Walk on Stars
(Implementation Guide)

Rohan Sawhney, Bailey Miller, Ioannis Gkioulekas, Keenan Crane

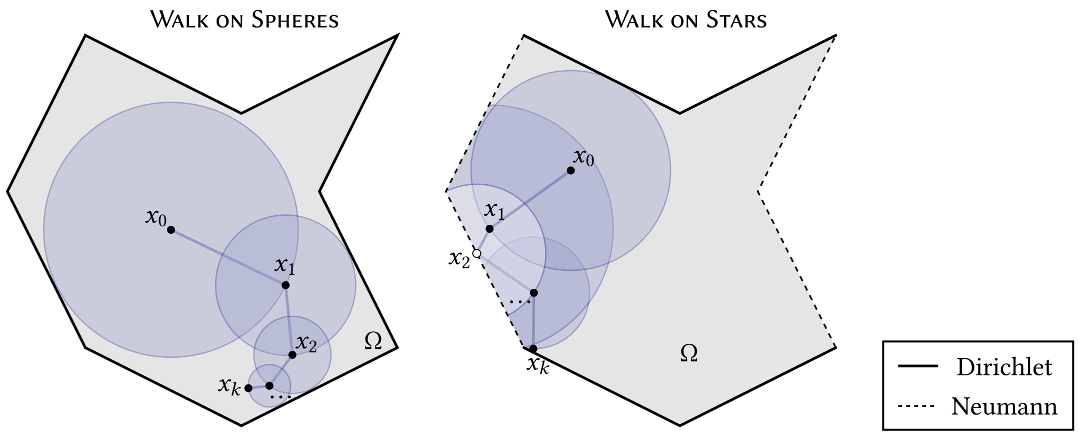

Figure 1: Walks taken by the WoS algorithm *(left)* and the WoSt algorithm *(right)*. The algorithms are virtually identical, except that WoSt replaces spheres with star-shaped regions, enabling walks to reflect off Neumann boundaries.

This note provides a step-by-step tutorial on how to implement the *walk on stars (WoSt)* algorithm of Sawhney et al. [2023]. Though the derivation of algorithm takes some work, the final algorithm remains about as simple as the original *walk on spheres (WoS)* method of Muller [1956]. For a Laplace equation with Dirichlet boundary conditions, WoS essentially says:

(Walk on Spheres) *Repeatedly jump to a random point on a **sphere** around the current point until you reach the boundary. The average boundary value over many such walks yields the solution.*

For a Laplace equation with Dirichlet and zero-Neumann boundary conditions, WoSt is nearly identical:

(Walk on Stars) *Repeatedly jump to a random point on a **star-shaped region** around the current point until you reach the **Dirichlet** boundary. The average boundary value over many such walks yields the solution.*

The main implementation effort is in enriching the closest point queries used by WoS with *closest silhouette point queries*, as well as standard "first hit" ray intersection queries. A complete 2D implementation with no external dependencies is provided in about 150 lines of C++ code (ignoring I/O), in the file `WoStLaplace.cpp`. For simplicity, we omit nonzero Neumann conditions and acceleration of geometric queries—full 3D implementation is detailed in Sawhney et al. [2023].

To emphasize the core ideas of the WoSt algorithm, this tutorial focuses on a bare-bones 2D version of WoSt that solves the Laplace equation

$$\begin{array}{rcll} \Delta u &=& 0 & \text{on } \Omega, \\ u &=& g & \text{on } \partial\Omega_D, \\ \frac{\partial u}{\partial n} &=& 0 & \text{on } \partial\Omega_N, \end{array} \tag{1}$$

where $\Omega \subset \mathbb{R}^2$ is a closed polygon in the plane, g is a real-valued function on the Dirichlet part of the boundary $\partial\Omega_D$, and $\partial\Omega_N$ is the complementary Neumann part of the boundary.

