \middle| \vec{d} \right\rangle\right) \left(p_z \left\langle \vec{p} \middle| \vec{d} \right\rangle - d_z \langle \vec{p} | \vec{p} \rangle\right) \\
&\quad + \left(p_z \left\langle \vec{p} \middle| \vec{d} \right\rangle - d_z \langle \vec{p} | \vec{p} \rangle\right)^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle \\
&= \langle \vec{p} | \vec{p} \rangle \left(d_z^2 \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle \left\langle \vec{d} \middle| \vec{d} \right\rangle + p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle^2\right) \\
&\quad + 2 \left\langle \vec{p} \middle| \vec{d} \right\rangle \left(p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 - d_z^2 \left\langle \vec{p} \middle| \vec{d} \right\rangle \langle \vec{p} | \vec{p} \rangle - p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle \left\langle \vec{p} \middle| \vec{d} \right\rangle + p_z d_z \left\langle \vec{d} \middle| \vec{d} \right\rangle \langle \vec{p} | \vec{p} \rangle\right) \\
&\quad + \left\langle \vec{d} \middle| \vec{d} \right\rangle \left(p_z^2 \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle \langle \vec{p} | \vec{p} \rangle + d_z^2 \langle \vec{p} | \vec{p} \rangle^2\right) \\
&= -d_z^2 \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 \langle \vec{p} | \vec{p} \rangle - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle \langle \vec{p} | \vec{p} \rangle \left\langle \vec{d} \middle| \vec{d} \right\rangle + p_z^2 \langle \vec{p} | \vec{p} \rangle \left\langle \vec{d} \middle| \vec{d} \right\rangle^2 \\
&\quad + 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle^3 - p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 + d_z^2 \langle \vec{p} | \vec{p} \rangle^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle \\
&= \|\vec{p}\|^2 \left\|\vec{d}\right\|^2 \left(p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle + d_z^2 \langle \vec{p} | \vec{p} \rangle\right) \\
&\quad - \left\langle \vec{p} \middle| \vec{d} \right\rangle^2 \left(p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle + d_z^2 \langle \vec{p} | \vec{p} \rangle\right) \\
&= \left(\|\vec{p}\|^2 \left\|\vec{d}\right\|^2 - \left\langle \vec{p} \middle| \vec{d} \right\rangle^2\right) \left(p_z^2 \left\langle \vec{d} \middle| \vec{d} \right\rangle - 2p_z d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle + d_z^2 \langle \vec{p} | \vec{p} \rangle\right) \\
&= \left(\|\vec{p}\|^2 \left\|\vec{d}\right\|^2 - \left\langle \vec{p} \middle| \vec{d} \right\rangle^2\right) \left(\left\langle p_z\vec{d} \middle| p_z\vec{d} \right\rangle - 2 \left\langle d_z\vec{p} \middle| p_z\vec{d} \right\rangle + \langle d_z\vec{p} | d_z\vec{p} \rangle\right) \\
&= \left(\|\vec{p}\|^2 \left\|\vec{d}\right\|^2 - \left\langle \vec{p} \middle| \vec{d} \right\rangle^2\right) \left\langle p_z\vec{d} - d_z\vec{p} \middle| p_z\vec{d} - d_z\vec{p} \right\rangle \\
&= \left(\|\vec{p}\|^2 \left\|\vec{d}\right\|^2 - \left\langle \vec{p} \middle| \vec{d} \right\rangle^2\right) \left\|p_z\vec{d} - d_z\vec{p}\right\|_2^2
\end{aligned}$$

The value of z at the zenith-hinge is $p_z - \frac{\left\langle \vec{p} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle}{\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle} d_z = \frac{\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle p_z - \left\langle \vec{p} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle d_z}{\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle} = -\frac{\left\langle d_z\vec{p} - p_z\vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle}{\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle} = -\frac{\left\|d_z\vec{p} - p_z\vec{d}\right\|^2}{\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle}$. $\left\|d_z\vec{p} - p_z\vec{d}\right\|^2$ must be nonnegative, so the sign of z at the zenith-hinge the same as the sign of $-\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle = \left\langle \vec{d} \middle| p_z\vec{d} - d_z\vec{p} \right\rangle = p_z \left\|\vec{d}\right\|^2 - d_z \left\langle \vec{p} \middle| \vec{d} \right\rangle$.

If $\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle = 0$, we can construct a sequence of lines with $\left\langle \vec{d}^{(i)} \middle| d_z^{(i)}\vec{p}^{(i)} - p_z^{(i)}\vec{d}^{(i)} \right\rangle \neq 0$ that converge to the actual line.

□

As an example, for $\vec{d} = \begin{bmatrix}1\\0\\1\end{bmatrix}$ and $\vec{p} = \begin{bmatrix}0\\1\\0\end{bmatrix}$, $d_z\vec{p} - p_z\vec{d} = \vec{p} = \begin{bmatrix}0\\1\\0\end{bmatrix}$, which is orthogonal to $\vec{d}$ (so the zenith angle is monotone by Lemma 4) but not to $\vec{p}$.

If $\left\langle \vec{d} \middle| d_z\vec{p} - p_z\vec{d} \right\rangle = 0$ (so that the zenith angle is monotone along the line by Lemma 4), then we can define the extremum to equal the zenith angle of $\pm\vec{d}$, $\pm\frac{d_z^2}{\|d\|^2}$, and we would like if the calculated extremum came out the same in that case. However, this is not yet proven. We do have a couple of suggestive observations:

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$, then the numerator $\left\langle d_z\vec{p} - p_z\vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = \left\langle d_z\vec{p} - p_z\vec{d} \,\middle|\, d_z\vec{p} \right\rangle = d_z^2 \|p\|^2 - p_z d_z \langle d \,|\, p\rangle = d_z(d_z p_x^2 + d_z p_y^2 - p_z p_x d_x - p_z p_y d_y)$.

If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$, then the denominator

$$\begin{aligned} \|p\|^2 \|d\|^2 - \langle p \,|\, d\rangle^2 &= \left\langle p \,\middle|\, \|d\|^2 p - \langle p \,|\, d\rangle\, d \right\rangle = \left\langle \|p\|^2 d - \langle p \,|\, d\rangle\, p \,\middle|\, d \right\rangle \\ &= \left\langle (p_x^2 + p_y^2)d - (p_x d_x + p_y d_y)p + p_z(p_z d - d_z p) \,\middle|\, d \right\rangle = \left\langle (p_x^2 + p_y^2)d - (p_x d_x + p_y d_y)p \,\middle|\, d \right\rangle \end{aligned}$$

As we rotate a line so that we approach $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$, if $\left\langle \vec{p} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle \neq 0$, then the t-value of the hinge goes to $\frac{1}{0} = \pm\infty$, and thus the computed zenith angle extremum approaches $\pm\frac{d_z^2}{\left\|\vec{d}\right\|^2}$. We simply calculate the hinge, and if it turns out that the hinge is at infinity, the entire line is on one side of the hinge, and that's perfectly fine.

But even if the computing architecture supports floating-point infinity, $\frac{(p_z + \infty d_z)}{\|p + \infty d\|^2}$ will evaluate to `NaN` rather than $\pm\frac{d_z^2}{\left\|\vec{d}\right\|^2}$. And $\frac{(p_z/t + d_z)}{\|p/t + \infty d\|^2}$ is more expensive to calculate, so we don't want to do that all the time just for one special case. We would prefer to do without a special case for when $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$ and the zenith angle is monotone (especially because with floating-point numbers it would be impossible to be sure of when $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle$ was exactly zero).

$\begin{bmatrix} t \\ 1 \\ t \end{bmatrix}$ has zenith angle $\pm\frac{t^2}{2t^2+1}$, which approaches $\pm\frac{1}{2}$ but never achieves it (and certainly never achieves anything larger). For $\vec{p} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ and $\vec{d} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$, $(\|d\|^2 \cos^2\theta - d_z^2)t^2 + 2(\langle p \,|\, d\rangle \cos^2\theta - p_z d_z)t + \|p\|^2 \cos^2\theta - p_z^2 = 0$ becomes $(2\cos^2\theta - 1)t^2 + \cos^2\theta = 0$ becomes $t^2 = \frac{-\cos^2\theta}{2\cos^2\theta - 1}$. From the other direction, $\cos^2\theta = \frac{t^2}{2t^2+1}$ becomes $(2t^2+1)\cos^2\theta = t^2$ becomes $(2\cos^2\theta - 1)t^2 = -\cos^2\theta$. $\frac{\left\|p_z\vec{d} - d_z\vec{p}\right\|^2}{\|p\|^2 \|d\|^2 - \langle p \,|\, d\rangle^2} = \frac{\|-\vec{p}\|^2}{2-0} = \frac{1}{2}$. For $\cos^2\theta \geq \frac{1}{2}$, this can never be achieved, because the signs don't match.

2.3 Conclusion

The mathematical properties of this coordinate system look useful, but we still need to use them. Now that we can view a spherical domain as represented by these coordinates, Chapter 3 will bring this all together (along with a few details which we won't actually prove until Chapter 4) into a concrete method for marching rays.

Chapter 3

Algorithm

Before we describe the algorithm, we must be concrete about what we expect the algorithm to produce. The algorithm takes a set of parallel rays (or any rays, really, but if they aren't parallel we don't have special techniques to save computation, so in those cases we might as well take rays one at a time) and outputs, for each ray, a list of cells in the spherical grid and the points at which the ray enters and exits each cell.

We return the entry and exit points, rather than the lengths alone, because some interpolation techniques rely on this information. This does not increase internode communication, because the entry and exit points can be described concisely by the t-values, with $\vec{p}$ and $\vec{d}$ implicit. And of course the exit point of one cell is the entry point of the next cell, except at the edge of the domain. A given ray can cross the edge of the domain at most four times, so this is only a question of transmitting $n+4$ numbers rather than n numbers in the worst case.

There are two flavors of the algorithm: a separable version, which takes memory linear in the number of cells a ray might intersect but which is much cleaner to implement, and a serial version, which takes constant memory (so might be suitable for GPUs) but is much more fiddly. The serial version can also be accelerated by using approximate t-values, saving `FSQRT`s at the cost of introducing some error. (Of course, either version can save time at the cost of introducing some error by simply dropping some of the resolution of the dataset, as discussed in Section 6.3.)

We will present the simple version first, as this is what we recommend using (if you do not use the pre-implemented version to be integrated into `yt`).

3.1 Algorithm quick-start

We separately find the entry and exit points for the r-annuli, θ-shells, and ϕ-wedges (storing them in $O(n)$ memory) and then interleave them (in $O(n)$ time) to complete the list of entry and exit points for all cells.

It is easier to create a picture for the algorithm (Fig. 3.1 and Fig. 3.2) if we define some objects:

- The volume container knows about the domain. The volume container is a data structure that can tell us how many cells there are and the bounds of each cell (pre-converted to our alternate coordinate system). The volume container can also, if necessary, take the coordinates of a point and turn that into a cell address — for non-regular grids this might not take constant time, so we can't do this too often. The volume container is agnostic to what is being cast through the domain (though some of the functions only apply to ditone parameters).

- The line object knows about the line. The line object can convert between t-values and the various coordinates. The line object is agnostic to the domain.

$$\begin{tikzcd}
\text{ray-caster} \arrow[d, "0:\ t \text{ range}"] & \\
\text{line} \arrow[r, bend left, "1:\ \phi \text{ range (monotone)}"] \arrow[d, "3:\ t\text{-values of boundary-crossings}"] & \text{volume container} \arrow[l, "2:\ \phi \text{ boundary values}"] \\
\text{AzimuthRangeTs} &
\end{tikzcd}$$

Figure 3.1: Since the azimuth angle is monotone, the given `tStart` and `tStop` lets the line tell us the range ϕ will cover. The volume container simply fills in the intermediate azimuth-boundaries. The line then calculates its intersection with each of those azimuth-values.

$$\begin{tikzcd}
\text{ray-caster} \arrow[d, "0:\ t \text{ range}"] & \\
\text{line} \arrow[r, bend left, "1:\ \text{value at start, stop, and hinge if applicable}"] \arrow[d, "3:\ t\text{-values of boundary-crossings}"] & \text{volume container} \arrow[l, "2:\ \text{boundary values}"] \\
\text{DitoneRangeTs} &
\end{tikzcd}$$

Figure 3.2: The ditone parameters are only a little more complicated. The given `tStart` and `tStop` lets the line tell us the range the coordinate will cover, as well as whether the hinge falls within the given line segment. The volume container fills in the boundaries between the start and the hinge and between the hinge and the exit. The line then calculates its intersection with each of those boundary-values.

3.1.1 Finding each series of indices

We can use monotonicity and ditonicity to fully list out the indices of the coordinate-ranges that the ray passes through, before calculating the t-values (the exact entry and exit points).

For each coordinate, we first ask the line what the value is of the coordinate at `tStart` and `tStop` (and at the hinge, if ditone). We then fill in the full list of indices (this requires a memory allocation) from those fulcra. For the monotone coordinate ϕ, the ray must pass through all intermediate ϕ-wedges; for the ditone coordinates, the ray must proceed from the starting value of the ditone coordinate to the hinge and then reverse to the ending value of the ditone coordinate. (If the hinge happens to fall outside the domain, then the ditone coordinate is treated as monotone.)

The hinge of the zenith angle is given by Lemma 4. The hinge of the radius is given by Lemma 8.

Given the list of indices, we ask the volume container for the values of the coordinate at those boundary-crossings (this requires a memory allocation). We will save these values, or normalized versions, to pass to the sampler function later.

3.1.2 Calculating entry and exit points

Given the list of values for the coordinate, we ask the line for the t-value for each boundary-crossing. It's actually a little more complicated since we use ditonicity to sometimes save `FSQRT`s — we can sometimes get two t-values for the price of one (see Section 4.3 and Section 4.4). Each r^2-value has two t-values, and some $(\operatorname{sign}\cos\theta)\cos^2\theta$-values have two t-values. We can get both for the price of one (plus one `FADD`). Due to saving `FSQRT`s this way, the simpler version of the algorithm may in some cases be faster than the constant-memory serial version.

augmented tangent versus sine-cosine pair Finding the t-value at which the ray intersects a plane requires only a division.

Finding an intersection with a plane has the same problem as using the unaugmented tangent: we can't tell which side it's on, so we'd have to classify that in advance before finding t.

The ray intersects the plane when $0 = (p_x + d_x t)\sin\phi_2 - (p_y + d_y t)\cos\phi_2 = p_x \sin\phi_2 - p_y\cos\phi_2 + t(d_x\sin\phi_2 - d_y\cos\phi_2)$ id est $t = -\dfrac{p_x\sin\phi_2 - p_y\cos\phi_2}{d_x\sin\phi_2 - d_y\cos\phi_2}$. (On the 'opposite side', sin and cos both reverse signs so this ratio comes out the same.) (This matches the version with tangent but avoids the issue when tangent is ∞.) We now have an issue if $d_x\sin\phi_2 = d_y\cos\phi_2$, but that will happen iff $\vec{d}$ is parallel to the plane, which can't be blamed on the specific procedure.

With the augmented tangent, we can plug t back in to check whether $p_x + d_x t > 0$, but it's much simpler to have a single procedure that will also work for a vertical line instead of specifically checking for that and checking $y > 0$.

3.1.3 Finding cell-indices of points

Since r^2 is monotone in r and sgnSqrCos θ is monotone in θ, we can find those indices by binary search. In practice, we pre-calculate the pseudospherical coordinates for the vertices of the spherical grid, and use those instead of the true spherical coordinates we were given. Given a coordinate value r^2 equal to, say, 7, and breakpoints $[1, 4, 9, 16]$, simple binary search reveals the interval where 7 fits: $4 < 7 < 9$.

The azimuth is a little more complicated to deal with, simply because determining whether a point is inside an azimuth-wedge is easy to get wrong. In practice, we implement using the sine and cosine to find whether a point is in each half-space, then taking the union or intersection of half-spaces :

```
bool phi1_half_space_contains_xy(double x, double y) {
  return cosphi1*y >= sinphi1*x;
}
bool phi2_half_space_contains_xy(double x, double y) {
  return cosphi2*y <= sinphi2*x;
}
bool contains_xy(double x, double y) {
  bool intersectHalfSpaces = range_at_most_pi();
  // if range is >pi, union of half-spaces;
  // if range is <pi, intersection of half-spaces
  // If full circle (cosphi1 == cosphi2 and sinphi1 == sinphi2),
  // then intersectHalfSpaces is false
  // and one of the half_space functions is always true
  return logical_xor(intersectHalfSpaces,
                     logical_xor(phi1_half_space_contains_xy(x,y),
                                 intersectHalfSpaces) or
                     logical_xor(phi2_half_space_contains_xy(x,y),
                                 intersectHalfSpaces) );
}
```

The cost of binary search doesn't end up being significant, because we only need to do this at the beginning and end of each ray, as well as at the hinges of the ditone coordinates. The fact that the pseudospherical coordinates are monotone and ditone *along the ray* means that we can fill in the other indices without even calculating any coordinates. So we don't actually need the fact that the pseudospherical coordinates are each monotone in their respective base spherical coordinates; we could laboriously check every single coordinate-range and it still wouldn't end up being a significant cost. However, we *do* need the fact that our coordinates correspond individually to each of the three true spherical coordinates, because that's what allows us to find a cell as three separate indices. Otherwise we'd have to check every individual cell, taking time $O(n^3)$ rather than $3n$. And any continuous one-to-one function on a single coordinate must be monotone anyway.

3.1.4 Interleaving t-values

Given the three lists of t-values, we merge them. This is like sorting but takes linear time instead of $\Theta(n \log n)$ because we take advantage of the fact we know we have three sorted lists.

To interleave, we walk along the ray through the transition points in order of t.

```
function CELLRANGETS(radiusTs, zenithTs, azimuthTs)
    while at least one transition remains in radiusTs or zenithTs or azimuthTs do
        nextT ← $\min\{t \in \texttt{radiusTs.tExit} \cup \texttt{zenithTs.tExit} \cup \texttt{azimuthTs.tExit}\}$ ▷ exit the cell on the facet corresponding to the coordinate selected
            set the new index of whichever coordinate had the min tExit
            add the new cell index-triple to the list
        delete the minimum entry from radiusTs $\cup$ zenithTs $\cup$ azimuthTs
    end while
end function
```

Interleaving t-values from the three arrays takes at most $2n$ comparisons, not $\Theta(n \log n)$ as it would if we were truly sorting, because each of the three arrays is already individually sorted.

Note that if a ray passes through a corner of a cell, we list a zero-length intersection with an (arbitrarily chosen) in-between cell so that only one coordinate index changes at a time. This is for ease of testing and has no physical meaning.

It is possible to compress the list of cell-intersections into just (# cells intersect +1) structs of which each is a floating-point t-value, an enum for which coordinate to transition, and a boolean for which direction to transition. We have not done this because having the list available in uncompressed form makes it easier to verify that the contents are correct.

3.1.5 Overall pseudocode

```
function LINE.COORDINATE_FROM_T(t, coordEnum)
    $r^2 = |\vec{p}|^2 + t\left((2\left\langle\vec{p}\,\middle|\,\vec{d}\right\rangle + t\left|\vec{d}\right|\right)$
    if coordEnum == radius$^2$ then
        return $r^2$
    else if coordEnum == zenith then
        return $(p_z + td_z)\,|p_z + td_z|\,/r^2$
    else if coordEnum == azimuth then
        return $(p_x + td_x, p_y + td_y)$
    end if
end function
function LINE.T_FROM_COORDINATE(value, coordEnum)
                    ▷ Implementation-dependent details are elided where values become infinite.
    if coordEnum == radius$^2$ then
        $r^2$ = value
        return tNearest0 $\pm \sqrt{\texttt{tNearest0}^2 - (r^2 - |\vec{p}|^2)/\left|\vec{d}\right|^2}$    ▷ tNearest0$^2$ is precomputed and cached,
since it does not depend on $r$.
    else if coordEnum == zenith then
        cos2 = value
        Solve quadratic equation for $t$ in terms of cos$^2$, then accept or reject solutions based on sign.
        numerator $\leftarrow \sqrt{\cos^2\theta\left(p_z^2\left|\vec{d}\right|^2 - 2p_z d_z\left\langle\vec{p}\,\middle|\,\vec{d}\right\rangle + d_z^2|\vec{p}|^2 + \cos^2\theta\left(\left\langle\vec{p}\,\middle|\,\vec{d}\right\rangle^2 - \left|\vec{d}\right|^2|\vec{p}|^2\right)\right)}$
        $t \leftarrow$ numerator/ $\left(\left|\vec{d}\right|^2\cos^2\theta - d_z^2\right)$
    else if coordEnum == azimuth then
        $(\cos\phi, \sin\phi)$ = value
        return $(p_x\sin\phi - p_y\cos\phi)/(d_y\cos\phi - d_x\sin\phi)$    ▷ $1/(d_y\cos\phi - d_x\sin\phi)$ if worth precomputing
and caching if we are casting several parallel rays through the same azimuth-domain.
    end if
end function
```

```
function VOLUME CONTAINER.INDEX_OF_COORDINATE_VALUE(value)
    return index of range that contains value
end function
function FILL_IN_INDICES(start/stop/hinge indices)
    Fill in the gaps, so that each transition is only to the next index (1 up or 1 down)
end function
function VOLUME CONTAINER.COORDINATE_AT_BOUNDARY(index)
    return The value from the array of coordinate-boundaries.
end function
for each coordinate do
    Ask the line the value of the coordinate at tStart and tStop (and hinge, if ditone)
    Ask the volume container the indices of those coordinate values
    Fill in the full list of indices from those fulcra
    Ask the volume container the coordinate values of all boundary-crossings
    Ask the line the t-values for those coordinate values
end for
Call function cellRangeTs (Section 3.1.4).
```

3.2 Algorithm justification

Here we provide the details we elided above.