1 Overview

1.1 Walk on Spheres (WoS)

A good way to understand WoSt is to start with the basic WoS algorithm, which solves the same problem but with pure Dirichlet boundary conditions ($\partial_N = \emptyset$). The basic idea is quite simple: to estimate the solution at a point $x_0 \in \Omega$, just take M random walks to the boundary $\partial\Omega$ (for some large number M), and grab the average value g at these boundary points. The average boundary value across all walks then gives an estimate of the solution, which becomes more accurate as M increases. The reason this method is called "walk on *spheres*" is that these random walks are simulated by repeatedly jumping to the boundary of the largest empty sphere $S_r(x_k) \subset \Omega$ around the current point x_k to get the next point x_{k+1}. Steps hence tend to be big, and the method quickly makes progress toward the boundary (in comparison to taking small, fixed-sized steps). In a bit more detail, the WoS algorithm looks like this:

- **for** $i = 1, \ldots, M$:
  - **until** x_k is within a small distance ε of the domain boundary $\partial\Omega$:
    - Compute the distance r to the closest point $\overline{x}_k$ on $\partial\Omega$.
    - Pick a random point x_{k+1} on the sphere $S_r(x_k)$ of radius r around x_k.
  - Add the value of $g(\overline{x}_k)$ to a running total $\widehat{u}$.
- Return the estimate $\widehat{u}/M$.

Figure 1, *left* shows an example walk. To actually implement this algorithm, we just need two kinds of subroutines:

- **Geometric queries.** For a given point $x_k \in \Omega$, we need to know how to compute the closest point $\overline{x}_k$ on the boundary.
- **Random sampling.** For a sphere S with center x_k and radius r, we need to know how to pick a point x_{k+1} on S uniformly at random.

In fact, this basic structure will be the same for any "walk on X" algorithm (known more broadly as *grid-free Monte Carlo methods*). We generally need to perform some small set of geometric queries, and sample from some small set of probability distributions. Otherwise, the logic is pretty similar: repeatedly apply these operations to simulate a random walk, and average values obtained from each walk to get the final solution estimate.

1.2 Walk on Stars (WoSt)

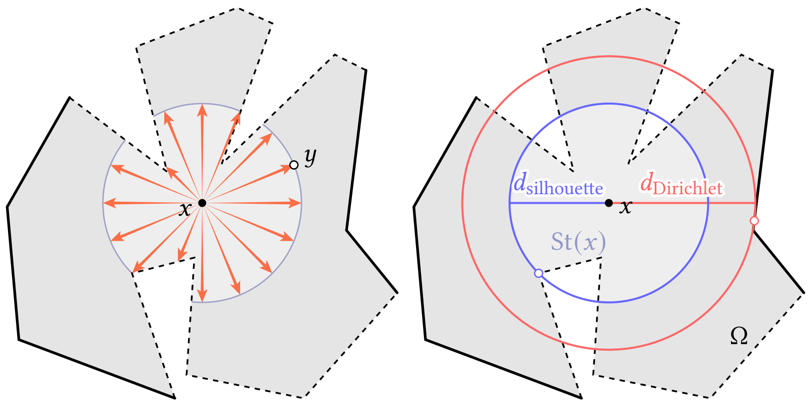

Figure 2: *Left:* a region R is *star shaped* relative to a point x if every point $y \in R$ is visible from x along a straight line. *Right:* WoSt identifies star shaped regions St by finding the largest ball that contains neither the Dirichlet boundary nor the silhouette of the Neumann boundary.

The WoSt algorithm is almost like the WoS algorithm, except that it simulates random walks that "reflect" off the Neumann boundary $\partial\Omega_N$, in addition to the walks already "absorbed" into the Dirichlet boundary $\partial\Omega_D$. To do so, WoSt makes just two small changes to WoS. First, in place of spheres $S(x)$, WoSt uses *star-shaped regions* $\mathrm{St}(x)$ (see Figure 2). Unlike the spheres used by WoS, which touch the boundary only at isolated points, the star-shaped regions used by WoSt can include large pieces of the Neumann boundary $\partial\Omega_N$. Second, rather than terminating walks at all boundaries, WoSt continues walking from any point on the Neumann part of the boundary $\partial\Omega_N$. The overall flow now goes like this:

```
• for $i = 1, \ldots, M$:
    – until $x_k$ is within a small distance $\varepsilon$ of the Dirichlet boundary $\partial\Omega_D$:
        – Compute the distance $d_{\text{Dirichlet}}$ to the closest point on $\partial\Omega_D$.
        – Compute the distance $d_{\text{Silhouette}}$ to the closest silhouette point on $\partial\Omega_N$.
        – Let $r := \min(d_{\text{Dirichlet}}, d_{\text{Silhouette}})$.
        – Shoot a ray from $x_k$ in a random direction $v$, letting the next point $x_{k+1}$ be the first point where the ray hits
          either (i) the sphere $S(x_k, r)$, or (ii) the Neumann boundary $\partial\Omega_N$.
    – Add the value of $g(\overline{x}_k)$ to a running total $\widehat{u}$.
• Return the estimate $\widehat{u}/M$.
```