3.2.1 Determine whether azimuth angle is increasing

It is worth recalling a basic fact about fractions:

Remark. If $\frac{a}{b} < \frac{c}{d}$, then $\frac{a}{b} < \frac{a+c}{b+d} < \frac{c}{d}$.

Proof. Since we can assign the sign to either numerator or denominator, we can arbitrarily declare that $b > 0$ and $d > 0$. Then $ad < bc$. Since $ab + ad < ab + bc$ and $b + d > 0$, $a < \frac{(a+c)b}{b+d}$, so since $b > 0$, $\frac{a}{b} < \frac{a+c}{b+d}$. Since $ad + cd < bc + cd$ and $b + d > 0$, $\frac{(a+c)d}{b+d} < c$, so since $d > 0$, $\frac{a+c}{b+d} < \frac{c}{d}$. ☐

We should take a moment to clarify what we do and do not mean by 'increasing'.

Consider $\vec{p} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$, $\vec{d} = \begin{bmatrix} -2 \\ 0 \\ 0 \end{bmatrix}$, so $\vec{p} + \vec{d} = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$. Then $d_y p_x = 0$ and $d_x p_y = -2$. The final tangent is less — in fact, *negative* — but the tangent has been increasing everywhere except at the discontinuity, and certainly ϕ is increasing.

Consider $\vec{p} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$, $\vec{d} = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix}$, so $\vec{p} + \vec{d} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$. ϕ may wrap around from 2π to 0, but ϕ is still 'increasing' for our purposes; we can say mathematically that we proceed to $\frac{9}{4}\pi$, even if we don't represent that in the computer.

Conceptually, we determine whether ϕ is increasing along a ray by comparing $\frac{d_y}{d_x}$ to $\frac{p_y}{p_x}$ — if $\frac{d_y}{d_x} < \frac{p_y}{p_x}$, then $\frac{p_y + d_y}{p_x + d_x} < \frac{p_y}{p_x}$ — but computationally this works even if $d_x = 0$ or $p_x = 0$.

Lemma 6.

- *If $d_y p_x < d_x p_y$, then ϕ is strictly decreasing in t (subject to ϕ's wraparound), and* $\tan\phi$ *is decreasing in t except at $x = 0$, where* $\tan\phi$ *jumps from $-\infty$ to $+\infty$.*
- *If $d_y p_x > d_x p_y$, then ϕ is strictly increasing in t (subject to ϕ's wraparound), and* $\tan\phi$ *is increasing in t except at $x = 0$, where* $\tan\phi$ *jumps from $+\infty$ to $-\infty$.*
- *If $d_y p_x = d_x p_y$, then ϕ is constant along the ray.*

Proof. $\tan\phi$ is increasing in ϕ everywhere it is differentiable, and

$$\frac{\partial}{\partial t}\tan\phi = \frac{\partial}{\partial t}\frac{p_y + d_y t}{p_x + d_x t} = \frac{d_y(p_x + d_x t) - d_x(p_y + d_y t)}{(p_x + d_x t)^2} = \frac{d_y p_x - d_x p_y}{(p_x + d_x t)^2}$$

has the same sign as $d_y p_x - d_x p_y$, ignoring the fact that it goes to infinity at $x = 0$. So ϕ and $\tan\phi$ are always either increasing or decreasing according to $d_y p_x - d_x p_y$, except for discontinuities at $x = 0$ where $\tan\phi$ jumps between $\pm\infty$. □

```
np.float64_t delta_phi_sign(np.float64_t v_pos[3], np.float64_t v_dir[3]):
  # Return +1 if phi is increasing,
  # -1 if phi is decreasing,
  # 0 if phi is constant.
  return sign(v_dir[1]*v_pos[0] - v_dir[0]*v_pos[1])
```

3.2.2 Closest approach to origin

In Section 2.2, we handwaved off the radius, since it's obvious that it *is* ditone and that's all we needed to know at the time. But to actually fill in the indices on either side of the hinge, we need to find the hinge.

```
positionDirectionAndPrecomputed(double v_pos[3], double v_dir[3],
                                precomputedForDirectionVec predone)
                                : dirVecPrecomp(predone) {
  for(unsigned int i = 0; i < 3; i++) {
    dirVec[i] = v_dir[i];
    posVec[i] = v_pos[i];
  }
  componentOfDirAwayFrom0 = < v_pos | v_dir >;
  posSize2 = v_pos[0]*v_pos[0] + v_pos[1]*v_pos[1] + v_pos[2]*v_pos[2];
}
double positionDirectionAndPrecomputed::tNearest0() {
  return -componentOfDirAwayFrom0*dirVecPrecomp.recipSqr2norm;
}
```

Lemma 7. *Along a line $\vec{p}+t\vec{d}$, the rate of change of the square of the radius is* $\frac{\partial}{\partial t}r^2 = 2\left\|\vec{d}\right\|^2 t + 2\left\langle \vec{d}\,\middle|\,\vec{p}\right\rangle = 2\left\langle \vec{d}\,\middle|\,\vec{p}+\vec{d}t\right\rangle$.

Proof. $\frac{\partial}{\partial t}\left\langle \vec{p}+\vec{d}t\,\middle|\,\vec{p}+\vec{d}t\right\rangle = \frac{\partial}{\partial t}\left(\|p\|^2 + 2t\langle p\,|\,d\rangle + t^2\|d\|^2\right) = 2\langle p\,|\,d\rangle + 2t\|d\|^2$. □

Lemma 8. *The closest approach of a line $\vec{p}+t\vec{d}$ to the origin occurs when $\vec{p}+t\vec{d}$ is orthogonal to $\vec{d}$, that is, when* $t = -\frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle}{\left\|\vec{d}\right\|_2^2}$, *at which point* $r^2 = \|\vec{p}\|^2 - \frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle^2}{\left\|\vec{d}\right\|_2^2} = \frac{\|\vec{p}\|_2^2\left\|\vec{d}\right\|_2^2 - \left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle^2}{\left\|\vec{d}\right\|_2^2}$.

Proof. We can prove this using either geometry (anywhere $\vec{d}$ is *not* orthogonal to $\vec{p}+t\vec{d}$, the line is not tangent to the sphere of radius $\left\|\vec{p}+\vec{d}t\right\|$, so the size of the vector must increase in one direction and decrease in the other direction) or algebra.

$\|r\| = \sqrt{(p_x+td_x)^2+(p_y+td_y)^2+(p_z+td_z)^2}$ is minimized at the same t as $r^2 = (p_x+td_x)^2+(p_y+td_y)^2+(p_z+td_z)^2$, so it suffices to show that r^2 is minimized at $t = -\frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle}{\left\|\vec{d}\right\|_2^2}$.

By Lemma 7, $\frac{\partial}{\partial t}r^2 = 0$ if and only if $t = -\frac{\left\langle \vec{d}\,\middle|\,\vec{p}\right\rangle}{\left\|\vec{d}\right\|^2}$.

There $r^2 = \langle p+td\,|\,p+td\rangle = t^2\|d\|^2 + 2t\langle p\,|\,d\rangle + \|p\|^2 = \frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle^2}{\left\|\vec{d}\right\|_2^4}\|d\|^2 - 2\frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle}{\left\|\vec{d}\right\|_2^2}\langle p\,|\,d\rangle + \|p\|^2 = \frac{\langle p\,|\,d\rangle^2}{\|d\|^2} - 2\frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle^2}{\left\|\vec{d}\right\|_2^2} + \|p\|^2 = \|p\|^2 - \frac{\left\langle \vec{p}\,\middle|\,\vec{d}\right\rangle^2}{\left\|\vec{d}\right\|_2^2}$. □

3.3 Cylindrical coordinates

Adapting the algorithm to cylindrical coordinates is simplicity itself, though taking advantage of the potential for cylindrical coordinates to be processed faster than spherical coordinates would take a bit of work. On the simplest level, instead of calculating the list of t-values for the boundary-crossings of the zenith angle, calculate them for the z-coordinate. If the breakpoints of the z-coordinate are evenly spaced, then the t-values will also be evenly spaced (a property that is not true for the zenith angle in spherical coordinates), so in that case we don't even need to explicitly store all of the t-values; the first, last and step-length Δ_z/d_z suffice, so this can be done in constant time and constant memory space.

3.4 Two-dimensional polar

The two-dimensional polar case is even simpler than the cylindrical case: the algorithm works unaltered if all points happen to have $z = 0$ and hence $\theta = \frac{\pi}{2}$.

3.5 Axially symmetric data

To adapt the algorithm to an axially symmetric dataset, simply skip the step of finding the azimuth-breakpoints, because there will be none. In our implementation (and probably most implementations), this can be done immediately by listing a single azimuth 'wedge', the full range 0 to 2π (or less, if the domain is not truly axially symmetric and the region under consideration is only a fraction of the full circle).

If the data is not truly axially symmetric, but rather has a perturbation function such as AxiSEM uses, then this still works, *but* a custom sampler function that will be required: just as a standard sampler function might assume the field data is linear in the azimuth angle and interpolate according to that assumption, you will need a sampler function that interpolates according to the function you've decided to assume.

3.6 Why we use this hierarchy of loops

Our algorithm takes each ray in parallel and determines which cells that ray intersects. A ray marching through the entire domain will usually pass through more than one cell (unless it is near the edge), and if we know the exit-point from one cell, we automatically know the entry-point to the next cell. We use this implicitly, only finding one intersection-point per cell on average.

We could reverse the hierarchy of loops and instead ask for each cell which rays intersect that cell. Depending on how many rays we are casting and how close together they are relative to the fineness of the spherical grid, for each cell there might be only one ray that intersects it. As such, even if we found a mathematical property that allowed us to use the t-length of the intersection with one ray to more quickly find the t-length of the intersection with a ray shifted 1 to the left — analogous to how using the ray as the top-level loop means we know the t at which we enter a cell from leaving the previous cell — it might not give us anything.

3.7 Cython implementation

The implementation may now be found at `https://bitbucket.org/dHannasch/yt_grid_traversal`. The `cref` links in the comments of the code refer to targets in the `.tex` of this document.

3.8 Conclusion: alternate coordinate system reprise

Strictly speaking, if all you want is to implement the algorithm, then Chapter 3 is all you need. The pseudocode in Section 3.1.5 even gives in brief the necessary conversions from the coordinates. However, to modify the algorithm, you'll need to understand how our alternate coordinate system gives rise to this algorithm. This comes down to the available conversions between the coordinates, specifically how we can convert between the coordinates *given that we know we are on a one-dimensional line.* Further improvements to the algorithm are likely to come hand-in-hand with finding another alternate representation of the space — for example, perhaps representing the zenith angle as a pair of numbers, or representing the azimuth angle as a single number. How useful such a representation is depends on the conversions it makes available: how easy or difficult those conversions are. In Chapter 4, we go in depth into what conversions our alternate coordinate system enables, and what conversions it does *not* enable (specifically, between the radius and the zenith angle; see Section 4.10).

Chapter 4

Conversions

A line has only one degree of freedom, so in principle, if a parameter never repeats along the line (for example $\tan\phi$ never repeats along the line because a line covers a ϕ-range of length only π), that parameter determines a unique point on the line and therefore we can use it to determine a point of intersection. In particular, any value that is one-to-one along a ray can in principle be converted into any other value that is one-to-one along a ray.

Of course, sometimes we have to work with values that are not globally one-to-one along the ray, but they may still be *locally* one-to-one — for example, the radius and the zenith angle are both ditone along the ray, so if we know where the hinge is and the hinge isn't in a particular interval, then the quantity is one-to-one along the ray in that interval.

The significance of this is that when a facet of a cell is defined by a coordinate-value, we can find out whether the ray reaches that facet before reaching any other facet, or find the exact distance (t-value) at which the ray reaches that facet.

A ray is defined by $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \vec{p} + t\vec{d} = \begin{bmatrix} p_x + td_x \\ p_y + td_y \\ p_z + td_z \end{bmatrix}$. This is what we mean when we refer to a t-value. The t-value is trivially convertible to and from any of the Cartesian coordinates x or y or z.

The radius r is never one-to-one on any line (so the same is true of the squared distance to the origin r^2), while by Lemma 4, the zenith angle is one-to-one only if $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle = 0$.

However, the ditone parameters r^2 and $\operatorname{sgnSqrCos}\theta$ *are* two-to-one on any line, id est at most two points on the line can have the same r^2 and at most two points on the line can have the same $\operatorname{sgnSqrCos}\theta$. Furthermore, since these coordinates are ditone, those two points must lie on opposite sides of the hinge for each coordinate. Since in both cases we can calculate the hinge, we can determine t from r^2 or t from $\operatorname{sgnSqrCos}\theta$ as long as we keep track of whether we have yet passed the point nearest the origin (for r^2) and the θ-hinge (for $\operatorname{sgnSqrCos}\theta$).

The precise conversions available for our alternate coordinate system — how quickly the conversions can be done, and how they propagate numerical errors — are used both for determining whether an alternate coordinate system is a good one, and for seeing what algorithms we can construct.

A note on costing The costs of these conversions are heavily hardware-dependent. As such, where appropriate we give the costs in terms of floating-point division `FDIV` and floating-point square root `FSQRT`.

Table 4.1 summarizes the conversions; a bit more detail is below.

- Converting between any Cartesian coordinate (x or y or z) and t requires only one division, and not even that if we precompute $1/d_x$, $1/d_y$, and $1/d_z$.

Table 4.1: Conversions among coordinates.

From	To	Cost
t	x, y, z	trivial (`FMUL` + `FADD`)
x, y, z	r^2	trivial
t	r^2	trivial
r^2	t	1 FSQRT (+ 1 FDIV)[a]
t	$\mathrm{sgnSqrCos}\,\theta$	1 FDIV
$\mathrm{sgnSqrCos}\,\theta$	t	1 FSQRT + 1 FDIV[b]
t	$\cos\phi, \sin\phi$	trivial
$\cos\phi, \sin\phi$	t	1 FDIV[c]
t	$\mathrm{mtan}\,\phi$	1 FDIV
$\tan\phi$	t	1 FDIV[d]
$\tan^2\theta$	t	1 FSQRT + 1 FDIV
r^2	$\mathrm{sgnSqrCos}\,\theta$	unobtained

[a]When casting many parallel rays, the FDIV can be removed by precomputing $1/\left\|\vec{d}\right\|^2$.

[b]When casting many parallel rays, the FDIV can be removed by precomputing $1/(\|d\|^2\cos^2\theta - d_z^2)$ for all zenith breakpoints θ.

[c]When casting many parallel rays, the FDIV can be removed by precomputing $1/(d_y\cos\phi - d_x\sin\phi)$ for all azimuth breakpoints ϕ.

[d]When casting many parallel rays, the FDIV can be removed by precomputing $1/(d_x\tan\phi - d_y)$ for all azimuth breakpoints ϕ.

- Converting $\tan\phi \to t$ requires only one division, by $d_x\tan\phi - d_y$, and we can pre-store those for each breakpoint ϕ.
- Converting $(x, y) \to \tan\phi$ requires only the division y/x.
- Converting $\cos^2\theta \to t$ requires one FSQRT; even if we have r^2 as well, $(\cos^2\theta, r^2) \to z^2 \to t$ also requires an FSQRT.
- Converting $(\phi, \theta) \to r$ requires sines and cosines, so we don't do that.
- Converting $\cos^2\theta = \dfrac{z^2}{r^2} \leftrightarrow r^2$ requires z^2 which requires t. Which is a shame, because this would be useful.
- $r^2\,\mathrm{sgnSqrCos}\,\theta = \mathrm{sgnSqr}\,z$, and $\mathrm{sgnSqr}\,z$ is monotone in t, but $\mathrm{sgnSqr}\,z$ is not monotone in θ, so we could only convert the zenith breakpoints to $\mathrm{sgnSqr}\,z$ on a per-ray basis. This could be done *if* we were able to quickly convert a zenith angle value to the corresponding radius value (or pair thereof) when the ray achieves that zenith angle, so that we could convert the zenith breakpoints to r^2 values and thus trivially to $\mathrm{sgnSqr}\,z$ values; but we were unable to do this.

4.1 Cartesian xyz to r^2

As mentioned above, the square of the radius, unlike the radius, is trivially obtainable from Cartesian coordinates: $r^2 = x^2 + y^2 + z^2$.

There are other representations of the line, of course. $\vec{p}$ can be *any* point on the line: replacing $\vec{p}$ with $\vec{p} + T\vec{d}$ simply shifts all t-values by T. As for $\vec{d}$, only the direction matters; scaling $\vec{d}$ by k only requires dividing all t-values by k.

More radically, any two points on the line can define that line, such as $\vec{p}$ and the point on the line nearest the origin.

Nevertheless, we usually use $\vec{p}$ and $\vec{d}$ for convenience.

4.2 t to r^2

Since Cartesian coordinates are trivially obtainable from t and r^2 is trivially obtainable from Cartesian coordinates, we can easily calculate $r^2 = (p_x + td_x)^2 + (p_y + td_y)^2 + (p_z + td_z)^2$. However, we actually find it more convenient to calculate as $r^2 = \left\langle \vec{p} + \vec{d}t \,\middle|\, \vec{p} + \vec{d}t \right\rangle = t^2 \left\|\vec{d}\right\|^2 + 2t \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle + \|\vec{p}\|^2$, where we can precompute $\left\|\vec{d}\right\|^2$ and $\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle$ and $\|\vec{p}\|^2$. The difference, obviously, isn't great enough to worry about too much.

4.3 r^2 to t

Lemma 9. *On a line $\vec{p} + t\vec{d}$, the t-pair (or singleton) corresponding to a given value of r^2 is:*

$$t = \frac{-\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle \pm \sqrt{\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2 - \left\|\vec{d}\right\|_2^2 \left(\|\vec{p}\|_2^2 - r^2\right)}}{\left\|\vec{d}\right\|_2^2}$$

Note that this is symmetric about the hinge $t = -\frac{\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle}{\left\|\vec{d}\right\|_2^2}$, see Lemma 8.

Both t-values could be positive if $\left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle < 0$, meaning we are moving towards the origin, so we will pass through the spherical shell once going in and once going out.