Here we've omitted a couple small but important details which we'll cover later—*e.g.*, the ray v must be sampled from a hemisphere rather than a sphere if x_k is on the boundary, and we generally want to set some minimum radius r_{min}. But our final implementation really won't be much more complicated than the short outline above. The main thing you may notice is that we now have three geometric queries to implement: a *closest point query*, a *closest silhouette point query*, and a *closest ray intersection*. We'll first walk through how to implement these queries in Section 2, followed by a more detailed implementation of WoSt in Section 3.

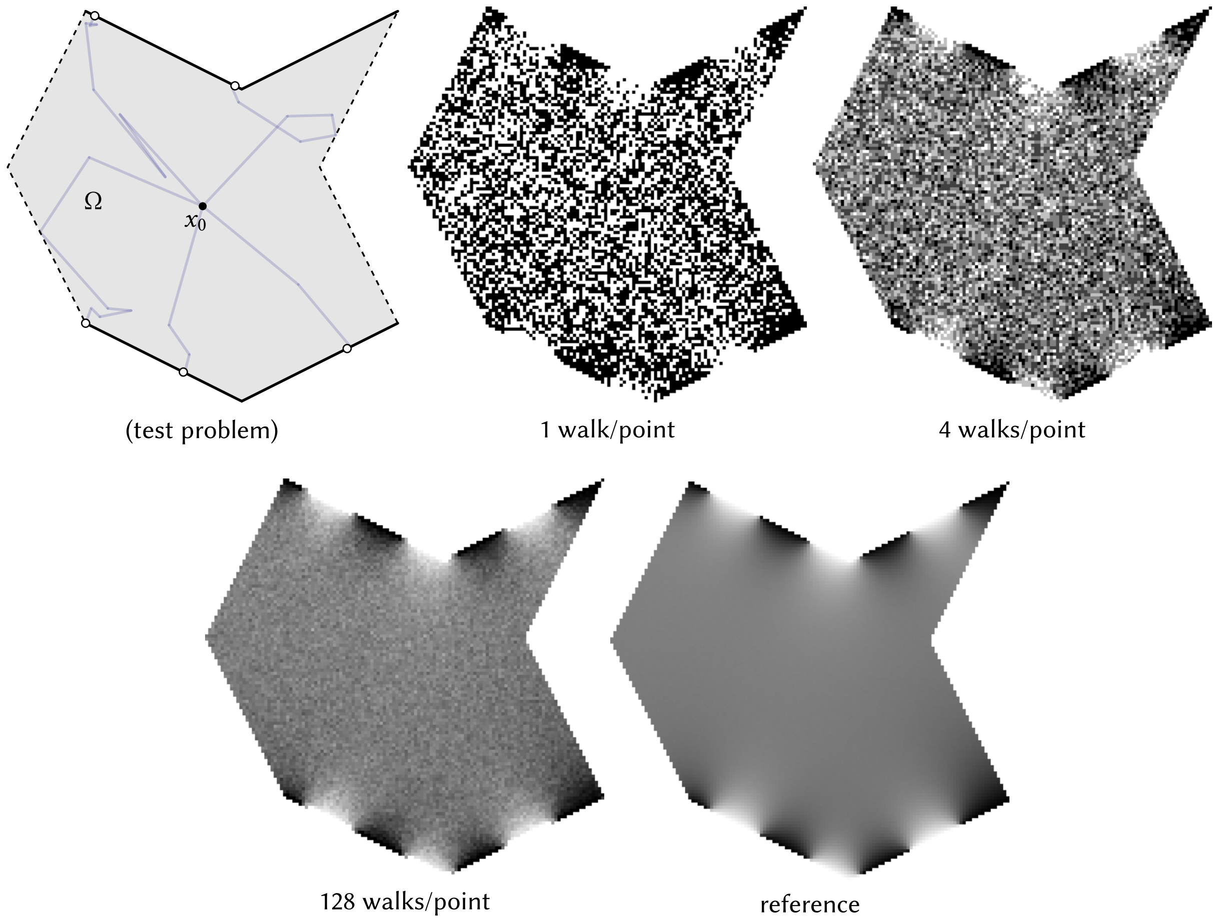

Figure 3: Example walks taken by the WoSt algorithm are shown at *top, left.* Taking more and more walks per point (blue lines), and averaging the resulting boundary values (white points), yields a smoother and more accurate solution estimate *(bottom right).*