Actual derivation.

$$\begin{aligned}
r &= \|p + td\|_2 \\
r^2 &= \langle p + td \,|\, p + td \rangle \\
&= \|p\|^2 + 2 \langle p \,|\, d \rangle t + \|d\|^2 t^2 \\
0 &= \|d\|_2^2 t^2 + 2 \langle p \,|\, d \rangle t + \|p\|_2^2 - r^2 \\
t &= \frac{-2 \langle p \,|\, d \rangle \pm \sqrt{4 \langle p \,|\, d \rangle^2 - 4 \|d\|_2^2 \left(\|p\|_2^2 - r^2\right)}}{2 \|d\|_2^2} \\
&= \frac{-\langle p \,|\, d \rangle \pm \sqrt{\langle p \,|\, d \rangle^2 - \|d\|_2^2 \left(\|p\|_2^2 - r^2\right)}}{\|d\|_2^2} \\
&= -\frac{\langle p \,|\, d \rangle}{\|d\|_2^2} \pm \sqrt{\left(\frac{\langle p \,|\, d \rangle}{\|d\|_2^2}\right)^2 + \frac{r^2 - \|p\|_2^2}{\|d\|_2^2}}
\end{aligned}$$

□

Alternate proof. Of course, once the value is already known, we can prove this more simply but plugging in the given value for t.

$$
\begin{aligned}
t &= \frac{-2\langle p \,|\, d\rangle \pm \sqrt{2^2\langle p \,|\, d\rangle^2 - 4\|d\|^2(\|p\|^2 - r^2)}}{2\|d\|^2} \\
t^2 &= \frac{\langle p \,|\, d\rangle^2 \mp 2\langle p \,|\, d\rangle\sqrt{\langle p \,|\, d\rangle^2 - \|d\|_2^2(\|p\|_2^2 - r^2)} + \langle p \,|\, d\rangle^2 - \|d\|_2^2(\|p\|_2^2 - r^2)}{\|d\|_2^4} \\
t^2\|d\|^2 &= \frac{\langle p \,|\, d\rangle^2 \mp 2\langle p \,|\, d\rangle\sqrt{\langle p \,|\, d\rangle^2 - \|d\|_2^2(\|p\|_2^2 - r^2)} + \langle p \,|\, d\rangle^2 - \|d\|_2^2(\|p\|_2^2 - r^2)}{\|d\|_2^2} \\
2t\langle p \,|\, d\rangle &= \frac{-2\langle p \,|\, d\rangle^2 \pm 2\langle p \,|\, d\rangle\sqrt{\langle p \,|\, d\rangle^2 - \|d\|_2^2(\|p\|_2^2 - r^2)}}{\|d\|_2^2} \\
2t\langle p \,|\, d\rangle + t^2\|d\|^2 &= \frac{-\|d\|_2^2(\|p\|_2^2 - r^2)}{\|d\|_2^2} = -(\|p\|_2^2 - r^2) \\
\|p\|_2^2 + 2t\langle p \,|\, d\rangle + t^2\|d\|^2 &= r^2
\end{aligned}
$$

□

We want to avoid $\langle p \,|\, d\rangle^2 - \|d\|_2^2\|p\|_2^2$ to avoid catastrophic cancellation when $\vec{d}$ and $\vec{p}$ are nearly parallel.

4.4 $\cos^2\theta$ to t

Lemma 10. *Let $\vec{p}, \vec{d} \in \mathbb{R}^3$ such that $\vec{p} \neq \vec{0}$ and $\vec{d} \neq \vec{0}$.*

On a line $\vec{p} + t\vec{d}$, given an absolute zenith angle $\cos^2\theta$, if $\|d\|^2\cos^2\theta \neq d_z^2$ (the line has a different z-slope from the double-cone defined by $\cos^2\theta$), the values of t corresponding to a given value of $\cos^2\theta$ are

$$
t = \frac{p_z d_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm |\cos\theta|\sqrt{\left\|p_z\vec{d} - d_z\vec{p}\right\|^2 - \cos^2\theta(\|p\|^2\|d\|^2 - \langle p \,|\, d\rangle^2)}}{\|d\|^2\cos^2\theta - d_z^2}
$$

(If $\left\langle \vec{d} \,\middle|\, d_z\vec{p} - p_z\vec{d} \right\rangle \neq 0$ so the zenith angle has a hinge (see Lemma 4), then by Lemma 5, $\cos^2\theta$ achieves the extremal value $\dfrac{\left\|p_z\vec{d} - d_z\vec{p}\right\|^2}{\|p\|^2\|d\|^2 - \langle p \,|\, d\rangle^2}$ once at the zenith-hinge.)

If $\|d\|^2\cos^2\theta = d_z^2$ (the line has the same z-slope as the double-cone) but $\langle p \,|\, d\rangle\cos^2\theta \neq p_z d_z$, then the line will intersect one cone once, at $t = -\dfrac{1}{2}\dfrac{\|p\|^2\cos^2\theta - p_z^2}{\langle p \,|\, d\rangle\cos^2\theta - p_z d_z}$.

Proof. The line $\vec{p} + t\vec{d}$ will achieve the absolute zenith angle $\cos^2\theta$ only when:

$$
\begin{aligned}
\cos^2\theta := \frac{z^2}{r^2} &:= \frac{(p_z + d_z t)^2}{(p_x + d_x t)^2 + (p_y + d_y t)^2 + (p_z + d_z t)^2} \\
&= \frac{(p_z + d_z t)^2}{\left\langle \vec{p} + \vec{d}t \,\middle|\, \vec{p} + \vec{d}t \right\rangle}
\end{aligned}
$$

$$
\begin{aligned}
\left\langle \vec{p} + \vec{d}t \,\middle|\, \vec{p} + \vec{d}t \right\rangle \cos^2\theta &= (p_z + d_z t)^2 \\
(\|p\|^2 + 2\langle p \,|\, d\rangle t + \|d\|^2 t^2)\cos^2\theta &= p_z^2 + 2p_z d_z t + d_z^2 t^2 \\
(\|d\|^2\cos^2\theta - d_z^2)t^2 + 2(\langle p \,|\, d\rangle\cos^2\theta - p_z d_z)t + \|p\|^2\cos^2\theta - p_z^2 &= 0
\end{aligned}
$$

Case 1: $\left|\vec{d}\right|^2 \cos^2\theta \neq d_z^2$. We can solve the quadratic equation for t in terms of $\cos^2\theta$ at the cost of an `FSQRT`:

$$\begin{aligned}
t &= \frac{-2\vec{p}\cdot\vec{d}\cos^2\theta \mp 2p_zd_z \pm \sqrt{4(\vec{p}\cdot\vec{d}\cos^2\theta - p_zd_z)^2 - 4(\vec{d}\cdot\vec{d}\cos^2\theta - d_z^2)(\vec{p}\cdot\vec{p}\cos^2\theta - p_z^2)}}{2(d_x^2+d_y^2+d_z^2)\cos^2\theta - 2d_z^2} \\
&= \frac{p_zd_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm \sqrt{(\cos^2\theta p^\intercal d - p_zd_z)^2 - (\cos^2\theta d^\intercal d - d_z^2)(\cos^2\theta p^\intercal p - p_z^2)}}{(d_x^2+d_y^2+d_z^2)\cos^2\theta - d_z^2} \\
&= \frac{p_zd_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm |\cos\theta|\sqrt{\cos^2\theta(\langle p\,|\,d\rangle^2 - \|p\|^2\|d\|^2) - 2p_zd_zp^\intercal d + p_z^2d^\intercal d + d_z^2p^\intercal p}}{(d_x^2+d_y^2+d_z^2)\cos^2\theta - d_z^2} \\
&= \frac{p_zd_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm \sqrt{\cos^4\theta(\langle p\,|\,d\rangle^2 - \|p\|^2\|d\|^2) + \cos^2\theta(p_z^2d^\intercal d - 2p_zd_zp^\intercal d + d_z^2p^\intercal p)}}{(d_x^2+d_y^2+d_z^2)\cos^2\theta - d_z^2} \\
&= \frac{p_zd_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm |\cos\theta|\sqrt{\cos^2\theta(\langle p\,|\,d\rangle^2 - \|p\|^2\|d\|^2) + \left\langle p_z\vec{d}\,\middle|\,p_z\vec{d}\right\rangle - 2\left\langle p_z\vec{d}\,\middle|\,d_z\vec{p}\right\rangle + \langle d_z\vec{p}\,|\,d_z\vec{p}\rangle}}{\|d\|^2\cos^2\theta - d_z^2} \\
&= \frac{p_zd_z - \vec{p}\cdot\vec{d}\cos^2\theta \pm |\cos\theta|\sqrt{\left\|p_z\vec{d} - d_z\vec{p}\right\|^2 - \cos^2\theta(\|p\|^2\|d\|^2 - \langle p\,|\,d\rangle^2)}}{\|d\|^2\cos^2\theta - d_z^2}
\end{aligned}$$

Case 2: $\|d\|^2\cos^2\theta = d_z^2$. $\|d\|^2\cos^2\theta = d_z^2$ means $\dfrac{d_z^2}{\|d\|^2} = \cos^2\theta$ id est the line has the same z-slope as the cone. In particular, unless $\vec{d} = \vec{0}$ (in which case our line is not a line at all, just a point), $d_z = 0$ if and only if $\cos^2\theta = 0$.

This leaves us with the linear equation $2(\langle p\,|\,d\rangle\cos^2\theta - p_zd_z)t + \|p\|^2\cos^2\theta - p_z^2 = 0$.

Case 2.1: $\langle p\,|\,d\rangle\cos^2\theta \neq p_zd_z$. In that case the line will intersect one cone once and remain inside thereafter (as opposed to glancing off, as it will if we hit the case where the square root term is zero). The intersection will have $t = -\dfrac{1}{2}\dfrac{\|p\|^2\cos^2\theta - p_z^2}{\langle p\,|\,d\rangle\cos^2\theta - p_zd_z}$.

Case 2.2: $\langle p\,|\,d\rangle\cos^2\theta = p_zd_z$. Recall that we are in the case where $\|d\|^2\cos^2\theta = d_z^2$. Multiplying both sides by $\|d\|^2$, $p_zd_z\|d\|^2 = \langle p\,|\,d\rangle\|d\|^2\cos^2\theta = \langle p\,|\,d\rangle d_z^2$. Thus, $d_z\left\langle \vec{d}\,\middle|\,p_z\vec{d} - d_z\vec{p}\right\rangle = 0$.

Case 2.2.1: $d_z = 0$. Then, since $\|d\|^2\cos^2\theta = d_z^2 = 0$ and $\vec{d} \neq \vec{0}$, $\cos^2\theta = 0$.

Case 2.2.2: $\left\langle \vec{d}\,\middle|\,p_z\vec{d} - d_z\vec{p}\right\rangle = 0$.

We are left with the simple equation $\|p\|^2\cos^2\theta - p_z^2 = 0$ (this zenith angle is only achieved if it is achieved at the starting point).

□

For computation, we might calculate $\|d\|^2\cos^2\theta$ and $\langle p\,|\,d\rangle\cos^2\theta$ and then multiply by $\|p\|^2$ and $\langle p\,|\,d\rangle$ respectively.

On any given line, almost every value of the absolute zenith angle that is achieved is achieved twice: Lemma 10 yields only one such t if and only if either $\theta = \dfrac{\pi}{2}$ (so $\cos\theta = 0$; we intersect the plane $z = 0$ once at $t = -p_z/d_z$) or $\cos^2\theta = \dfrac{\left\|p_z\vec{d} - d_z\vec{p}\right\|^2}{\|p\|^2\|d\|^2 - \langle p\,|\,d\rangle^2}$.

All absolute zenith angles with $0 < \cos^2\theta < \dfrac{\left\| p_z\vec{d} - d_z\vec{p} \right\|^2}{\|\vec{p}\|^2 \left\|\vec{d}\right\|^2 - \left\langle \vec{p} \,\middle|\, \vec{d} \right\rangle^2}$ are achieved twice, but that doesn't mean all signed zenith angles are achieved twice; neither does it mean they are all *not* achieved twice.

Consider $\vec{p} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and $\vec{d} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$. $d_z\vec{p} - p_z\vec{d} = \vec{p} - \vec{d} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$, with norm 1. $\|p\|^2 \|d\|^2 - \langle p \,|\, d\rangle^2 = 2 - 1 = 1$. The line achieves every zenith angle from 1 down to *almost* $-\frac{1}{2}$, but not below that. It does, however, achieve every absolute zenith angle in its range (except 1 and $\frac{1}{2}$) twice; the absolute angles between 0 and $\frac{1}{2}$ are achieved once above and once below the xy-plane, while the angles between $\frac{1}{2}$ and 1 are achieved twice above the xy-plane. $(\|d\|^2 \cos^2\theta - d_z^2)t^2 + 2(\langle p \,|\, d\rangle \cos^2\theta - p_z d_z)t + \|p\|^2 \cos^2\theta - p_z^2 = 0$ becomes $(2\cos^2\theta - 1)t^2 + 2(\cos^2\theta - 1)t + \cos^2\theta - 1 = 0$.

4.5 Signed square cosine (zenith) to t

It would be nice to have a computational procedure to directly find t from $\operatorname{sgnSqrCos}\theta$, rather than finding t from $\cos^2\theta$ and individually checking the sign for each solution. Unfortunately, we have not yet found a way.

$$(\operatorname{sign} z)\cos^2\theta = (\operatorname{sign} z)z^2/r^2$$
$$r^2(\operatorname{sign} z)\cos^2\theta = (\operatorname{sign} z)z^2 = (\operatorname{sign} z)(p_z + td_z)^2$$

Or to use our special function, $\operatorname{sgnSqrCos}\theta = \dfrac{z\sqrt{z^2}}{r^2} = \dfrac{(p_z + td_z)\sqrt{(p_z + td_z)^2}}{\langle p + td \,|\, p + td\rangle}$. We can multiply through by r^2 to get $\operatorname{sgnSqrCos}\theta \,\langle p + td \,|\, p + td\rangle = (p_z + td_z)\sqrt{(p_z + td_z)^2}$.

We could represent the zenith angle by a representative pair of numbers, rather than a single number. Using $\operatorname{sgnSqrCos}\theta$ is essentially a special case of using the pair $(\operatorname{sgnSqr} z, r^2)$ where r^2 is normalized to 1.

A zenith cone (a single cone, not a double-cone) is represented by an equation $az^2 = br^2$ for some a and b. In the case where $\operatorname{sgnSqrCos}\theta < 0$ (and thus $z < 0$), this becomes the single cone $-z^2 = cr^2$ for some $c \in [-1, 0)$, or $-(p_z + d_z t)^2 = c\,\langle p + td \,|\, p + td\rangle$. But the sign of a doesn't really matter.

Plugging in $z^2 = (p_z + td_z)^2$ and $r^2 = \langle p + td \,|\, p + td\rangle$,

$$a\left(p_z^2 + 2p_z d_z t + d_z^2 t^2\right) = a(p_z + d_z t)^2 = b\,\langle p + td \,|\, p + td\rangle = b\left(\|p\|^2 + 2\,\langle p \,|\, d\rangle\, t + \|d\|^2 t^2\right)$$

which becomes the quadratic equation $0 = (b\,\|d\|^2 - ad_z^2)t^2 + 2(b\,\langle p \,|\, d\rangle - ap_z d_z) + (b\,\|p\|^2 - ap_z^2)$. Solving this equation yields

$$t = \frac{2(ap_z d_z - b\,\langle p \,|\, d\rangle) \pm \sqrt{4(b\,\langle p \,|\, d\rangle - ap_z d_z)^2 - 4(b\,\|d\|^2 - ad_z^2)(b\,\|p\|^2 - ap_z^2)}}{2(b\,\|d\|^2 - ad_z^2)}$$
$$= \frac{ap_z d_z - b\,\langle p \,|\, d\rangle \pm \sqrt{(b\,\langle p \,|\, d\rangle - ap_z d_z)^2 - (b\,\|d\|^2 - ad_z^2)(b\,\|p\|^2 - ap_z^2)}}{b\,\|d\|^2 - ad_z^2}$$

The problem is that to make use of this, we'd have to know in advance which cone to intersect the ray with. If we intersect with both single cones, then we might as well just intersect with the double-cone they make together.

4.6 $(\cos\phi, \sin\phi)$ to t

Storing a pair of numbers $(\cos\phi, \sin\phi)$ for the azimuth angle, rather than a single number, is motivated by how simple it makes the code to intersect a ray with an azimuth-facet, as shown below.

```
bool t_from_cosphi_sinphi(double cosphi, double sinphi, double& t) {
  // Sets the parameter t to the t-value at which
  // the line achieves the given azimuth angle.
  // Returns false iff there is no such t.
  double denominator = cosphi*dirVec[1] - dirVec[0]*sinphi;
  if(sign(denominator) != azimuthCCWsign)
    return false;
  double numerator = posVec[0]*sinphi - posVec[1]*cosphi;
  if(denominator == 0) {
    // If d_x == d_y == 0, then the denominator will be zero
    // for all cosphi and sinphi.
    if(dirVec[0] == 0 and dirVec[1] == 0 and numerator != 0)
      return false;
    // Now if d_x == 0 == d_y, then we know p_x*sinphi == p_y*cosphi,
    // so we know we're on the right line.
    // The only question that remains is
    // whether we're on the right side of the z-axis.
    if(sign(cosphi) == sign(posVec[0]) && sign(sinphi) == sign(posVec[1]) )
      // posVec itself is on the correct side of the z-axis, so use that.
      t = 0;
    else if(dirVec[0] == 0 and dirVec[1] == 0)
      return false;
    else if(cosphi == 0 and sinphi == 0)
      return t_at_z_axis(t);
    else
      // far side of the z-axis from posVec
      t = max(max(abs(posVec[0]),
                  abs(posVec[1]) ),
                  abs(posVec[2]) ) + 1;
    return true;
  }
  t = numerator/denominator;
  return true;
}
```

4.7 t to $\tan\phi$ and $\tan\phi$ to t

In the whole space of $\mathbb{R}^3$, $\phi = y/x$ alone does not in general tell us either x or y unless we already know θ and r, but on a given line, ϕ tells us $\tan\phi = \frac{y}{x} = \frac{p_y + d_y t}{p_x + d_x t}$. This becomes $(p_x + d_x t)\tan\phi = p_y + d_y t$ and then, solving for t, $(d_x \tan\phi - d_y)t = p_y - p_x \tan\phi$, so $t = \frac{p_y - p_x \tan\phi}{d_x \tan\phi - d_y}$ and this tells us exactly where we are on the line.

Note that there are two opposite azimuth angles with the same $\tan\phi$; only one of each pair can ever be achieved by a line that does not intersect the z-axis.

If the line does not intersect the z-axis, then the azimuth angle $\tan\phi = \frac{d_y}{d_x}$ (really two opposite azimuth angles) is what the line approaches in the limits at infinity.

If the line does intersect the z-axis, then of course the azimuth angle is constant (really, two opposite azimuth angles that remain constant away from the origin)

If d_x and d_y are the same for many sightlines (which they are in the case of a screen that is a flat plane, projecting one sightline for each pixel) then we can precompute $\frac{1}{d_x \tan\phi - d_y}$ for each breakpoint ϕ.

In the reverse direction, t can easily get us $\tan\phi$. $\phi \mapsto \tan\phi$ is not one-to-one on $[0, 2\pi)$, but the sign of x or the sign of y is sufficient to disambiguate it. Thus $t \leftrightarrow \tan\phi$ *is* one-to-one on any particular line, as is $t \leftrightarrow \phi$, hence so is $\phi \leftrightarrow t \leftrightarrow \tan\phi$.

To make things simpler, however, we can use the augmented tangent mtan : $\left[-\frac{\pi}{2}, 3\frac{\pi}{2}\right) \longleftrightarrow \mathbb{F}_2 \times \mathbb{R}$ with the lexicographic ordering on $\mathbb{F}_2 \times \mathbb{R}$, defining the map monotonically via $\phi \mapsto \left(\phi \geq \frac{\pi}{2}, \tan\phi\right)$.

In actual practice, though, managing the augmented tangent ends up being more expensive than just saving the pair $(\cos\phi, \sin\phi)$. The augmented tangent can of course be compressed into a single double-precision number plus one bit, but it requires unpacking to work with, whereas the pair $(\cos\phi, \sin\phi)$ can be used directly.

4.8 θ to $\cos^2\theta$

Since we can't very well ask all the application scientists to use our special coordinate system that we defined for ray-tracing convenience, whenever we ingest a data set, we must convert it. On very fine meshes, precision of conversion can be important.

The boundary-values of the zenith angle are frequently defined as rational multiples of π — for example, if the mesh adaptively refined near the xy-plane, we might end up with zenith breakpoints of $\left[0, \frac{\pi}{4}, \frac{14}{32}\pi, \frac{15}{32}\pi, \frac{16}{32}\pi, \frac{17}{32}\pi, \frac{18}{32}\pi, \frac{3}{4}\pi, \pi\right]$.

We have not yet made use of this.

4.9 $\tan\theta$ to t

sgnSqrCos θ wasn't handed down from on high; we could in theory use a different derived quantity, such as $\tan\theta$.

If $0 \leq \theta \leq \frac{\pi}{2}$, $z^2 \tan^2\theta = x^2 + y^2$ so $(p_z + td_z)^2 \tan^2\theta = (p_x + td_x)^2 + (p_y + td_y)^2$ so

$$t^2(d_z^2 \tan^2\theta - d_x^2 - d_y^2) + 2t(p_z d_z \tan^2\theta - p_x d_x - p_y d_y) + p_z^2 \tan^2\theta - p_x^2 - p_y^2 = 0$$

$$t =$$
$$\frac{-2(p_z d_z \tan^2\theta - p_x d_x - p_y d_y) \pm \sqrt{4(p_z d_z \tan^2\theta - p_x d_x - p_y d_y)^2 - 4(d_z^2 \tan^2\theta - d_x^2 - d_y^2)(p_z^2 \tan^2\theta - p_x^2 - p_y^2)}}{2(d_z^2 \tan^2\theta - d_x^2 - d_y^2)}$$
$$= \frac{-(p_z d_z \tan^2\theta - p_x d_x - p_y d_y) \pm \sqrt{\xi}}{d_z^2 \tan^2\theta - d_x^2 - d_y^2}$$

where

$$\begin{aligned}
\xi &= (p_z d_z \tan^2\theta - p_x d_x - p_y d_y)^2 - (d_z^2 \tan^2\theta - d_x^2 - d_y^2)(p_z^2 \tan^2\theta - p_x^2 - p_y^2) \\
&= p_z^2 d_z^2 \tan^4\theta - p_z^2 d_z^2 \tan^4\theta \\
&\quad - 2p_x p_z d_x d_z \tan^2\theta \dagger p_x^2 d_z^2 \tan^2\theta \dagger d_x^2 p_z^2 \tan^2\theta \\
&\quad - 2p_y p_z d_y d_z \tan^2\theta \dagger p_y^2 d_z^2 \tan^2\theta \dagger d_y^2 p_z^2 \tan^2\theta \\
&\quad \dagger (p_x d_x + p_y d_y)^2 - (d_x^2 + d_y^2)(p_x^2 + p_y^2) \\
&= 0 \\
&\quad + (p_x d_z - p_z d_x)^2 \tan^2\theta \\
&\quad + (p_y d_z - p_z d_y)^2 \tan^2\theta \\
&\quad + p_x^2 d_x^2 + 2p_x d_x p_y d_x + p_y^2 d_y^2 - p_x^2 d_x^2 - p_y^2 d_y^2 - d_x^2 p_y^2 - d_y^2 p_x^2 \\
&= \left[(p_x d_z - p_z d_x)^2 + (p_y d_z - p_z d_y)^2\right] \tan^2\theta \\
&\quad - (d_x p_y - d_y p_x)^2
\end{aligned}$$

We can probably agree that sgnSqrCos θ is more convenient overall.

4.10 $(\operatorname{sign}\cos\theta)\cos^2\theta$ to r^2

One thing that slows us down is that we always calculate the exact length of the intersection of the ray with each cell, which requires an `FSQRT` for each cell. In the case of a very fine grid — or in the case when the source feeding us field data is linearly interpolating behind the scenes so that the grid appears finer than it is — using an approximate length instead would speed up the operation without necessarily being the dominant source of error.

At the moment, the only method of determining whether we depart a given cell through its r-facet or its θ-facet is by calculating the t-values when the ray intersects each, requiring *two* `FSQRT`s. Fortunately we can skip this if we leave via the azimuth facet first, but this is still something that must be overcome if we want to not have to calculate exact t-values every time.

Despite the fact that both r and θ are not one-to-one along a line, in most cases it is possible *in principle* to convert one into the other:

Lemma 11. *If the radius r and zenith angle θ do not always determine a unique point on the line, then the line has constant z (and may therefore be treated as a line in two-dimensional polar coordinates, completely ignoring the zenith angle).*

Proof. The level sets of the zenith angle are cones. The level sets of the radius are spheres. The intersection of a θ-cone with an r-sphere is a circle of constant z centered around the z-axis (id est its center has $x = y = 0$). A line that contains two distinct points on such a circle has two distinct points of the same z, so $d_z = 0$. $\square$

So it *is* possible to get an equation relating θ and r for all the cases that matter, but figuring out how to use a fact like "z varies" is nontrivial.

If we project onto the plane $y = 0$, which corresponds to $\phi \in \{0, \pi\}$, we lose r.

If we know that r^2 is decreasing and $\frac{z^2}{r^2}$ is increasing (or the reverse, just as long as not moving in the same direction), then if we multiply r_1^2 with $\frac{z_2^2}{r_2^2}$: if the result is less than z_2^2, then we know $r_1^2 < r_2^2$. But we have no way to get z_2^2 to be able to do that.

Conclusion The lack of a ready conversion between the zenith angle and the radius is the main reason we cannot recommend using the alternative, more complicated serial version of our algorithm, which we have avoided describing until now. Nevertheless, if you've read this section you may have ideas for improvements, so we will proceed to describe the serial version of the algorithm in Chapter 5. However, we expect the more likely route to immediate improvements will come from using the geometry of the grid to pre-truncate the ray to the domain, to be described in a forthcoming paper.

Chapter 5

More complicated algorithm

Hewett[Hew12], for each (non-rectangular) cell, calculated the t-value for all six facets; the ray leaves through the lowest t-value. That works when all facets are planar, but curved surfaces are more expensive to intersect, requiring `FSQRT`. That's why, in both versions of the algorithm, we put more emphasis on minimizing the number of intersection-calculations.

However, the reader will note that in the simpler version of the algorithm above, precisely because we calculate the boundary-crossings for each of the three coordinates in parallel, we calculate more intersections than we need to. Indeed, when we go to interleave the t-values in Section 3.1.4, we end up discarding t-values that turn out to lie outside the domain. For example, if the domain only includes azimuth angles between 0 and $\frac{\pi}{8}$, but has many divisions in the r-domain and the θ-domain, then for many rays, most of the r- and θ-boundary-crossings will happen outside the domain. Thus, while the simpler algorithm works well when the domain is close to a full sphere, for smaller domains it can have wasted intersection-calculations.

Of course, *azimuth* facets *are* planar, as shown in Fig. 5.1. Using our alternate coordinate system, we can quickly calculate intersections of rays with azimuth-facets, so we don't need to worry so much about minimizing those calculations. This suggests calculating all the azimuth-boundary-crossings first, and then calculating the boundary-crossings for the radius and zenith as needed, in a serial fashion.

One obvious alternative is to start by finding all the segments of the ray that are in the domain, of which there will be multiple, because a spherical grid that is less than a full sphere is not in general convex. We can then find the boundary-crossings only within those intervals, which would guarantee that no ray-surface-intersection calculations are wasted. This may indeed be feasible, and will be addressed in a later paper that delves into the geometry of how rays intersect spherical grids.

ϕ and $\tan\phi$ are monotone only in a wrap-around sense; they're always *locally* increasing, but to call them truly monotone we need to define an ordering of the real numbers specific to the given line. Mathematically we can always rotate our entire coordinate system for each line so that the line now runs from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ or somesuch, but if we did that in a computer it would probably make debugging a nightmare. But there is

Figure 5.1: Overhead view of breakpoints in the azimuth angle; each of these half-planes is the facet of two adjoining cells.

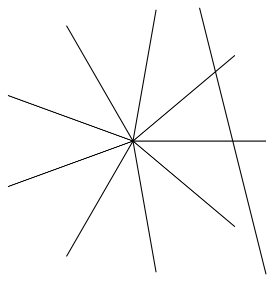
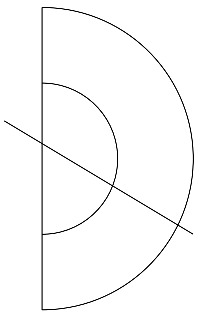


Figure 5.2: When we are in the angle-bracket of the closest approach to the origin, we will go down towards the origin and then come back up before leaving via an angle-facet, but we won't necessarily come up to the same r-annulus we started at.



still a way to take advantage of the fact that ϕ is monotone along the line.

$\{\cos\phi, \sin\phi\} \mapsto t \mapsto \{\cos^2\theta, r^2\}$ requires only 2 `FDIV` (plus some `FADD`). Since it is easy to get the others from $\tan\phi$, let ϕ be the independent variable some of the time, the basis of our outer for-loop. When determining whether we leave via the r-facet or the θ-facet before leaving via the ϕ-facet, we calculate $\tan\phi_{\max} \to t \to \big(r^2, (\operatorname{sign} z)\cos^2\theta\big)$ and check whether the ray departs the r-annulus or θ-bracket before leaving the ϕ-bracket. This can give the wrong answer only when the ray leaves via the r-facet or θ-facet and then returns before finally leaving via the ϕ-facet. It is possible to leave and then return only near the respective hinges of the ditone coordinates.

If we find that the ray did leave via r or θ first, then we need to know which of r and θ it was, and lacking a direct $r \leftrightarrow \theta$ conversion on a line, we must spend the two FSQRTs to calculate the exact t-values of both.

- If $\tan\phi_{\max} \to t \to r^2$ says we're still inside the r-annulus (and not in angle bracket of closest approach) while $\tan\phi_{\max} \to t \to (\operatorname{sign} z)\cos^2\theta$ is outside the θ-range, then the ray definitely leaves via the θ-facet first.
- If $\tan\phi_{\max} \to t \to r^2, (\operatorname{sign} z)\cos^2\theta$ says we're still inside both the r-annulus and zenith-range (and not near either hinge), then the ray definitely leaves via the ϕ-facet first.
- If $\tan\phi_{\max} \to t \to (\operatorname{sign} z)\cos^2\theta$ says we're still inside the zenith-range (and we're not near the zenith hinge, where the line go pass out and then back in) while $\tan\phi_{\max} \to t \to r^2$ is outside the r-annulus, then the ray definitely leaves via the r-facet first.
- If $\tan\phi_{\max} \to t \to r^2$ says we're outside the r-annulus and $\tan\phi_{\max} \to t \to (\operatorname{sign} z)\cos^2\theta$ says we're outside the zenith-range, then the ray intersects the azimuth-facet *last*, and we must calculate the exact t-values at which the line intersects the r-facet and θ-facet in order to determine which comes first; whichever has the lower t-value is the true exit point of the cell.

5.1 Multiple intersections

One shortcut we would like to be able to take is detect when we're in the angle-range of the closest approach to the origin and drop straight down towards the origin, then climb back as a special case. The point of this is that then everywhere else, we always know whether we're approaching the origin or retreating from the origin at any given moment, so we only have to check one r-facet.

Unfortunately, it is possible for a line to intersect a cell in three separate intervals, because really it's (convex set $\cap$ convex set) $\setminus$ (convex set) $\setminus$ (convex set) = (convex set) $\setminus$ (convex set) $\setminus$ (convex set):

Consider a zone defined by $1 < r \leq 2$ and $0 \lneq \theta \leq \pi$. Consider the ray $\begin{bmatrix}-1 & 0 & 20\end{bmatrix} + t\begin{bmatrix}1 & 0 & -10\end{bmatrix}$. Since $\theta = 0$ is not included, when t reaches 1 the ray reaches $\begin{bmatrix}0 & 0 & 10\end{bmatrix}$ and passes outside the zone and then re-enters. Since $r = 1$ is not included, when t reaches 2 the ray reaches $\begin{bmatrix}1 & 0 & 0\end{bmatrix}$ and passes outside the zone and then re-enters.

Consider a zone defined by $1 \leq r \leq 2$ and $\varepsilon \leq \theta \leq \pi$. Consider the ray $\begin{bmatrix}1 & 0 & 0\end{bmatrix} + t\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$. At $t = 0$, the ray departs the zone via the r-boundary. At $t = 1$, the ray reaches $\begin{bmatrix}0 & 0 & 1\end{bmatrix}$. Shortly after $t = 1$, the ray re-enters the zone via the θ-boundary. This shows that if the hinge of r falls near the hinge of θ, then the ray does not have to strictly go down-then-up for the closest approach. Running this in reverse, we see we can also leave on θ and come back via r.

Case 1: θ-hinge inside the θ-bracket of closest approach. Then we will only be in that θ-bracket for a single run (which might cross several r-boundaries).

Case 2: r-hinge before θ-hinge (in terms of t), θ-hinge outside the θ-bracket of the closest approach. Then, when we're first in the angle-bracket of closest approach, this must be the time for the closest approach because we cannot reverse on θ before the θ-hinge.

Case 3: θ-hinge before r-hinge, θ-hinge outside the θ-bracket of closest approach. Then the closest approach must occur the second time we are in the θ-bracket (but might never be in both the θ-bracket and the ϕ-bracket before the closest approach, though we could be). That is, we will leave on θ and return to this θ-bracket (not necessarily to this zone) before the closest approach.

Case 4: r-hinge before θ-hinge and not yet `retreatingFromOrigin`. Then if we're in the angle-bracket of closest approach, this must be the closest approach.

But the radius is not the only ditone parameter: there is also the zenith angle. Can we detect, from r and ϕ, when θ is about to reach its extremum, and otherwise know the direction θ is trending so that we only have to check for intersection with one θ-facet?

We need to know whether θ is increasing or decreasing when calculating `tAtWhichLeaveByTheta`. If θ is increasing, then we need to check the upper boundary; if θ is decreasing, then we need to check the lower boundary. (We could check both boundaries, but each such calculation requires an expensive FSQRT.) So we need to know at each step whether θ will increase or decrease, which requires knowing when we reach the hinge (where θ changes from increasing to decreasing or vice versa).

If we're in the r-annulus and ϕ-bracket of the θ-hinge,

Case 1: r-minimum in the r-annulus of the θ-hinge. Then we will only be in this r-annulus once. However, the closest approach to the origin might still occur in a different θ-bracket from the θ-hinge.

Case 2: r-minimum past the r-annulus of the θ-hinge, r-hinge before θ-hinge. Then the θ-hinge will occur the second time we are in that r-annulus.

Case 3: r-minimum past the r-annulus of the θ-hinge, r-hinge after θ-hinge. Then the θ-hinge will occur the first time we are in that r-annulus.

Of course, if the azimuth angle is decreasing along the line then we will be looking for $\phi_{\min}$ rather than $\phi_{\max}$.

Since r and θ are ditone in t and t is monotone in ϕ (and even $\tan\phi$; remember $\tan\phi$ *is* a one-to-one function when restricted to the ϕ-range of any line), r and θ are ditone in ϕ.

5.2 Overall pseudocode

The heart of the serial version of the algorithm is that we use only $O(1)$ memory (regardless of the size of the domain) per ray. Everything else is compromised in the name of that goal. This more complicated version of the algorithm only looks at the bounds on one cell at a time, so it can directly handle spherical grids with selectively refined patches, such as the one shown in Figure 1.3, without the need for an external framework to call the algorithm again on the patch of finer grid. This is less useful than it sounds, since there's no performance cost for calling the algorithm multiple times.

Instead of calculating all the t-values in advance, we calculate them as we go. We always have a current cell that we're traversing. Like Hewett's algorithm, we compare the facets against each other to determine which facet the ray leaves through. Unlike Hewett's algorithm, we have access to special geometric structure of the spherical grid which allows us to avoid intersecting the ray with all six facets. We find the ditone hinges of the radius and zenith angle in advance, so that we always know whether each of the coordinates is increasing or decreasing at any point. And of course the azimuth angle is monotone along any ray, so we can immediately cut the number of facets we need to intersect from six down to two. We always calculate the t-value of the intersection of the ray with the azimuth facet, simply because it doesn't cost much to do so, as the azimuth facet is planar. If we could convert an r^2 value to a $\operatorname{sgnSqrCos}\theta$ value or vice versa along the ray (see Section 4.10), then we could immediately eliminate one of the more expensive facets. What we *can* do is use the t-value of the azimuth facet to quickly compute values for r^2 and $\operatorname{sgnSqrCos}\theta$ and check whether those values are still in the cell. If either is not, then we know we *don't* leave the cell via the azimuth facet, so we'll need to intersect the ray with one of the curved facets. The question that remains is whether we can avoid intersecting the ray with *both* of the curved facets. There are two situations when we might need to get the exact t-value:

- When the computed values for r^2 and $\operatorname{sgnSqrCos}\theta$ (from the t-value of the azimuth facet) are both outside the cell, so that we know the ray leaves via one of those two facets but we don't know which.
- When the computed value for r^2 or $\operatorname{sgnSqrCos}\theta$ (from the t-value of the azimuth facet) is inside the cell, but we are near the ditone hinge, so that the ray might leave the cell and re-enter. A forthcoming paper will give a more complete treatment to how a ray can leave and re-enter one of these grid cells. For now, we must be conservative.

```
**procedure** PRECOMPUTE FOR VOLUMECONTAINER
    precompute $\tan\phi$ (or possibly $\cot\phi$) for breakpoints $\phi$, accessible by index
    precompute $r^2$ (strictly increasing array)
    precompute $(\operatorname{sign} z)\cos^2\theta$ (strictly decreasing array, probably store in reverse order so increasing, more intuitive to increase index when value increases)
    maybe for convenience maintain pointers r2min and r2max where r2max = r2min + sizeof(np.float) so never have to remember to use index+1 or index-1
**end procedure**
```

```
procedure PRECOMPUTE ONCE KNOW SIGHTLINES
    if $d_x$ and $d_y$ same for all sightlines then store $\frac{1}{d_x \tan\phi - d_y}$ for all breakpoints
end procedure
                    ▷ probably no point in exporting nextCellSphere as its own function, since we do different things depending on whether descending-due-to-closest-approach or not, so any function only called from one place
function WALKVOLUMESPHERE(VolumeContainer, CartPosition, CartDirection)
    $\phi_{\text{start}} \leftarrow \phi(\texttt{CartPosition})$
    $\phi_{\text{end}} \leftarrow \phi(\texttt{CartPosition + CartDirection})$
▷ $\phi_{\text{start}} < \phi_{\text{end}} < (\phi_{\text{start}} + \pi \mod 2\pi)$ won't catch if $(\phi_{\text{start}} + \pi \mod 2\pi)$ wraps around but $\phi_{\text{end}}$ does not
    if $\phi_{\text{start}} < \phi_{\text{end}} < \phi_{\text{start}} + \pi$ or $\phi_{\text{start}} < \phi_{\text{end}} + 2\pi < \phi_{\text{start}} + \pi$ then
        phiBreakpoints ← np.array view from VolumeContainer (increasing, all $> \phi_{\text{start}}$, all $< \phi_{\text{end}}$ since don't want to go past $\phi_{\text{end}}$)  ▷ if cross $2\pi$, want to wrap around whatever index VolumeContainer uses so that this array has exactly the sequence we'll visit
        $\tan\phi \leftarrow$ np.array view from VolumeContainer
    else                                                    ▷ get same arrays but in reverse order
        ASSERT $\phi_{\text{start}} - \pi < \phi_{\text{end}} < \phi_{\text{start}}$ or $\phi_{\text{start}} - \pi < \phi_{\text{end}} - 2\pi < \phi_{\text{start}}$
        phiBreakpoints ← np.array view from VolumeContainer (decreasing, all $< \phi_{\text{start}}$, all $> \phi_{\text{end}}$)
        $\tan\phi \leftarrow$ np.array view from VolumeContainer
    end if
                                        ▷ assume domain is full sphere ($\phi \in [0, 2\pi]$, $\theta \in [0, \pi]$)
▷ if $\langle p \mid d \rangle < 0$ id est moving towards the origin then might need to increase $\phi_{\text{start}}$ to where enter domain
    $t$ at which enter domain $= \frac{-\langle p \mid d \rangle - \sqrt{\langle p \mid d \rangle^2 - \|d\|_2^2 \left(\|p\|_2^2 - r_{\max}^2\right)}}{\|d\|_2^2}$
    $t$ at which will exit domain $= \frac{-\langle p \mid d \rangle + \sqrt{\langle p \mid d \rangle^2 - \|d\|_2^2 \left(\|p\|_2^2 - r_{\max}^2\right)}}{\|d\|_2^2}$
                                                    ▷ find angle bracket of closest approach
    closestApproach ← $\vec{p} - (\vec{p}\cdot\vec{d})\vec{d}/(\vec{d}\cdot\vec{d}) = \vec{p} + \vec{d} - \vec{d} - (\vec{p}\cdot\vec{d})\vec{d}/(\vec{d}\cdot\vec{d}) = \vec{p} + \vec{d} - \frac{\vec{d}\cdot\vec{d} + \vec{p}\cdot\vec{d}}{\vec{d}\cdot\vec{d}}\vec{d} = \vec{p} + \vec{d} - \frac{(\vec{p}+\vec{d})\cdot\vec{d}}{\vec{d}\cdot\vec{d}}\vec{d}$
    indexOfPhiRmin ← find *bracket* (don't need exact value) of $\phi$(closestApproach), $-1$ if between $\phi_{\text{start}} - \pi$ and $\phi_{\text{start}}$, len(phiBreakpoints) + 1 if between $\phi_{\text{end}}$ and $\phi_{\text{end}} + \pi$
    initRderiv ← $\frac{\partial}{\partial t} r^2 = \frac{\partial}{\partial t} \|p + td\|^2 = \frac{\partial}{\partial t}\left((p_x + td_x)^2 + (p_y + td_y)^2 + (p_z + td_z)^2\right) = (p_x + td_x)d_x + (p_y + td_y)d_y + (p_z + td_z)d_z$
    retreatingFromOrigin ← (indexOfPhiRmin == $-1$)      ▷ once we start retreating from the origin, we will keep retreating from the origin
    if not retreatingFromOrigin then
        indexOfThetaRmin ← find *bracket* (don't need exact value) of $\theta$(closestApproach)
        indexOfClosestR ← find *bracket* (don't need exact value) of $r$(closestApproach)
    end if
    currPhi ← $\phi_{\text{start}}$
    for nextPhi in phiBreakpoints $\cup \phi_{\text{end}}$ do
        moveOnPhi ← false
        while not leaving on $\phi$ do
    enterT ← exitT                                ▷ whatever we left the prev at, that's where we enter
    if $\phi, (\operatorname{sign} z)\cos^2\theta$ is the stored angle-pair of closest approach then
                                                  ▷ go down to indexOfClosestR, then come back up
        for rBreakpoints between current $r$ and indexOfClosestR do
            run sampler
        end for
        retreatingFromOrigin ← true        ▷ seems like it would be even more awkward to break into
two for-loops (one for before closest approach, one for after)
    end if                    ▷ no Else because after go down-then-up check normally how leave again
    tAtWhichLeaveByPhi ← $t = \frac{p_y - p_x \tan\phi}{d_x \tan\phi - d_y} = \frac{p_y \cot\phi - p_x}{d_x - d_y \cot\phi}$   ▷ only need $\tan\phi$ for breakpoints,
not for $\phi_{\text{start}}$ nor $\phi_{\text{end}}$
                                                  ▷ check what $r$- and $\theta$-proxies will be when leave on $\phi$
    $r^2 = (\vec{p} + t\vec{d}) \cdot (\vec{p} + t\vec{d})$
    $(\operatorname{sign} z)\cos^2\theta = (\operatorname{sign} z)\frac{z^2}{r^2} = \operatorname{sign}(p_z + d_z t)\frac{(p_z + d_z t)^2}{(p_x + d_x t)^2 + (p_y + d_y t)^2 + (p_z + d_z t)^2}$
    thetaIncreased ← $(\operatorname{sign} z)\cos^2\theta \geq$ signedSqrCosUpper[indexOfCurrTheta]
    thetaDecreased ← $(\operatorname{sign} z)\cos^2\theta \leq$ signedSqrCosLower[indexOfCurrTheta]
    thetaOutsideCell ← thetaIncreased or thetaDecreased
    if retreatingFromOrigin then
        rOutsideCell ← $r^2 \geq$ rMax
    else
        rOutsideCell ← $r^2 \leq$ rMin
    end if
    if thetaOutsideCell then
                                                  ▷ then we left on $\theta$ before leaving on $\phi$
        if rOutsideCell then
▷ Here we need to compare the radius and zenith values against each other. Unfortunately, as noted in
Section 4.10, we can't. So we go through $t$.
▷ It's marginally easier to transform $r^2 \to t \to \operatorname{sgnSqr} z/r^2$ than the reverse, but we still do need $t$ to get
the value of $z$.
            tAtWhichLeaveByR ← $\frac{-2\langle p \mid d\rangle \pm \sqrt{4\langle p \mid d\rangle^2 - 4\|d\|_2^2(\|p\|_2^2 - r_{\text{bound}}^2)}}{2\|d\|_2^2}$,
▷ The correct $t$-value to keep is the $t$-value that is greater than the current $t$-value; if both are, then the
correct $t$-value is the smaller (sooner) value.
            cosWhenLeaveOnR ← $\frac{z^2}{r^2}$
            if cosWhenLeaveOnR inside cell then
                leavingOnTheta ← false
                exitT ← tAtWhichLeaveByR
                leave on $r$ means run sampler then currR ←
            else                                  ▷ cosWhenLeaveOnR outside cell
                leavingOnTheta ← true
            end if
        else                                      ▷ r inside cell
```

```
                leavingOnTheta ← true
            end if
            if leavingOnTheta then
                leave on $\theta$ means run sampler then indexOfCurrTheta ←
                if thetaIncreased then
                    indexOfCurrTheta += 1
                else
                    indexOfCurrTheta -= 1
                end if
            end if
        else                                                            ▷ theta inside cell
            if rOutsideCell then
                leave on $r$ means run sampler then currR ←
            else                                                        ▷ r inside cell
                exitT ← tAtWhichLeaveByPhi
                moveOnPhi ← true
                leave on $\phi$ just means run sampler and continue the for-loop
            end if
        end if
    end while
  end for
end function
```