2 Geometric Queries

Numerous algorithms in visual, geometric, and scientific computing must perform *geometric queries*: given a piece of geometry (*e.g.*, a polygon mesh or implicit surface), answer some question about this geometry, relative to a query point x (*e.g.*, how far away is the closest point?). Many queries are well-studied, with extremely efficient implementations available on a variety of platforms. When the number of geometric primitives gets very large (*e.g.*, thousands or millions of triangles), it becomes especially important to use a *spatial data structure* like a bounding volume hierarchy (BVH) or binary space partition (BSP) to accelerate computation. To keep things simple, we'll here just assume that the number of primitives is small, and check each primitive directly. For an implementation-oriented introduction to spatial data structures, see Pharr et al. [2016, Section 4.2].

Assumptions. To keep our discussion (and our code) simple, we'll assume that the geometry of the domain Ω is a collection of closed simple polygons in the 2D plane, possibly with holes. More explicitly, we'll assume we're given two lists of polylines: one for the Dirichlet boundary, and one for the Neumann boundary. Each polyline is given by a list of 2D points $p_0, \ldots, p_m \in \mathbb{R}^2$, which are encoded via a `Vec2D` type with all the usual 2D vector operations. We also assume that all polylines have a consistent winding direction—for instance, if the domain is the letter "A", then polylines around the outer boundary should go in counter-clockwise order, and polylines around the inner hole should go in clockwise order (see Figure 7). This final assumption will just make it easier to get a consistent normal direction, and to do inside-outside tests if needed. In fact, most of these assumptions are not strictly required by the WoSt algorithm itself; they largely just make coding easier. See especially Sawhney et al. [2023, Appendix B] for handling of "messier" input geometry.

Here we break each query down into an "atomic" query for a single line segment, and a "global" query which basically just applies the atomic query to each segment in the overall list of polylines.

2.1 Closest Point Query

A *closest point query* seeks the point closest to the geometry relative to a given query point x. Although closest point queries sometimes come up in the context of signed distance fields, we won't need to worry about sign or orientation here: we simply want to find the closest point, full stop.

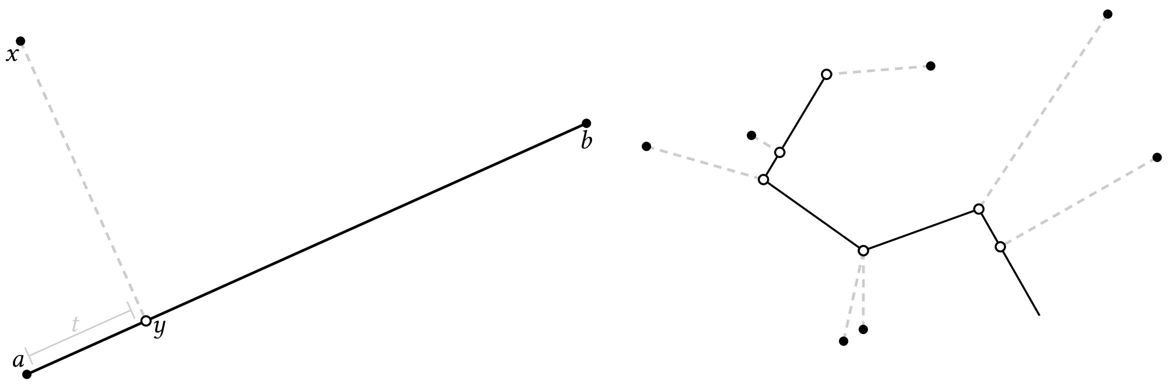

Figure 4: *Left:* given a query point x, the closest point y on a line segment between a and b can be at either an interior point ($0 < t < 1$) or one of the endpoints ($t = 0, 1$). *Right:* The closest point on the boundary of our polygonal domain is then just the closest point among all segments in the domain boundary (here each black point is connected to the closest white point).