Conclusion Regardless of whether we can get a bit more efficiency from the serial version — or even if the parallel version turns out to be more efficient overall — we need to know whether the new algorithm outperforms the simple alternative of resampling to a Cartesian grid and using pre-existing ray-casting algorithms. It seems *unlikely* to result in greater errors overall, since the central idea of the algorithm is to work with the data in-place rather than performing more operations than strictly necessary. But the different computations we do will certainly result in a different pattern of floating-point errors, especially when we bring in `FSQRT`s, and we do not currently have any rigorous proof bounding the propagation of those floating-point errors. This requires testing. We also need to know just how much those `FSQRT`s cost us in computation time. Chapter 6 will see how our new approach performs in practice.

Chapter 6

Validation and examples

Since we did not identify an existing accepted set of benchmark tests for volume-rendering spherical domains, we must construct our own.

Fig. 6.1 shows a simple example: we cast rays through a sphere of constant density and take simple integrals, so that the known analytic solution is $2\sqrt{y^2+z^2}$. This serves as a useful sanity check, but to test whether our method preserves critical features of the data, we need test cases that really do have features we need to preserve.

6.1 Methodology

Benchmark tests are constructed by taking an analytically defined function and sampling it on either a spherical grid or a Cartesian grid.

Which value to assign to a grid cell is not immediately obvious. Taking the midpoint of all coordinates for a spherical-grid cell does not give us the geometric center: the geometric center, the point at which the maximum distance to any other point is minimized, would actually be at a slightly lower radius.

If we're casting rays, we might want to pick a point in yet a third way: if there is a single point common to all longest line segments through the cell (as distinct from the point with the shortest longest line segment through that point, which definitely exists but doesn't mean much), then that point might be a good choice, since it's good for the longest line segments, the ones where the error in this cell matters most.

In rectangular box cells, all three points coincide: taking the midpoint in each coordinate yields the geometric center of the cell, and any longest line segment through the cell must pass through that point.

If we stick to vertex-centered data, sampling the true function at the vertices of the grid, then the question of where the field data for a *cell* truly resides is irrelevant. Alternatively, the question is equally irrelevant if we do no interpolation — which is likely better anyway, since almost any interpolation scheme would work better for one grid than the other. An assumption that the underlying field data is linear in the radius doesn't take much extra computation when marching through a spherical grid, but it's rather more expensive when marching through a Cartesian grid.

Sampling the true function on the Cartesian grid assumes that sampling can be done *perfectly*, which may not always be possible; the original simulation was (usually) on a spherical grid for a reason, and no matter what clever interpolation is used, it may not be perfect. So our testing here is leaning on the side of letting the Cartesian grid work as well as it possibly can.

We rotate our domain so that the rays are precisely aligned with the x-axis. This is, again, to ensure we allow the Cartesian ray-casting to work as well as it can when compared to the spherical ray-casting.

We use simple step functions with easily-calculable exact solutions. These roughly correspond to situations we might encounter, such as how the Earth's temperature and density splits into clearly defined layers

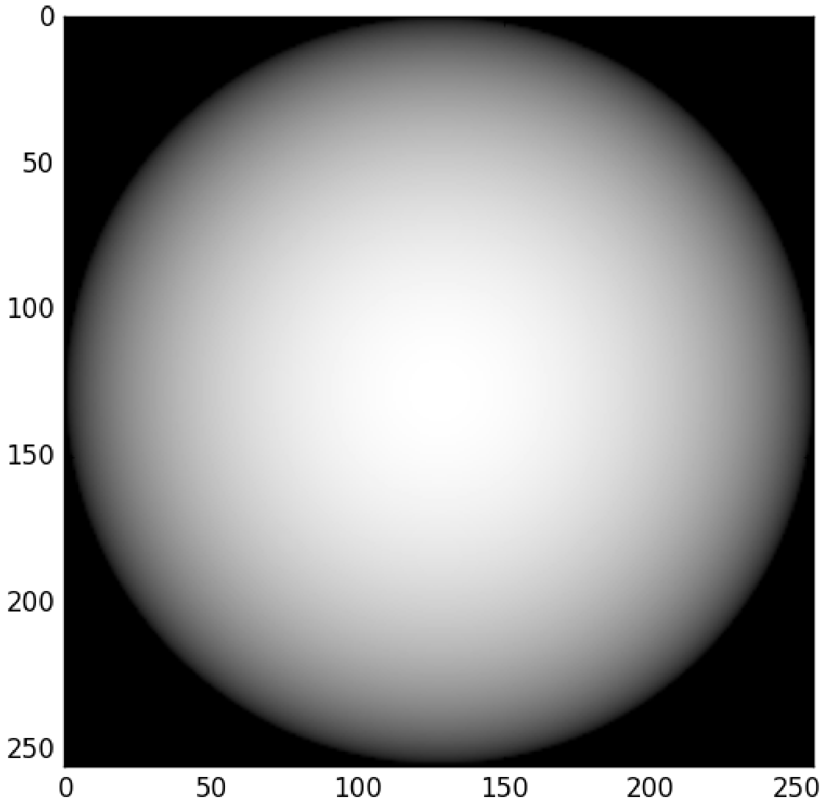


Figure 6.1: Ray-casting through sphere of constant density, showing limb-darkening. The brightness represents the value of the integral, the total thickness the ray passes through.

at the Mohorovicic Discontinuity and the Gutenberg Discontinuity.

For example, $\chi_{0.5}(\vec{v}) := \begin{cases} 2 & r \leq 0.5 \\ 1 & 0.5 < r \leq 1 \\ 0 & r > 1 \end{cases}$. We can calculate an analytic solution for any ray by rotating the domain so that the ray is parallel to the x-axis. Then the equation is $\begin{bmatrix} -R \\ y \\ z \end{bmatrix} + t \begin{bmatrix} 2R \\ 0 \\ 0 \end{bmatrix}$ for constant y and z. The length of the intersection of the ray with the sphere is $2\sqrt{\max(0, R^2 - y^2 - z^2)}$. For simplicity, we usually take $R = 1$.

If the function perfectly fit the spherical grid, then we couldn't say much, because while we're interested in functions that are more natural in spherical coordinates in some sense, we usually won't set the boundaries of the grid exactly right.

For the radius step functions, with steps at radii 1/2 and 3/4, the spherical grids are evenly spaced with *number of breakpoints* a power of two, so that the number of *ranges* is one less than a power of two, so that the step-point of the underlying true function is *not* a breakpoint in the grid. Instead the step 1/2 lies exactly midway between two breakpoints of the grid: $\frac{2^{k-1} - 1}{2^k - 1} < \frac{1}{2} < \frac{2^{k-1}}{2^k - 1}$. $(\frac{2^{k-1}}{2^k - 1} - \frac{1}{2} = \frac{2^k - 2^k + 1}{2^{k+1} - 2} = \frac{1}{2^{k+1} - 2} = \frac{1}{2}\frac{1}{2^k - 1})$

For *volume rendering*, we interpolate by treating the data as linear with respect to the radius (or sometimes the square of the radius, which saves a bit of computation time), which could potentially allow us to 'cheat' here with an exact computation, since the integral of a step function with the step exactly in the middle equals the integral of a linear function. However, the sampler function we use here for testing is a simple `yt.utilities.lib.image_samplers.ProjectionSampler`, which does no interpolation — it simply multiplies the cell value by the length of the intersection of the ray with the cell.

6.2 Results

There is a dramatic difference in the errors: compare Fig. 6.2 with Fig. 6.6 and Fig. 6.3 with Fig. 6.7. A note on reproducibility: all our plots are made using our implementation of the algorithm at `https://bitbucket.org/dHannasch/yt_grid_traversal`.

We show absolute error rather than relative error here because if we use relative error, inevitably the largest relative errors are at the very edges of the domain, where the rays just barely dip in so that the true answer is very small and thus the relative error is much larger than anywhere else; this washes out all the other errors, so that they become imperceptible. Even using absolute error, you can still compare the results of the spherical and Cartesian algorithms, since the true answer is of course the same in both cases. We do show relative error in Fig. 6.8 and Fig. 6.9.

As a point of interest, given an underlying field that is much larger at the edges, so that the true integrals at the edges are not much smaller than the true integrals in the middle, we don't have that problem and can thus show the errors as an image, for example Fig. 6.10. Of course, this is a bit academic, since in most current scientific applications it would be very strange for the field to be larger at the edges of the domain. (This is only really interesting if you want to experiment with our and future algorithms and make plots of these of your own.)

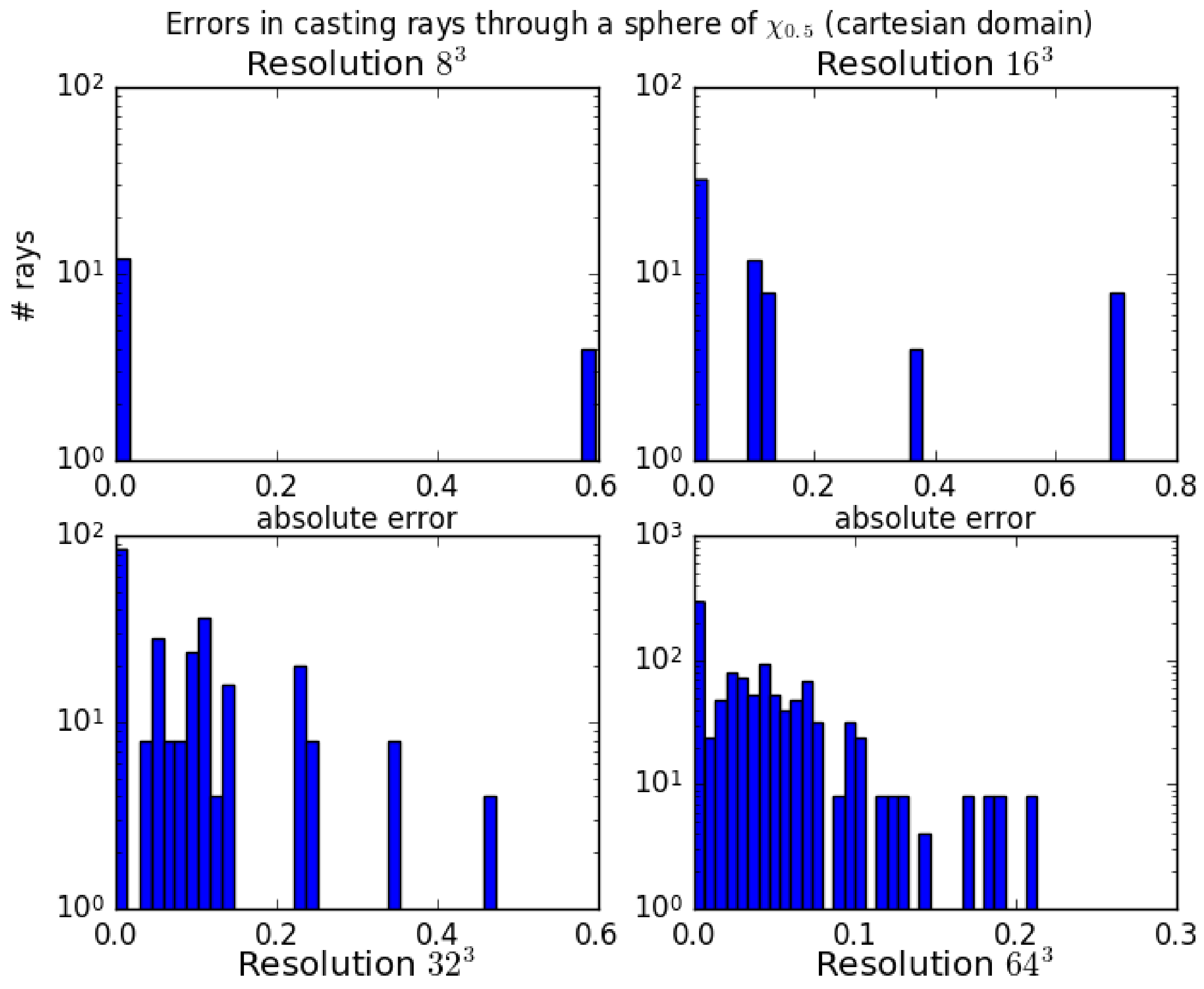


Figure 6.2

At higher resolutions, of course, Cartesian ray-tracing does converge to the true solution, as shown in Fig. 6.4, although Fig. 6.5 shows that the errors are still larger than in lower-resolution spherical ray-tracing.

6.3 Cost

We do not charge the Cartesian ray-casting for the time needed to resample a spherical domain to Cartesian, since for a static dataset that only needs to be done once to take any number of views onto that dataset. Note however that for real-time applications, such as watching an evolving simulation, resampling time would be a significant expense.

Unfortunately, calculating all those `sqrt`s to intersect rays with curved surfaces does increase the runtime, as shown in Fig. 6.11 and Fig. 6.12. It is worth noting that Cartesian ray-tracing has been heavily optimized over several years. The trend shown is more important than the specific numbers (which are of course specific to the machine we're running on). As the resolution increases, the runtime of the native-spherical ray-marching increases faster than the runtime of the Cartesian version, but as we're about to see in Fig. 6.14, this is paid for with increasing accuracy.

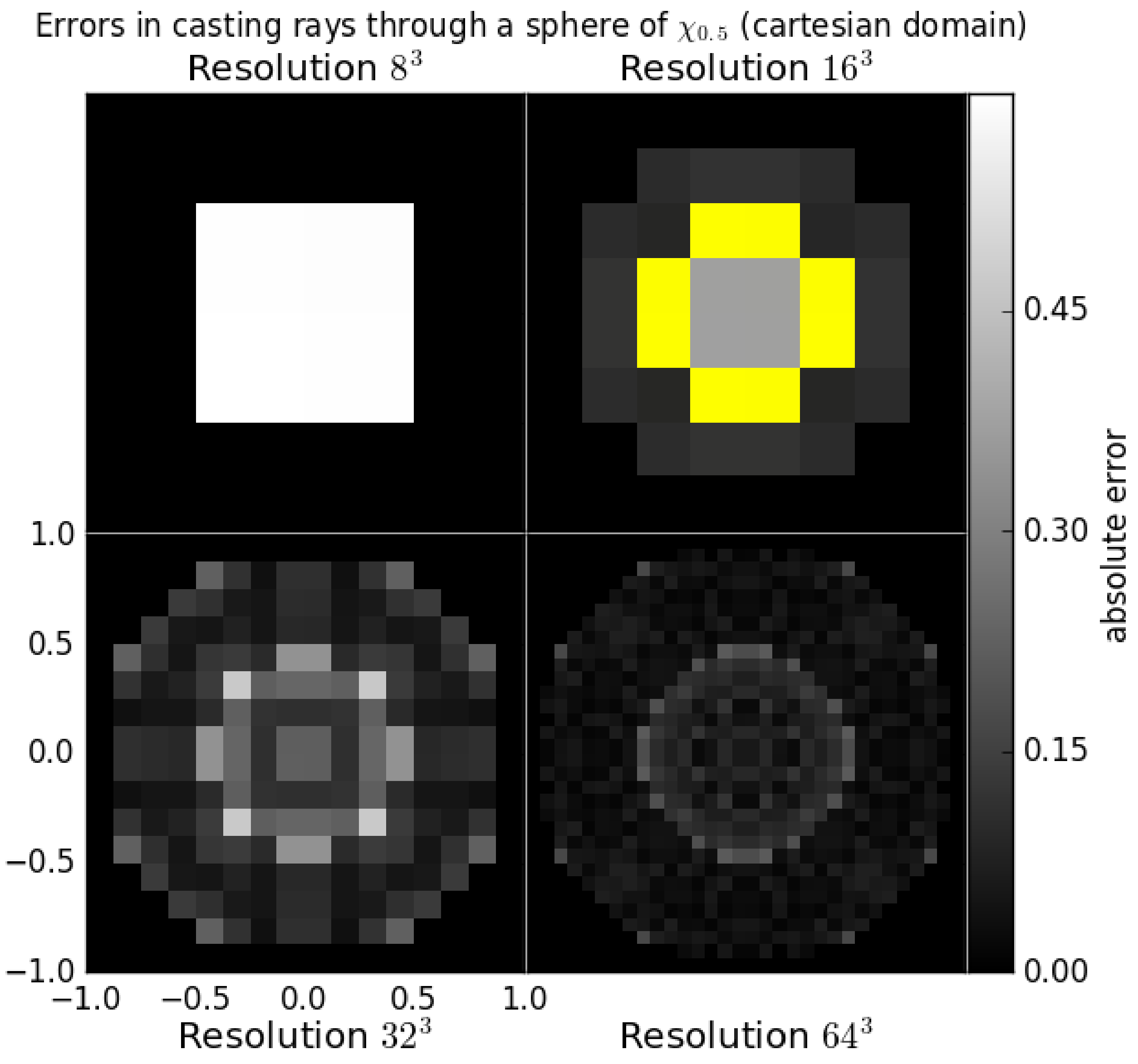

Errors in casting rays through a sphere of $\chi_{0.5}$ (cartesian domain)
Resolution 8^3
Resolution 16^3
Resolution 32^3
Resolution 64^3
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
0.45
0.30
0.15
0.00
absolute error

Figure 6.3

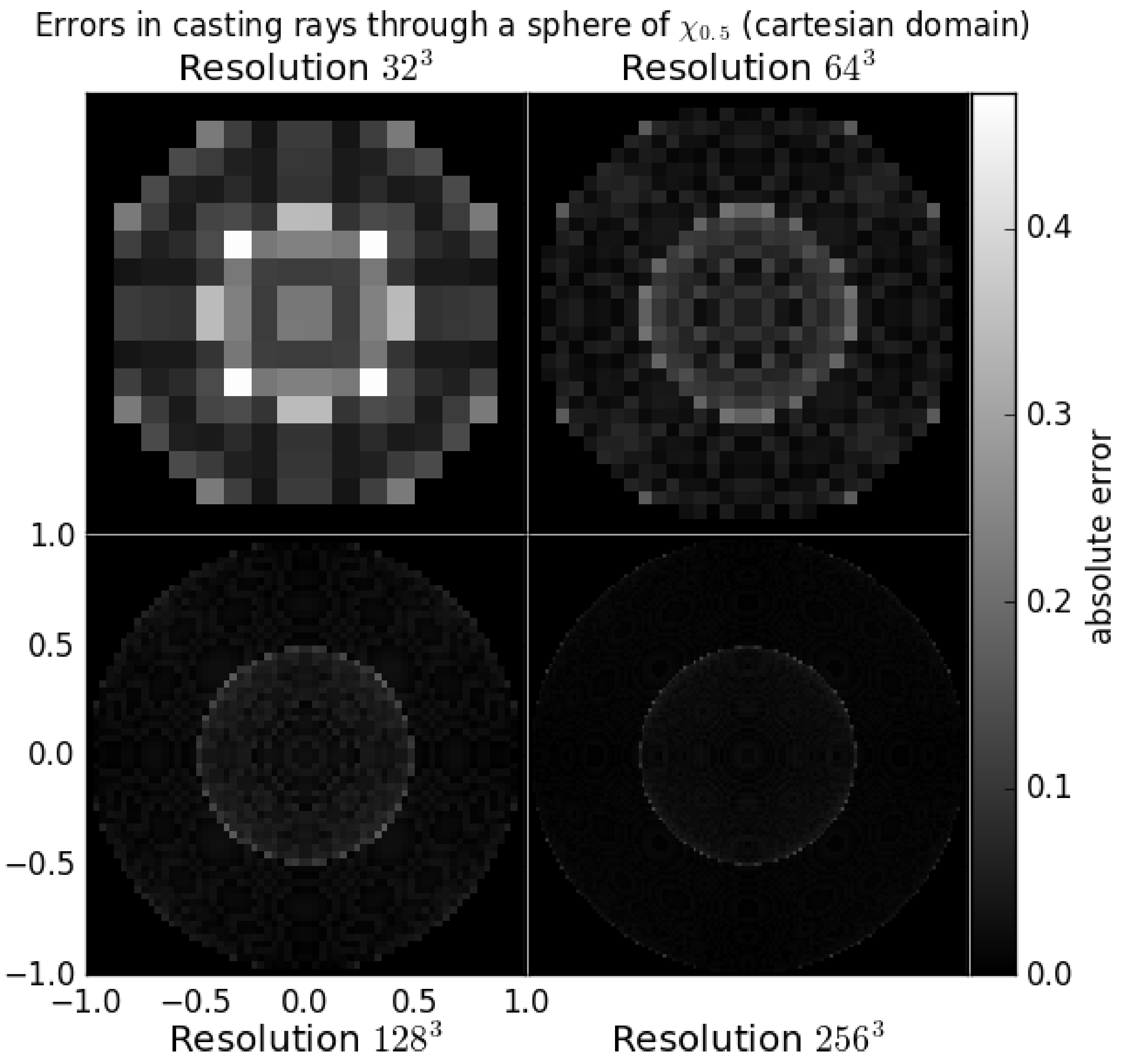


Figure 6.4: High-resolution Cartesian.

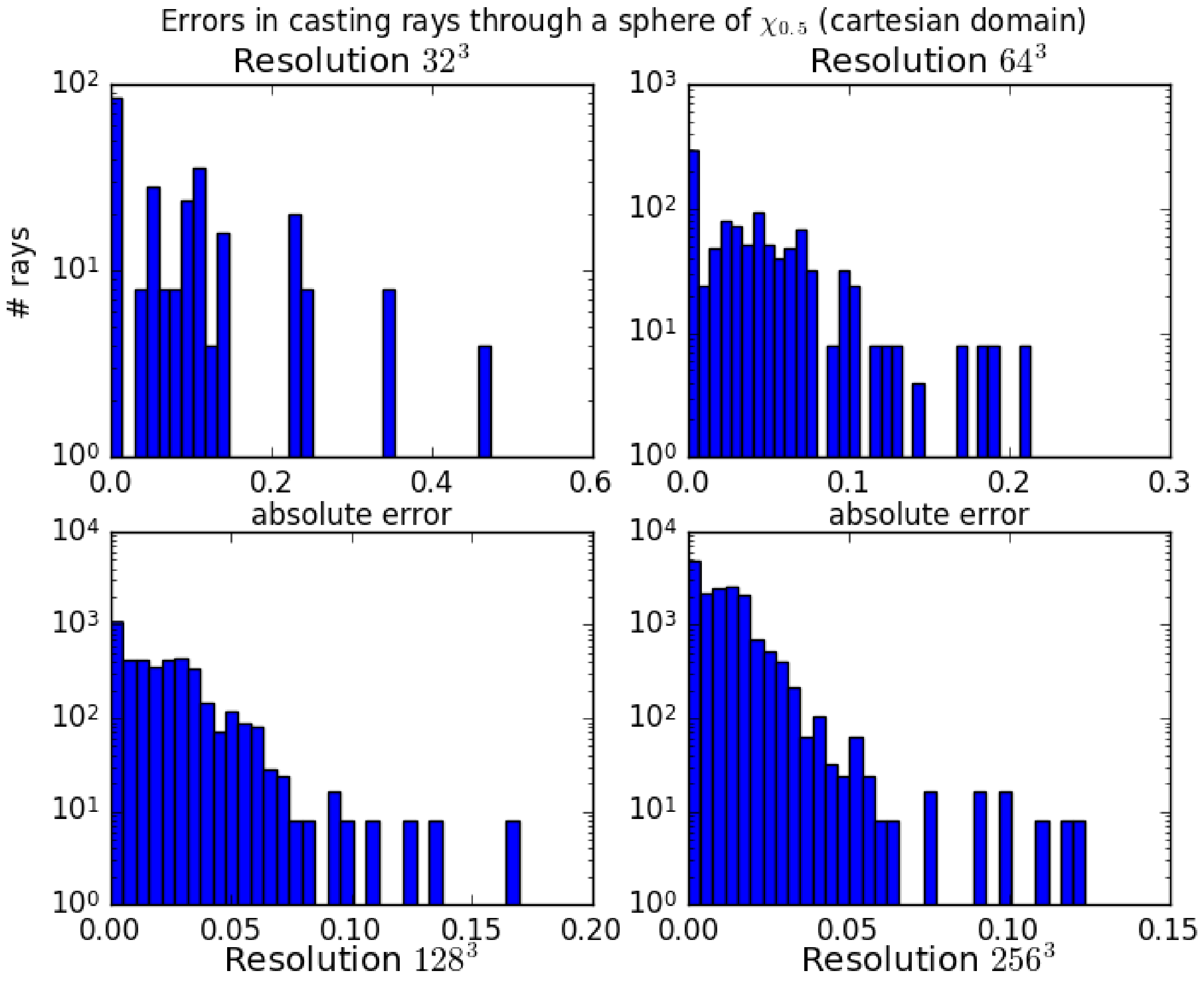

Errors in casting rays through a sphere of $\chi_{0.5}$ (cartesian domain)
Resolution 32^3
Resolution 64^3
Resolution 128^3
Resolution 256^3
# rays
absolute error
absolute error
10^0
10^1
10^2
10^3
10^4
0.0
0.2
0.4
0.6
0.1
0.3
0.00
0.05
0.10
0.15
0.20

Figure 6.5

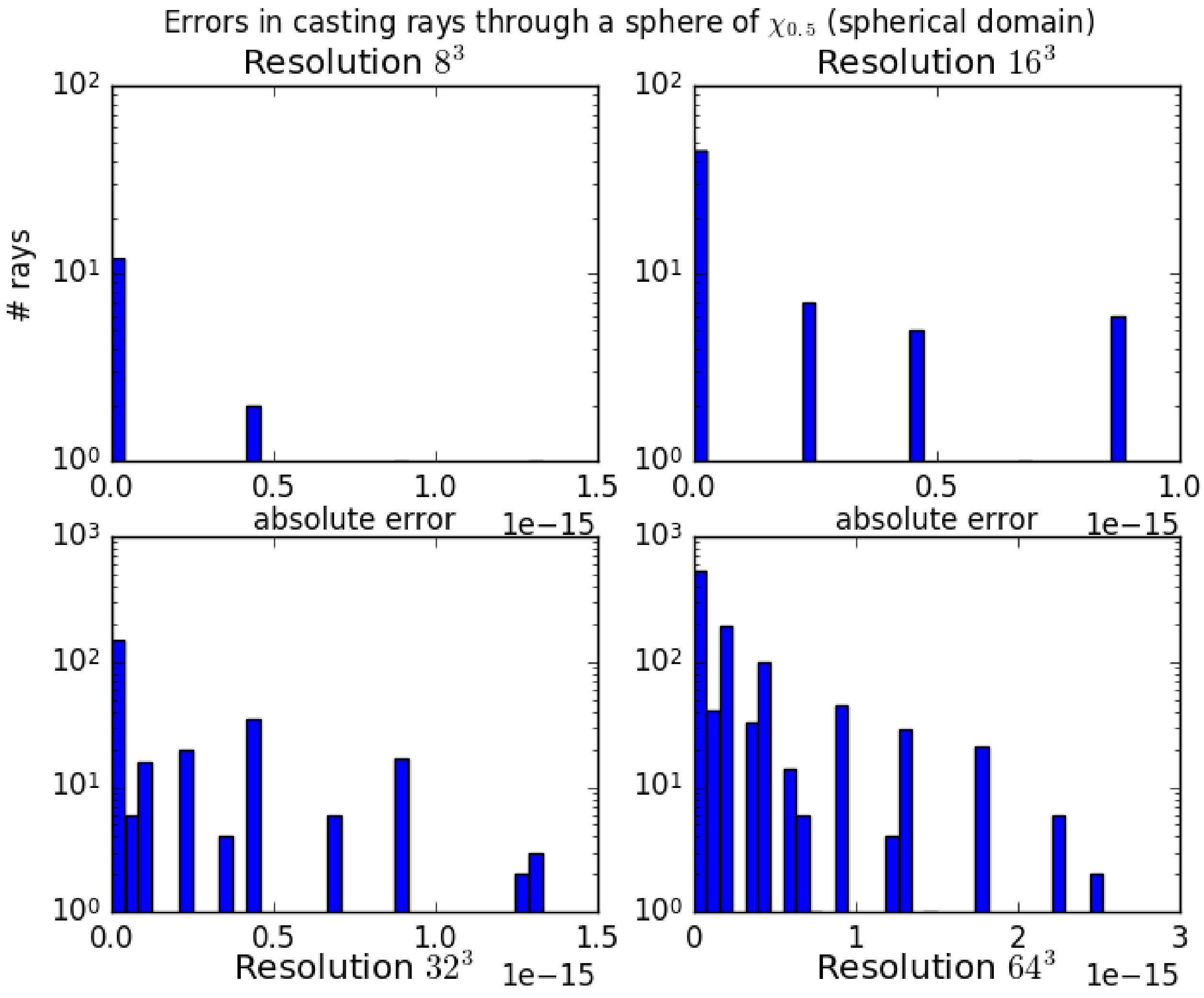

Errors in casting rays through a sphere of $\chi_{0.5}$ (spherical domain)
Resolution 8^3
Resolution 16^3
Resolution 32^3
Resolution 64^3
# rays
absolute error
absolute error
1e−15
1e−15
1e−15
1e−15
10^0
10^1
10^2
10^3
0.0
0.5
1.0
1.5
0
1
2
3

Figure 6.6

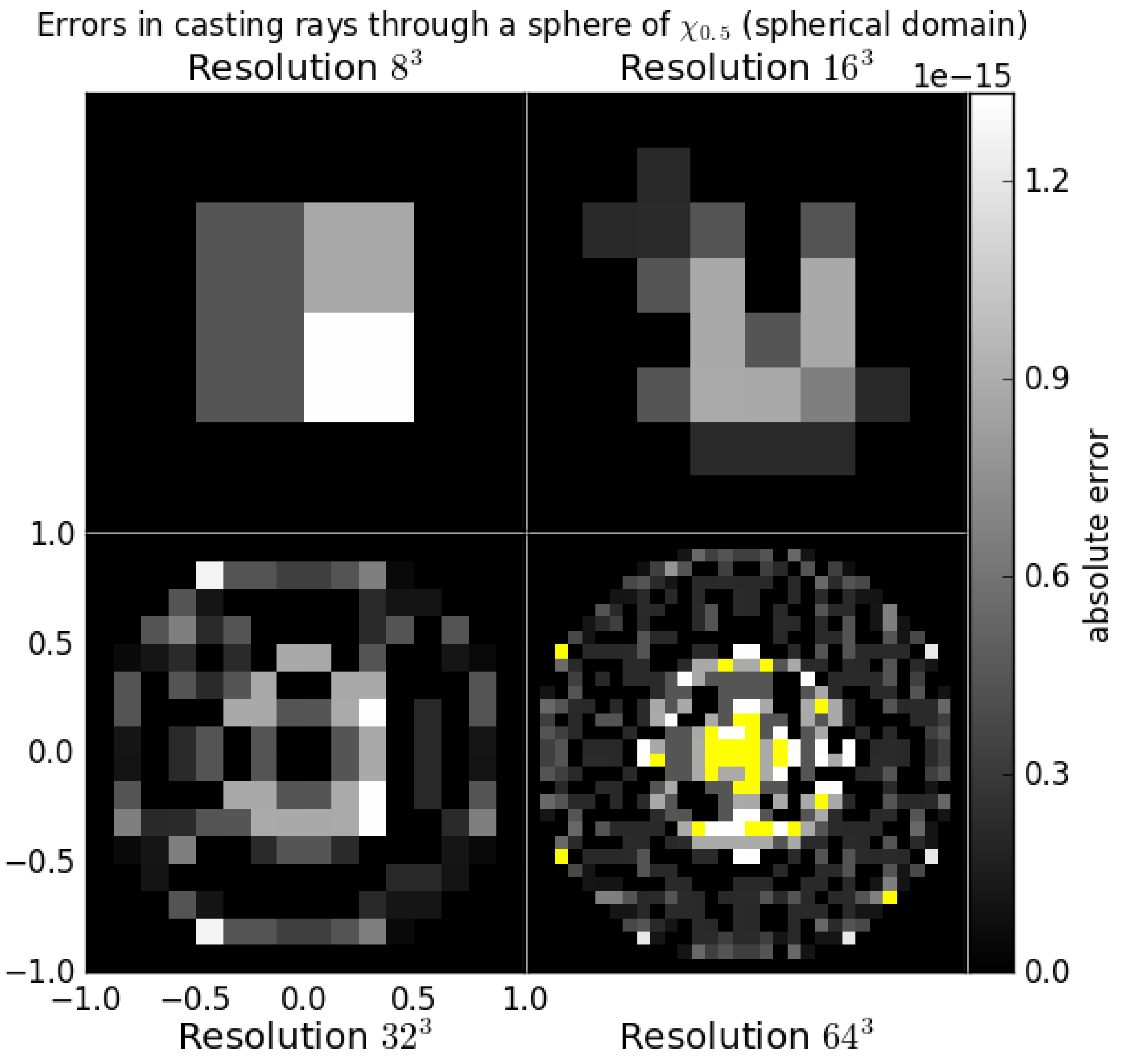
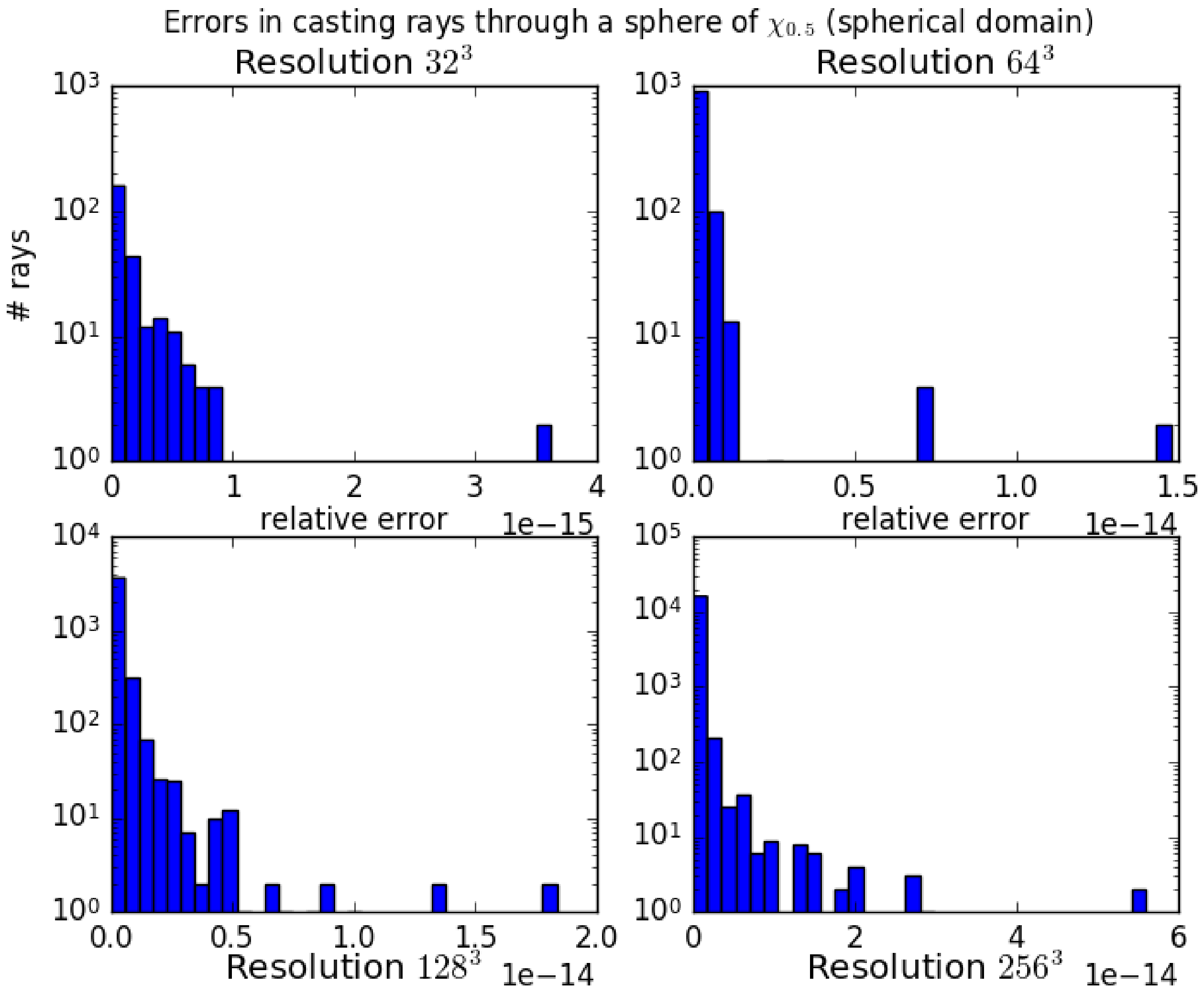