Atomic. Given a query point x, we want to find the closest point on a line segment with endpoints a, b (Figure 4). To find this point, we could first find the closest point on the full line, check if it's inside the segment, and if not, take the closer of the two endpoints. Fortunately, all these operations boil down to a very simple subroutine (Quilez [2020] gives a nice derivation):

```
// returns the closest point to x on a segment with endpoints a and b
Vec2D closestPoint( Vec2D x, Vec2D a, Vec2D b ) {
   Vec2D u = b-a;
   double t = clamp( dot(x-a,u)/dot(u,u), 0.0, 1.0 );
   return (1.0-t)*a + t*b;
}
```

Global. To find the distance from a query point x to a collection of polylines, we now just need to apply our atomic closest point query to all segments and keep the smallest distance:

```
// returns distance from x to closest point on the given polylines P
double distancePolylines( Vec2D x, const vector<Polyline>& P ) {
   double d = infinity; // minimum distance so far
   for( int i = 0; i < P.size(); i++ ) { // iterate over polylines
      for( int j = 0; j < P[i].size()-1; j++ ) { // iterate over segments
         Vec2D y = closestPoint( x, P[i][j], P[i][j+1] ); // distance to segment
         d = min( d, length(x-y) ); // update minimum distance
      }
   }
   return d;
}
```

The function `length` computes the length of the given vector, and the constant `infinity` is equal to floating-point infinity (`Inf`); as long as the polylines `P` contain at least one segment, the final distance `d` will be a finite value.

2.2 Closest Silhouette Point Query

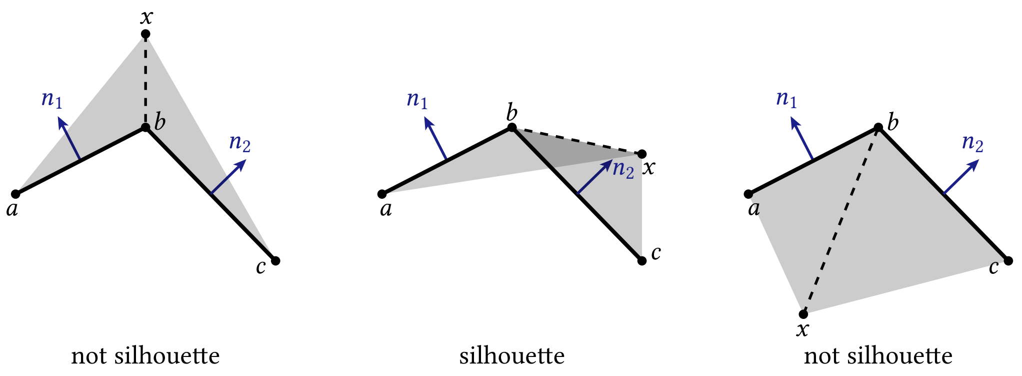

Figure 5: A point b is a *silhouette point* if only one of the segments ab or bc is front-facing relative to a point x.

Silhouette points are points where geometry changes from front-facing to back-facing, relative to a query point x. A *closest silhouette point query* seeks the closest such point along the entire silhouette. In general, queries of the visibility silhouette are far less common than closest ray or closest point queries—a good reference is the work on *spatialized normal cone hierarchies (SNCH)* by Johnson and Cohen [2001]; Sawhney et al. [2023, Section 5.1] explains how to accelerate closest silhouette point queries using an SNCH. Here we'll again just perform brute-force evaluation, rather than use an acceleration scheme.

Atomic. Given a query point x, we want to determine whether the point b on a connected pair of line segments ab, bc is a silhouette point. Equivalently, we want to know if x is on opposite sides of these two segments (*e.g.*, in front of ab and behind bc). We can easily check which side of ab the point x is on by just computing the determinant of the vectors $u := a - x$ and $v := b - x$, which is equivalent to the signed area of triangle xab. Explicitly, $\det(u, v) = u_1 v_2 - u_2 v_1$. Since this operation looks a lot like a cross product, we'll call it `cross` rather than `det` in our code (which also helps to distinguish it from `dot`!). Our final routine is the quite simple:

```
// returns true if the point b on the polyline abc is a silhoutte relative to x
bool isSilhouette( Vec2D x, Vec2D a, Vec2D b, Vec2D c ) {
   return cross(b-a,x-a) * cross(c-b,x-b) < 0;
}
```

The logic here is simply that if x is on the same side of both segments, then the determinants will have the same sign—hence their product will be positive. Otherwise it will be negative.

Global. To find the distance to the closest silhouette point we can once again iterate over all the geometry, check if each point is a silhouette point, and keep track of the closest distance seen so far:

```
// returns distance from x to closest silhouette point on the given polylines P
double silhouetteDistancePolylines( Vec2D x, const vector<Polyline>& P ){
   double d = infinity; // minimum distance so far
   for( int i = 0; i < P.size(); i++ ) { // iterate over polylines
      for( int j = 1; j < P[i].size()-1; j++ ) { // iterate over segment pairs
         if( isSilhouette( x, P[i][j-1], P[i][j], P[i][j+1] )) {
            d = min( d, length(x-P[i][j]) ); // update minimum distance
         }
      }
   }
   return d;
}
```

Note that in this code we check only the interior vertices of the polyline (by skipping the first and last index), since we assume that the domain boundary includes no open curves. In general, however, one should account for the possibility that the endpoints of a polyline may be silhouette points. Closed pure-Neumann loops can still be represented in the present code, by just repeating the points p_0, p_1 at the end of the list.

Note that if all evaluation points x_0 are inside the domain, an easy acceleration is to pre-process the Neumann boundary and keep only concave edge pairs (or triangle pairs, in the 3D case). We do not however implement that acceleration here.

2.3 Closest Ray Intersection

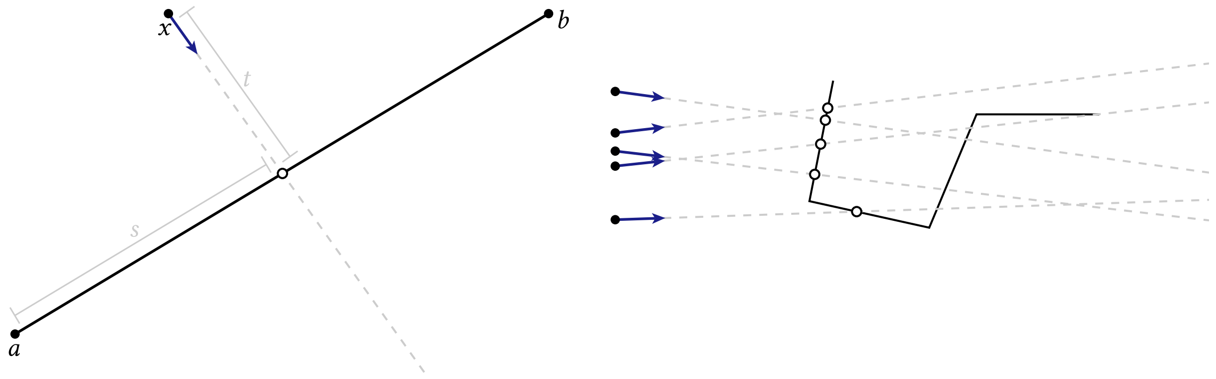

Figure 6: *Left:* a ray-segment intersection computes the time $t \in (0, \infty)$ at which the ray $r(t) = x + tv$ intersects the segment, and the parameter $s \in [0, 1]$ describing the location of the hit point along the line containing the segment. *Right:* the first point where a ray hits a polygonal boundary then corresponds to the smallest positive t value.

Finally, given a point x and a direction v, a *closest ray intersection query* seeks the "time" t at which the ray $x + tv$ first hits a given set. In our case the set we want to consider is the boundary of the star-shaped domain $\mathrm{St}(x, r)$. Equivalently, we want to find the first t such that the ray hits either (i) the sphere $S_r(x)$ of radius r around x, or (ii) the Neumann boundary $\partial\Omega_N$ of the domain. Our atomic query is still just an ordinary ray-segment intersection; logic in the global query then takes care of the ball radius.