Errors in casting rays through a sphere of $\chi_{0.5}$ (spherical domain)
Resolution 8^3
Resolution 16^3
Resolution 32^3
Resolution 64^3
1e−15
1.2
0.9
0.6
0.3
0.0
absolute error
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

Figure 6.7



Errors in casting rays through a sphere of $\chi_{0.5}$ (spherical domain)
Resolution 32^3
Resolution 64^3
Resolution 128^3
Resolution 256^3
# rays
relative error
relative error
1e−15
1e−14
1e−14
1e−14

Figure 6.8

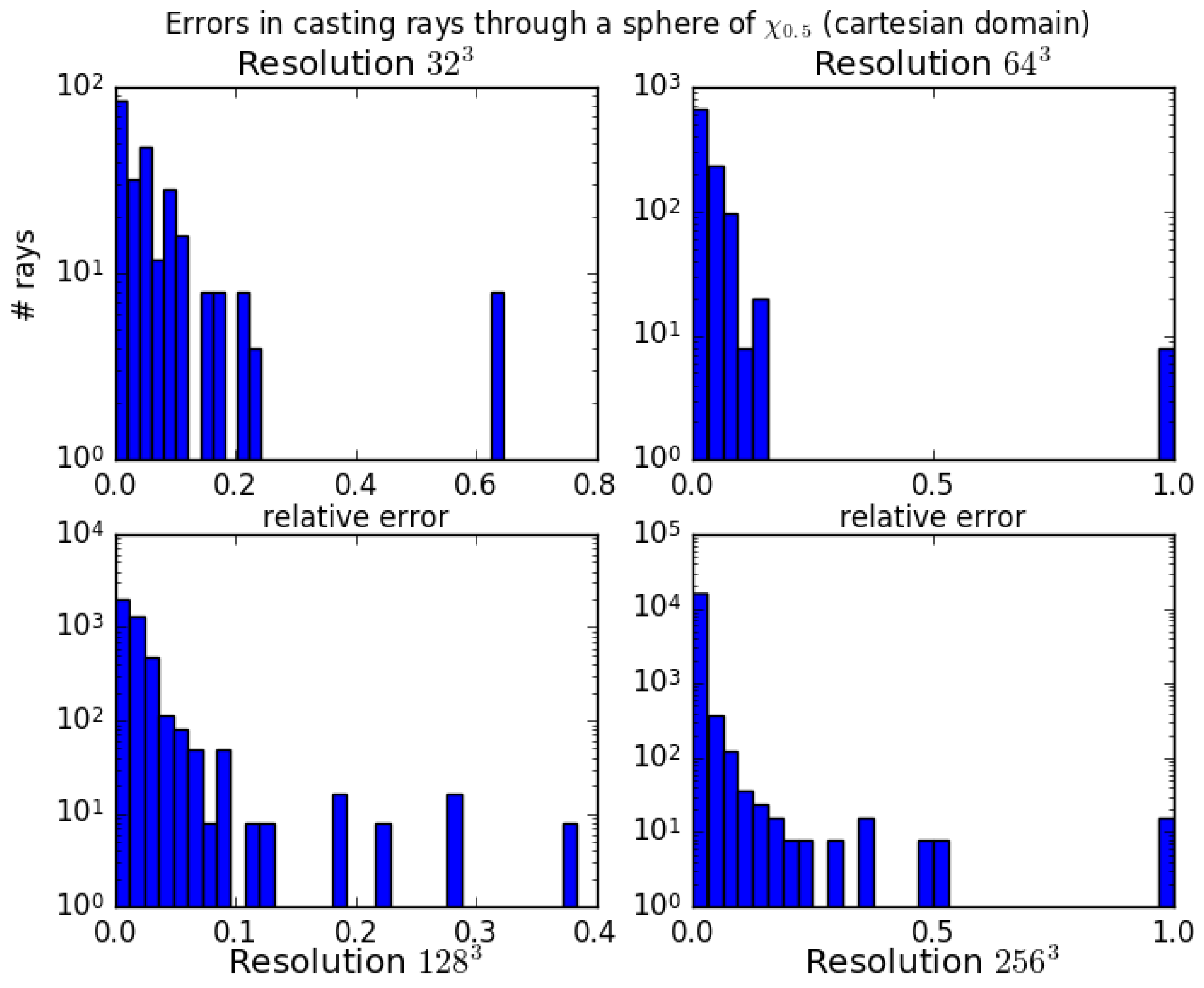


Figure 6.9: Cartesian relative errors; the very large relative errors are for the smallest true values, where the ray intersects the spherical domain at a shallow angle.

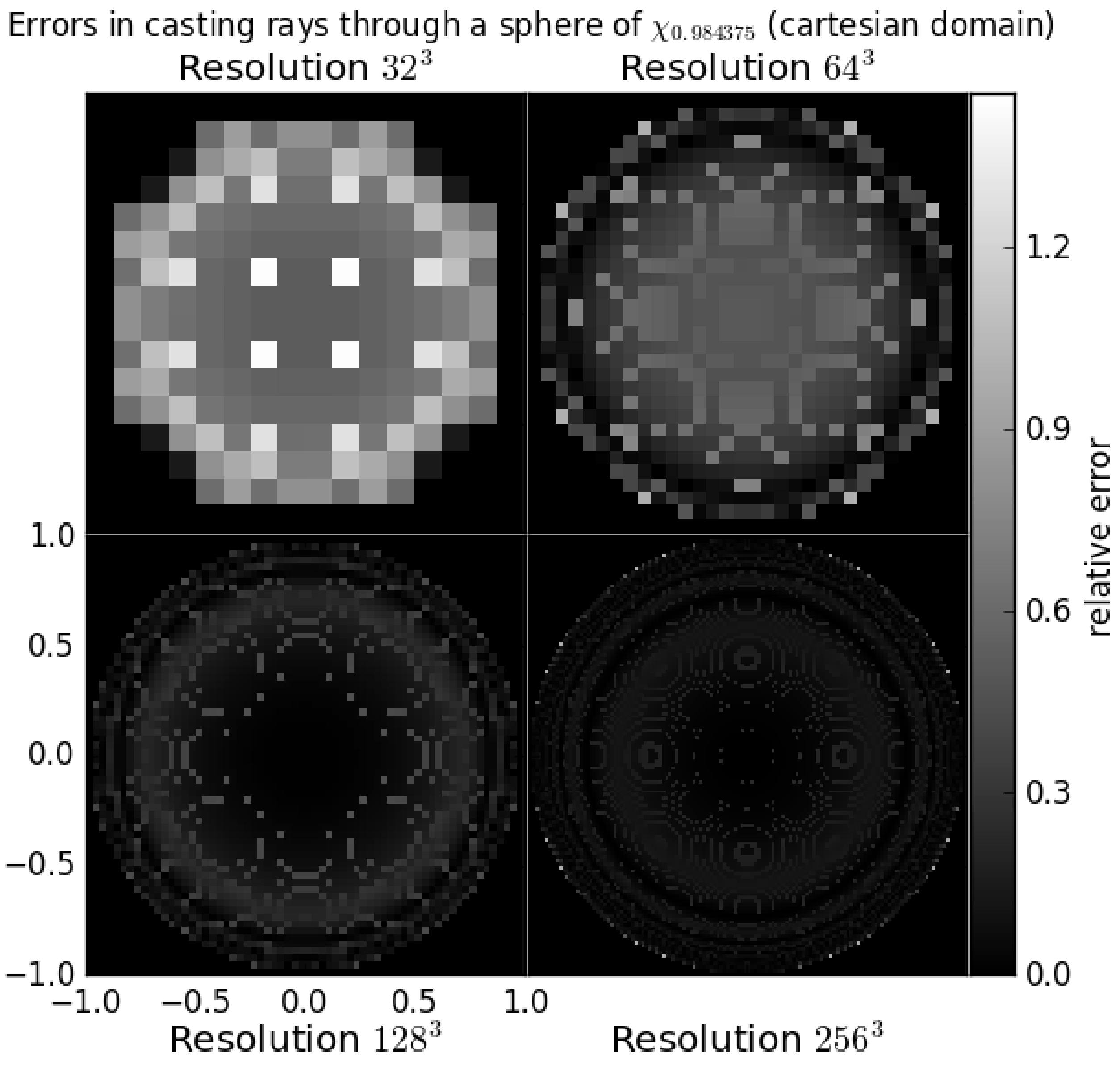


Figure 6.10: Here the field is 1 except for the top 1/64th of the sphere, where it is 64. Thus the true field values are actually larger at the edges (because the ray spends a little bit more time in the upper crust), so we don't get very large relative errors at the edges that swamp everything else. Using the native-spherical algorithm at these resolutions, the errors are too small to measure.

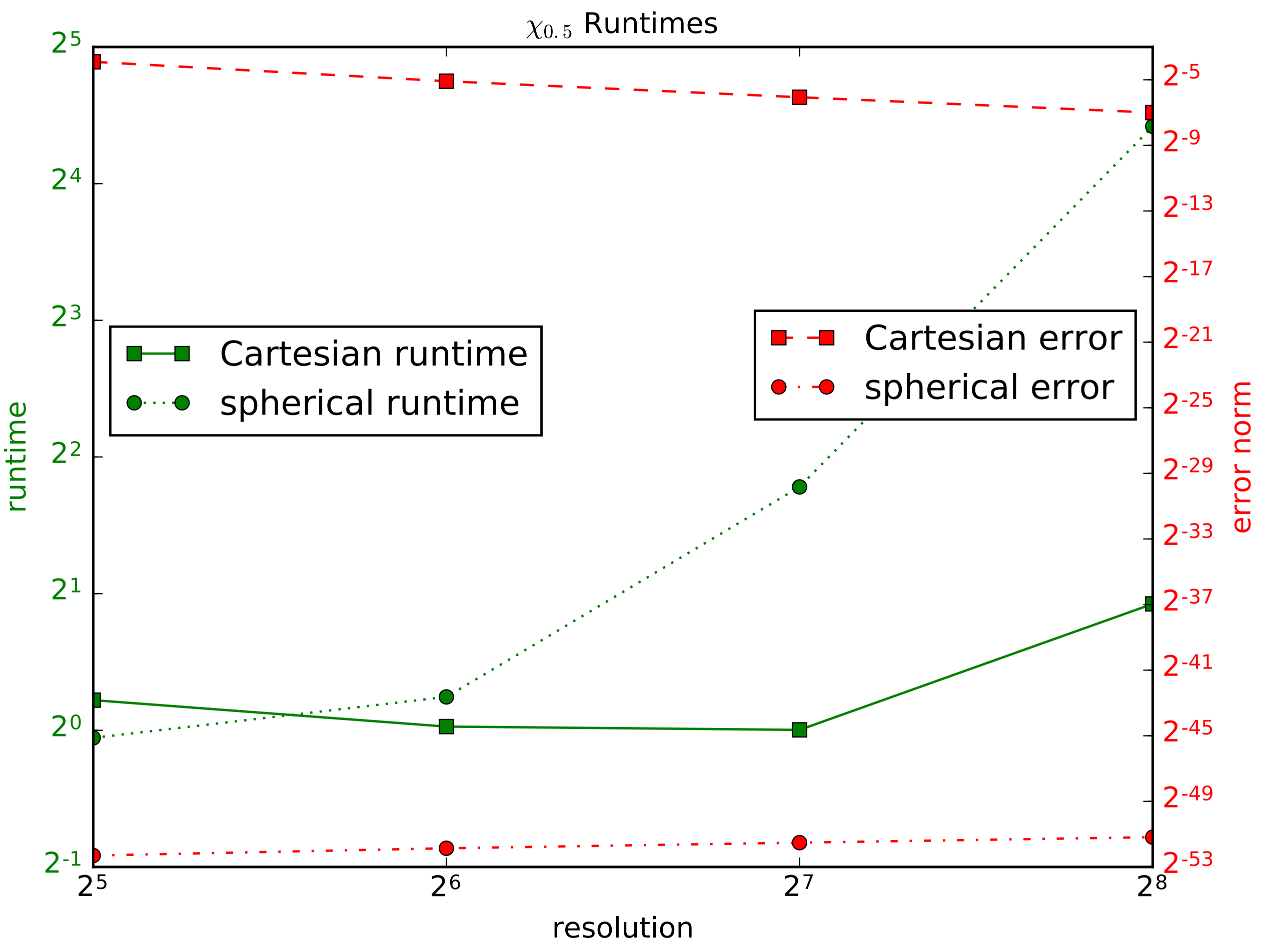


Figure 6.11: Runtimes for step function.

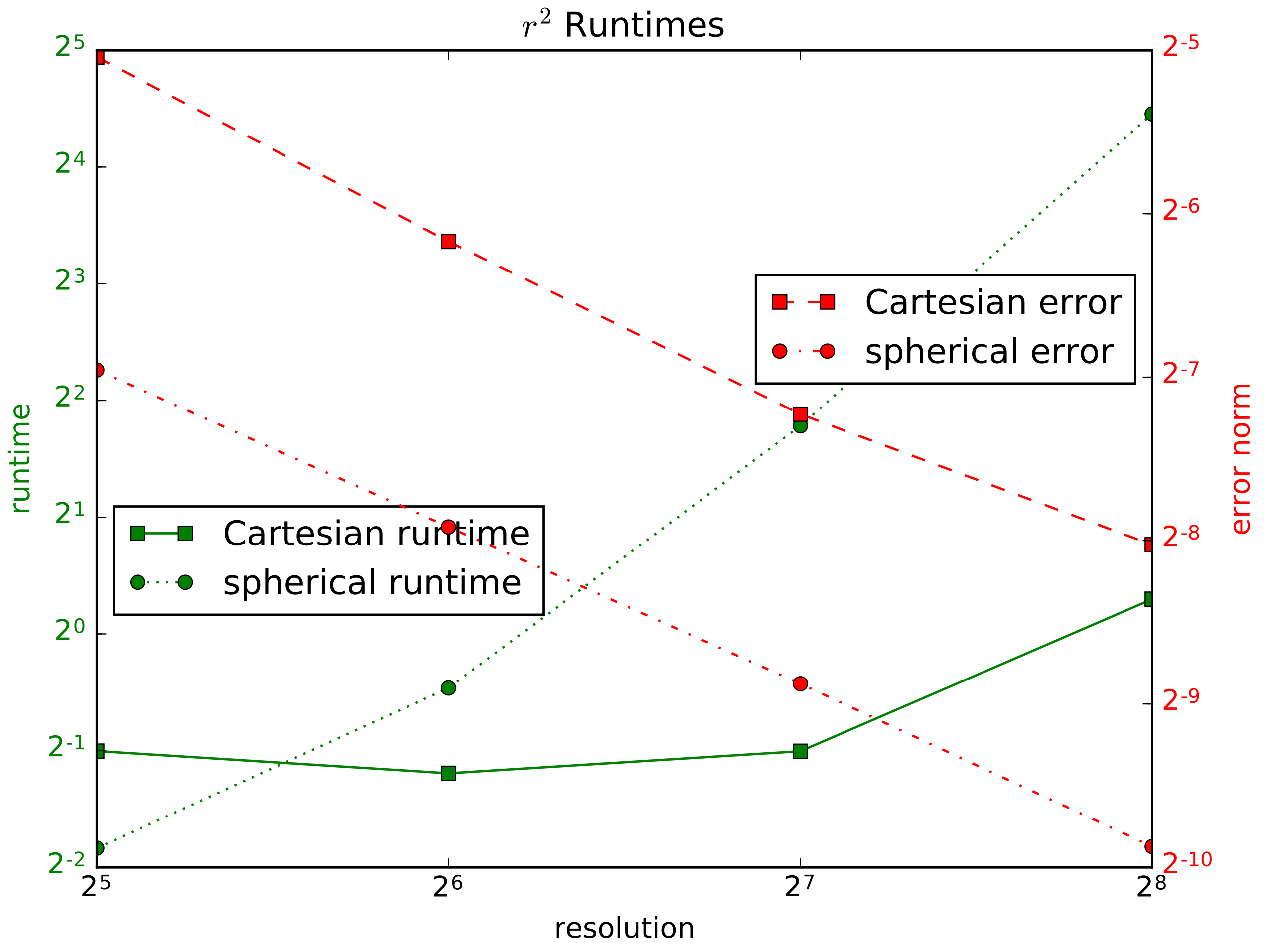


Figure 6.12: Runtimes for increasing radius function.

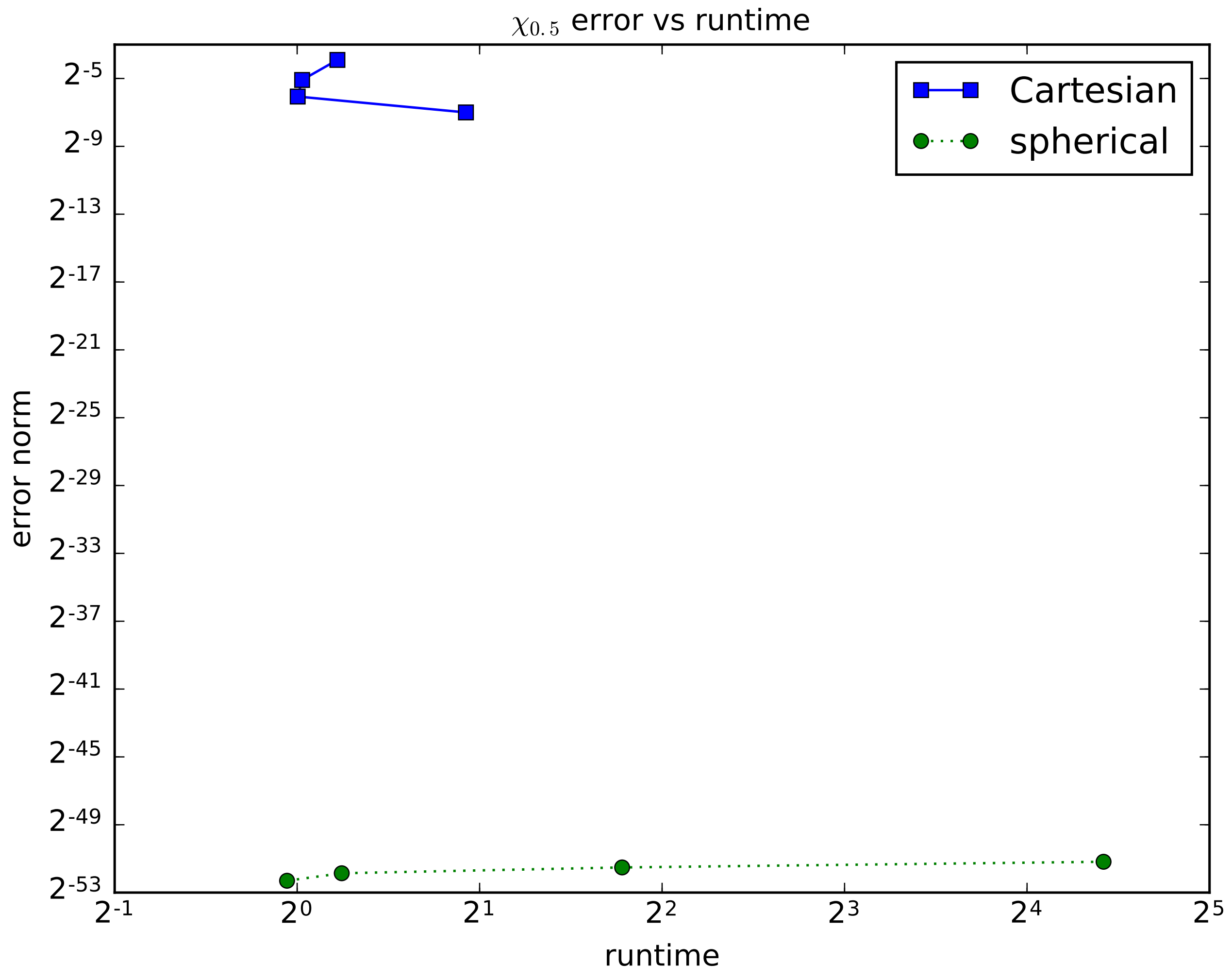


Figure 6.13: The norm of the error as a function of runtime.

We can also plot the runtime against the error. Obviously the resolution of the grid is determined by the simulation before it comes time to visualize, but of course we *can* make the visualization run faster by discarding some of our data. (But we must be careful that we don't discard the only interesting part of the data!) We sometimes accelerated the virtual-ultrasound of the Earth discussed in Chapter 7 by throwing away some of the layers.