Atomic. There are many ways to derive (and implement) ray-segment intersection, but the basic idea is that if we express the line along the ray as $f(t) := x + tv$ and the line along the segment as $g(s) := (1 - s)a + sb$, then we want to solve for the values s, t such that $f(t) = g(s)$, *i.e.*, the point at which the two lines coincide. We then check whether s is between 0 and 1 (*i.e.*, the intersection is within the segment) and $t > 0$ (*i.e.*, the intersection is in the direction of the ray). After some algebraic manipulation, we arrive at one possible implementation:

```
// returns the time t at which the ray x+tv intersects segment ab,
// or infinity if there is no intersection
double rayIntersection( Vec2D x, Vec2D v, Vec2D a, Vec2D b ) {
   Vec2D u = b - a;
   Vec2D w = x - a;
   double d = cross(v,u);
   double s = cross(v,w) / d;
   double t = cross(u,w) / d;
   if (t > 0. && 0. <= s && s <= 1.) {
      return t;
   }
   return infinity;
}
```

Global. To find the first hit, we can now iterate over all segments in the given polylines and keep the minimum time t at which any intersection occurs. Since we also want to intersect with a sphere of given radius r, we can use r as the initial value for t—if we never encounter a closer segment, then we've hit the sphere:

```
// finds the first intersection y of the ray x+tv with the given polylines P,
// restricted to a ball of radius r around x.  The flag onBoundary indicates
// whether the first hit is on a boundary segment (rather than the sphere), and
// if so sets n to the normal at the hit point.
Vec2D intersectPolylines( Vec2D x, Vec2D v, double r,
                          const vector<Polyline>& P,
                          Vec2D& n, bool& onBoundary ) {
   double tMin = r; // smallest hit time so far
   n = Vec2D{ 0.0, 0.0 }; // first hit normal
   onBoundary = false; // will be true only if the first hit is on a segment
   for( int i = 0; i < P.size(); i++ ) { // iterate over polylines
      for( int j = 0; j < P[i].size()-1; j++ ) { // iterate over segments
         const double c = 1e-5; // ray offset (to avoid self-intersection)
         double t = rayIntersection( x + c*v, v, P[i][j], P[i][j+1] );
         if( t < tMin ) { // closest hit so far
            tMin = t;
            n = rotate90( P[i][j+1] - P[i][j] ); // get normal
            n /= length(n); // make normal unit length
            onBoundary = true;
         }
      }
   }
   return x + tMin*v; // first hit location
}
```

Notice that we compute the location x and normal n of the intersection point, as well as a flag `onBoundary` indicating whether the hit occurred on the domain boundary (rather than the sphere), which we'll need for the WoSt algorithm. As is common practice in ray tracing, we also offset the ray origin by a small distance c so that the intersection point we find is not just the ray origin x; a more numerically robust treatment of ray offsetting is described by Wächter and Binder [2019].

3 Walk on Stars

With all of our geometric queries in place, the actual WoSt algorithm is quite short. The input is a query point x_0, a collection of polylines specifying the Dirichlet and Neumann parts of the boundary, and a function g that evaluates the Dirichlet value at any point x near the Dirichlet boundary $\partial\Omega_D$:

```
double solve( Vec2D x0, // evaluation point
              vector<Polyline> boundaryDirichlet, // absorbing part of the boundary
              vector<Polyline> boundaryNeumann, // reflecting part of the boundary
              function<double(Vec2D)> g ) { // Dirichlet boundary values
   const double eps = 0.0001; // stopping tolerance
   const double rMin = 0.0001; // minimum step size
   const int nWalks = 65536; // number of Monte Carlo samples
   const int maxSteps = 65536; // maximum walk length
```

Here we also specify several solver parameters, which provide trade offs between speed and accuracy:

- a value $\varepsilon > 0$ which says how close we need to get to the Dirichlet boundary before we stop and grab the boundary value,
- the parameter $r_{\min} > 0$ similarly limits how small our steps will shrink near the silhouette,
- the `nWalks` parameter says how many total walks we should take (hence how many boundary values we should average) to get our solution estimate, and
- the `maxSteps` parameter limits the number of steps taken by any walk.

The value `sum` will be used to accumulate all the values g we encounter at the boundary; normalizing this value by the number of walks gives us our final Monte Carlo estimate of the solution.