Fig. 6.13 shows error versus runtime. As you can see, the native-spherical algorithm takes longer for a given resolution, but we can afford to use a lower resolution. (At least, when we can choose the resolution by evenly distributing the loss of data somehow so that we don't lose all the interesting features.) In our testing, our resolutions are powers of 2, so lowering the resolution is easily done by discarding 7/8 of the data, half in each coordinate. A continuously varying function such as r^2 has higher initial error, so more room for the error to decrease, as shown in Fig. 6.14.

Remark. When the field is r^2, a ray with specified y and z and x going from one edge of the domain to the other yields an integral of $\frac{2}{3}\sqrt{\max\{0, R^2 - y^2 - z^2\}}\left(R^2 + 2y^2 + 2z^2\right)$.

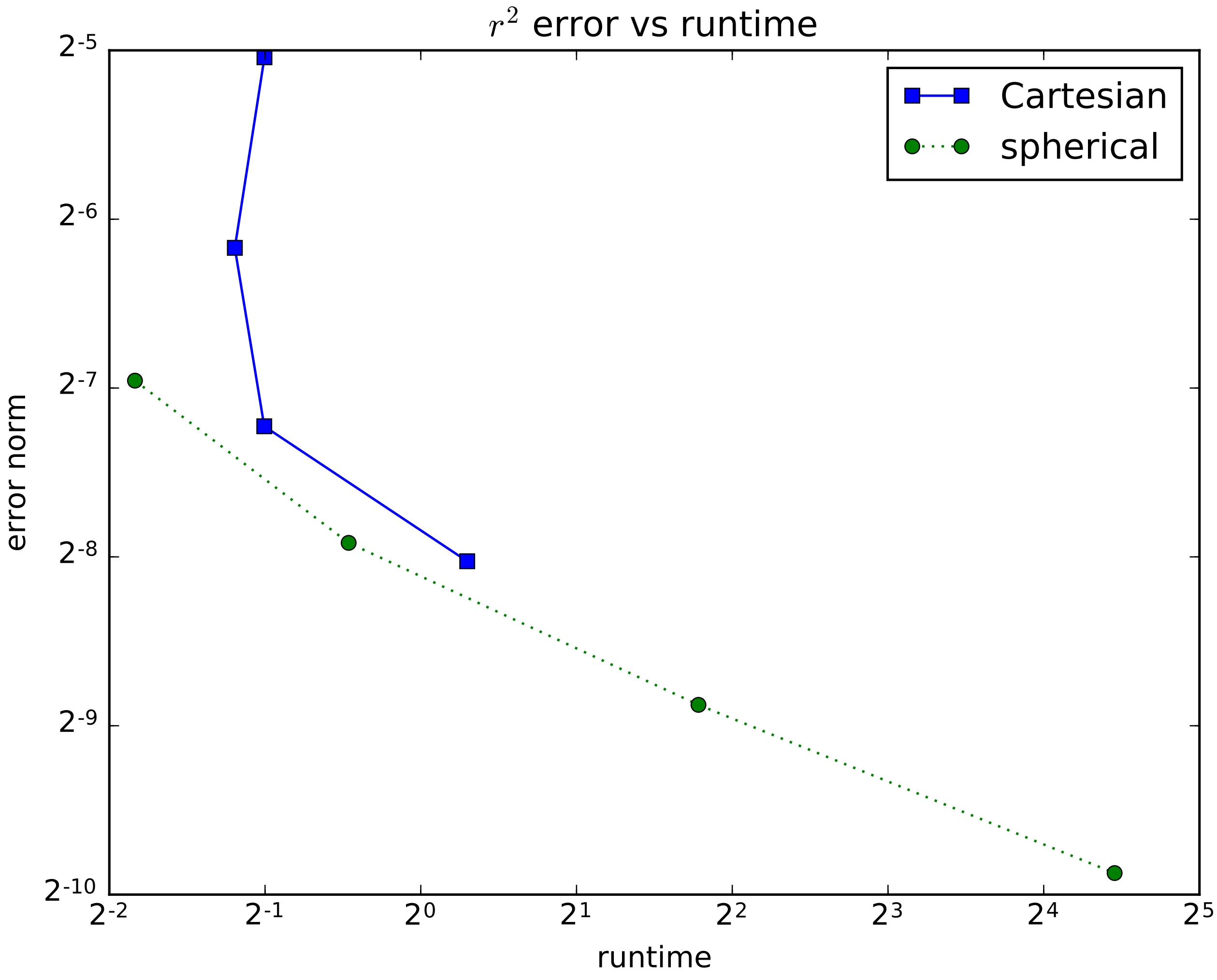


Figure 6.14: The norm of the error as a function of runtime.

Proof. When $y^2+z^2 \leq R^2$, $\int_{-1}^{1} (x^2+y^2+z^2) f(\vec{v})\,\mathrm{dx}$ where $f(\vec{v}) = \begin{cases} 1 & \text{if } r \leq R \\ 0 & \text{if } r > R \end{cases}$ is equal to $\int_{-\sqrt{R^2-y^2-z^2}}^{\sqrt{R^2-y^2-z^2}} (x^2 + y^2 + z^2)\,\mathrm{dx} = \int_{-\sqrt{R^2-y^2-z^2}}^{\sqrt{R^2-y^2-z^2}} x^2\,\mathrm{dx} + 2\sqrt{R^2-y^2-z^2}(y^2+z^2)$.

$$\int_{-\sqrt{R^2-y^2-z^2}}^{\sqrt{R^2-y^2-z^2}} x^2\,\mathrm{dx} = \left[\frac{x^3}{3}\right]_{-\sqrt{R^2-y^2-z^2}}^{\sqrt{R^2-y^2-z^2}} = \frac{2}{3}(R^2-y^2-z^2)^{3/2}$$

$$\frac{2}{3}(R^2-y^2-z^2)^{3/2} + 2\sqrt{R^2-y^2-z^2}(y^2+z^2) = 2\sqrt{R^2-y^2-z^2}\left(\frac{R^2-y^2-z^2}{3} + y^2 + z^2\right)$$
$$= 2\sqrt{R^2-y^2-z^2}\left(\frac{R^2}{3} + \frac{2}{3}y^2 + \frac{2}{3}z^2\right) = \frac{2}{3}\sqrt{R^2-y^2-z^2}\left(R^2+2y^2+2z^2\right)$$

□

Conclusion Our new method performs well on our artificial test cases. Intersecting rays with nonplanar facets does cost us, but the improvement in accuracy is significant enough that we can discard some of the data — 'resampling' to a reduced spherical grid rather than a Cartesian grid — and still come out ahead. It's difficult to be sure how representative our artificial test cases are of real-world data, but these results are promising enough that we've begun to apply our method in practice, as described in Chapter 7.

Chapter 7

Broader impacts

Our initial application is to geophysics, but astrophysics provides other problems such as stellar atmospheres and accretion disks. Other applications are possible, of course. Most, if not all, ultrasound machines produce data in polar (2D scanner) or spherical (3D scanner) coordinates, with the result that there is a chronic need for conversion. [Wil05] CT and MRI scans sometimes use Cartesian coordinates, but vary enough that volume-rendering by ray-casting must account for multiple coordinate systems.[KA15]

The software package yt serves as a front-end to a wide variety of scientific simulation software. yt put out a call for algorithms for traversing cylindrical and spherical coordinates.[Tur13]

Previously, yt was able to render time-dependent seismic wavefields found in SPECFEM[PKL$^+$11] simulations. SPECFEM is a tool for seismic simulation. Tomography of the interior of the planet is one of the central activities of geophysics.[Tur16]

Martin Pratt provided a composite data set of sound-wave propagation speed (which generally corresponds to density) that covered *almost* the entire Earth (latitude from -88 to $+88$) down to a depth of 2,890 kilometers. Volume-rendering this data set provides the effect of taking an ultrasound of the entire Earth.

`yt`'s existing volume-rendering infrastructure works by solving the radiative transfer equations. Each data point has an opacity (light-blocking) and an emissivity (light-generation); these are given by the `TransferFunction` which defines opacity and emissivity for a range of field values.[Gol] A sampler function samples the field value(s) in a cell (or interpolates from the vertices of the cell), blocks some light and generates some new light. That's why we need to march along a ray and call the sampler function on each cell.

Because small outlier regions will be washed out as we march along the ray (unless such an outlier region happens to fall at the rim, so a ray passes *only* through there), it is usually harmless to discard very high and very low field values.

Fig. 7.1 shows an example of how the `TransferFunction` maps field values to emissivity. This results in the volume-rendering shown in Fig. 7.2.

7.1 Mesh representation

Previously, `yt` always stored grids in AMR compressed form: the domain is broken into subdomains wherein the grid is evenly spaced, with only the boundaries of the subdomains and their level of refinement stored. This saves on memory, and in theory it's perfectly applicable to spherical grids as well.

This works well when we have a large sparse grid with a much smaller dense box — as when there is some critical region where more precision is needed. The TX data, however, doesn't look like that. It doesn't have isolated patches of higher resolution — rather, it has three arrays for depth, latitude and longitude, with the vertices being the three-dimensional product of those three arrays. This *could* be forced into AMR form, but

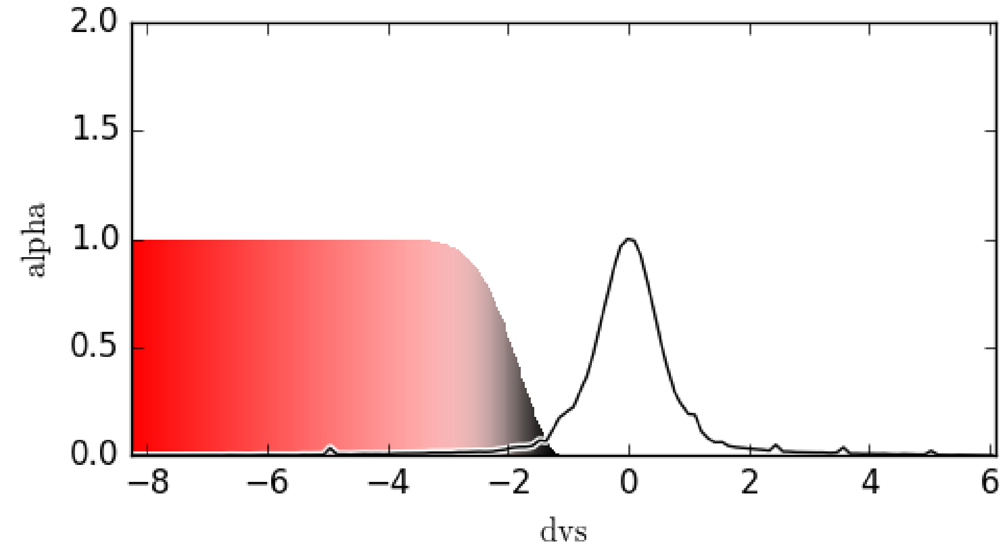


Figure 7.1: Transfer function.

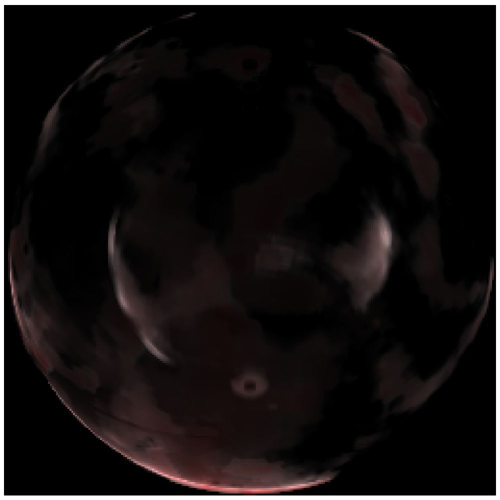


Figure 7.2: Volume rendering.

the depth doesn't have very long runs of evenly-spaced numbers, so each subdomain would only have a few layers in it; practically all of the calculations would be happening across subdomain boundaries. And the overhead of AMR representation isn't zero: each ray must be separately intersected with each subdomain.

With this algorithm, there's no actual benefit to the compressed representation of the grid anyway, because we work primarily with quantities derived from the coordinates rather than coordinates directly, and if the coordinates were evenly spaced, the derived quantities wouldn't be. However, we still might want to have a patch of much-higher resolution localized to one area — for example, make the depth measurements much denser near the particular latitude and longitude of an earthquake. Therefore, in the implementation, we don't close the door on the AMR infrastructure; rather, we slot in a larger object (the spherical grid with all $r^2, (\operatorname{sign}\cos\theta)\cos^2\theta, \cos\phi, \sin\phi$ arrays stored) in place of the usual fixed-size subdomain object.

7.2 Future applications

The NCSA Advanced Visualization Laboratory is often asked to visualize astrophysical data such as Yuhong Fan's. Kalina Borkiewicz wrote a C++ tool that currently resamples spherical data to Cartesian; it might be possible to slot in this new algorithm instead, dropping some data so that the ray-casting still runs just as fast as it used to.

Mark van Moer has a hand-tuned solution for Scott Noble's black hole simulations that doesn't resample, but instead keeps the vertices the same while treating the curved surfaces of the spherical-grid cells instead as flat planes defined by the four vertices of a facet. We have not yet been able to replicate this feat for general datasets; if we can, it would be interesting to directly compare the two approaches.

7.3 Future work

We do not always need to calculate the *exact* t-length of the intersection of each ray with each cell; with a very fine grid, this can be expensive, prodding us to use an approximation instead.

If we leave a cell through an azimuth-facet, then it is trivial to calculate the 'exact' (up to rounding) t-value at which we leave, but if we leave a cell through a zenith-facet or a radius-facet, the t-value requires a few extra computations including an FSQRT. If profiling reveals that this takes up significant time, we could potentially use the geometry of a cell to arrive at an approximate length-of-intersection more quickly.

Depending on the exact sampler function we use, it may be important that the *total* length across all cells is accurate, or it may be more important that the *fraction* of the total length that each cell takes up is accurate.

It would be nice to have some $f(r, \theta, \phi)$ such that we can bound $\dfrac{\partial f}{\partial t}$ at each step, so that we can approximate Δt by some multiple of Δf. ($\dfrac{\partial}{\partial t} \tan\phi$ depends only on $x^2 \ldots$)

7.3.1 Randomize

Las Vegas algorithm More radically, we could simply decide on a division of t, such as 0.01 for a total of 100 hops, and for each t-step, calculate the pseudospherical coordinates, find the appropriate cell, and sample the value there for a length equal to the length of the t-step. That would introduce many errors, but a Las Vegas algorithm is possible: When we happen to hop across the boundary of a cell, we can use simple binary search to locate the exact boundary-crossing.

7.4 Closing remarks

We have provided a new algorithm for casting rays through non-Cartesian domains for volume-rendering. This method is intentionally agnostic to how the data is interpolated, wrapping that in an abstraction we call the sampler function. There is a great deal of promising work on customizing interpolating functions to the expected shape of the underlying data, such as [BUS+15] and [NKH11], and best results will likely be obtained by combining this algorithm with the appropriate interpolation. Together, we can honor our promise that looking at the data won't mislead scientists.

References

[BUS$^+$15] T. Bolemann, M. Üffinger, F. Sadlo, T. Ertl, and C.-D. Munz, *Direct visualization of piecewise polynomial data*, Notes on Numerical Fluid Mechanics and Multidisciplinary Design **128** (2015), 535–550.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell, *The psychology of human-computer interaction*, L. Erlbaum Associates, 1983.

[FH16] F. G. Fuchs and J. M. Hjelmervik, *Interactive isogeometric volume visualization with pixel-accurate geometry*, IEEE Transactions on Visualization and Computer Graphics **22** (2016), no. 2, 1102–1114.

[Gol] Nathan Goldbaum, *3d visualization and volume rendering.*

[Har16] John C. Hart, *Clipping*, Prof. Hart is under contract with Pearson for a new introductory computer graphics textbook., 2 2016.

[Hew12] Russell J. Hewett, *Numerical methods for solar tomography in the stereo era*, Ph.D. thesis, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/29562, 02 2012, Finally, we develop a scalable algorithm for ray tracing dense meshes.

[Hew16] Russell J. Hewett, personal correspondence, 2016.

[Jon99] Philip W. Jones, *First- and second-order conservative remapping schemes for grids in spherical coordinates*, Monthly Weather Review **127** (1999), no. 9, 2204–2210.

[KA15] P. Kumar and A. Agrawal, *Hardware accelerated multi-coordinate viewing framework for volumetric visualization of large 3d medical dataset*, vol. 54, 2015, pp. 566–573.

[LCTD14] X. Li, W.-F. Chen, Y.-B. Tao, and Z. Ding, *Efficient quadratic reconstruction and visualization of tetrahedral volume datasets*, Journal of Visualization **17** (2014), no. 3, 167–179.

[LH14] Z. Liu and J. Heer, *The effects of interactive latency on exploratory visual analysis*, IEEE Transactions on Visualization and Computer Graphics **20** (2014), no. 12, 2122–2131, cited By 15.

[MC12] F.M. Miranda and W. Celes, *Volume rendering of unstructured hexahedral meshes*, Visual Computer **28** (2012), no. 10, 1005–1014, cited By 3.

[MCZ$^+$16] H.-H. Mei, H.-D. Chen, X. Zhao, H.-N. Liu, B. Zhu, and W. Chen, *Visualization system of 3d global scale meteorological data*, Ruan Jian Xue Bao/Journal of Software **27** (2016), no. 5, 1140–1150.

[NKH11] B. Nelson, R.M. Kirby, and R. Haimes, *Gpu-based interactive cut-surface extraction from high-order finite element fields*, IEEE Transactions on Visualization and Computer Graphics **17** (2011), no. 12, 1801–1811.

[NKH14] ______, *Gpu-based volume visualization from high-order finite element fields*, IEEE Transactions on Visualization and Computer Graphics **20** (2014), no. 1, 70–83.

[NLKH12] B. Nelson, E. Liu, R.M. Kirby, and R. Haimes, *Elvis: A system for the accurate and interactive visualization of high-order finite element solutions*, IEEE Transactions on Visualization and Computer Graphics **18** (2012), no. 12, 2325–2334.

[NNFJ13] Alexandre S. Nery, Nadia Nedjah, Felipe M.G. Frana, and Lech Jwiak, *Parallel processing of intersections for ray-tracing in application-specific processors and {GPGPUs}*, Microprocessors and Microsystems **37** (2013), no. 6–7, 739–749.

[NvS$^+$14] T. Nissen-Meyer, M. van Driel, S. C. Stähler, K. Hosseini, S. Hempel, L. Auer, A. Colombi, and A. Fournier, *Axisem: broadband 3-d seismic wavefields in axisymmetric media*, Solid Earth **5** (2014), 425–445.

[PBH12] Kyle Parfrey, Andrei M. Beloborodov, and Lam Hui, *Introducing phaedra: a new spectral code for simulations of relativistic magnetospheres*, Monthly Notices of the Royal Astronomical Society **423** (2012), 1416–1436.

[PKL$^+$11] Daniel Peter, Dimitri Komatitsch, Yang Luo, Roland Martin, Nicolas Le Goff, Emanuele Casarotti, Pieyre Le Loher, Federica Magnoni, Qinya Liu, Cline Blitz, Tarje Nissen-Meyer, Piero Basini, and Jeroen Tromp, *Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes*, Geophysical Journal International **186** (2011), no. 2, 721–739.

[SF14] A. Schollmeyer and B. Froehlich, *Direct isosurface ray casting of nurbs-based isogeometric analysis*, IEEE Transactions on Visualization and Computer Graphics **20** (2014), no. 9, 1227–1240.

[TSO$^+$11] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman, *yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data*, The Astrophysical Journal Supplement **192** (2011), 9.

[Tuf01] Edward R. Tufte, *The visual display of quantitative information*, 2 ed., Graphics Press, Cheshire, Conn., 2001, The first edition of Tufte's now classic text on the design of statistical graphics was published in 1983.

[Tur13] Matthew Turk, *Volume traversal*, Tech. Report 0016, September 2013.

[Tur16] Matthew J. Turk, personal correspondence, 2016.

[UFE10] M. Üffinger, S. Frey, and T. Ertl, *Interactive high-quality visualization of higher-order finite elements*, Computer Graphics Forum **29** (2010), no. 2, 337–346.

[Wil05] P. Willis, *Method and system for registering ultrasound image in three-dimensional coordinate system*, May 2005, US Patent 6,896,657.

[Wor08] Paul Wormer, *spherical polar coordinates*, 2008.