At the beginning of each step, we assume that we're at an interior point (and hence the normal n is not well-defined):

```
   double sum = 0.0; // running sum of boundary contributions
   for( int i = 0; i < nWalks; i++ ) {
      Vec2D x = x0; // start walk at the evaluation point
      Vec2D n{ 0.0, 0.0 }; // assume x0 is an interior point, and has no normal
      bool onBoundary = false; // flag whether x is on the interior or boundary
```

For each step, the first thing we do is compute the radius of the ball used to define a star shaped region $\mathrm{St}(x, r)$. This radius is the either the distance to the Dirichlet boundary, or the distance to the silhouette of the Neumann boundary—whichever one is closer. We also limit the ball size to be at *least* as big as the smallest step size parameter $r_{\min}$:

```
      double r, dDirichlet, dSilhouette; // radii used to define star shaped region
      int steps = 0;
      do { // loop until the walk hits the Dirichlet boundary

         // compute the radius of the largest star-shaped region
         dDirichlet = distancePolylines( x, boundaryDirichlet );
         dSilhouette = silhouetteDistancePolylines( x, boundaryNeumann );
         r = max( rMin, min( dDirichlet, dSilhouette ));
```

Next, we shoot a ray from the current point x in a random direction v. If x is on the domain interior, then we sample v from the full sphere (or in 2D: circle) around x; if it's on the boundary, then we sample the hemisphere around the inward-pointing normal direction v, so that the next point ends up inside the domain:

```
         // intersect a ray with the star-shaped region boundary
         double theta = random( -M_PI, M_PI );
         if( onBoundary ) { // sample from a hemisphere around the normal
            theta = theta/2. + angleOf(n);
         }
         Vec2D v{ cos(theta), sin(theta) }; // unit ray direction
         x = intersectPolylines( x, v, r, boundaryNeumann, n, onBoundary );
```

Recall that the routine `intersectPolylines` will also update the flag `onBoundary`, and sets the normal n if the new point is on the boundary. This loop continues taking steps along the walk until we get within a small distance ε of the Dirichlet boundary, or until we've exceeded the maximum number of steps—at this point we stop walking, and add the boundary contribution at the final location of x:

```
         steps++;
      }
      while(dDirichlet > eps && steps < maxSteps);
      //stop if we hit the Dirichlet boundary, or the walk is too long

      if( steps >= maxSteps ) cerr << "Hit max steps" << endl;

      sum += g(x); // accumulate contribution of the boundary value
   }
```

After all walks are finished we return the average boundary value, which gives our solution estimate. Figure 1, *right* shows an example walk. (If you're curious about *why* this algorithm works, you'll have to take a look at the full paper!)

4 Inside-Outside Test (optional)

Though not strictly part of the WoSt algorithm itself, we may also want to know whether a given evaluation point x_0 is inside or outside the problem domain Ω (*e.g.*, for visualizing the solution). To answer this query, we can just measure the total signed angle of the boundary polygon, relative to the query point:

```
// Returns true if the point x is contained in the region bounded by the Dirichlet
// and Neumann curves.  We assume these curves form a collection of closed polygons,
// and are given in a consistent counter-clockwise winding order.
bool insideDomain( Vec2D x,
                   const vector<Polyline>& boundaryDirichlet,
                   const vector<Polyline>& boundaryNeumann )
{
   double Theta = signedAngle( x, boundaryDirichlet ) +
                  signedAngle( x, boundaryNeumann );
   const double delta = 1e-4; // numerical tolerance
   return abs(Theta-2.*M_PI) < delta; // boundary winds around x exactly once
}
```

This signed angle is computed by just looping over all segments in the polylines, and adding up the signed angle of the individual segments. If we represent points by complex numbers, the angle for a segment with endpoints a, b is just $\theta := \arg((b - x)/(a - x))$, *i.e.*, the angle that rotates the vector $a - x$ to the vector $b - x$:

```
double signedAngle( Vec2D x, const vector<Polyline>& P )
{
   double Theta = 0.;
   for( int i = 0; i < P.size(); i++ )
      for( int j = 0; j < P[i].size()-1; j++ )
         Theta += arg( (P[i][j+1]-x)/(P[i][j]-x) );
   return Theta;
}
```

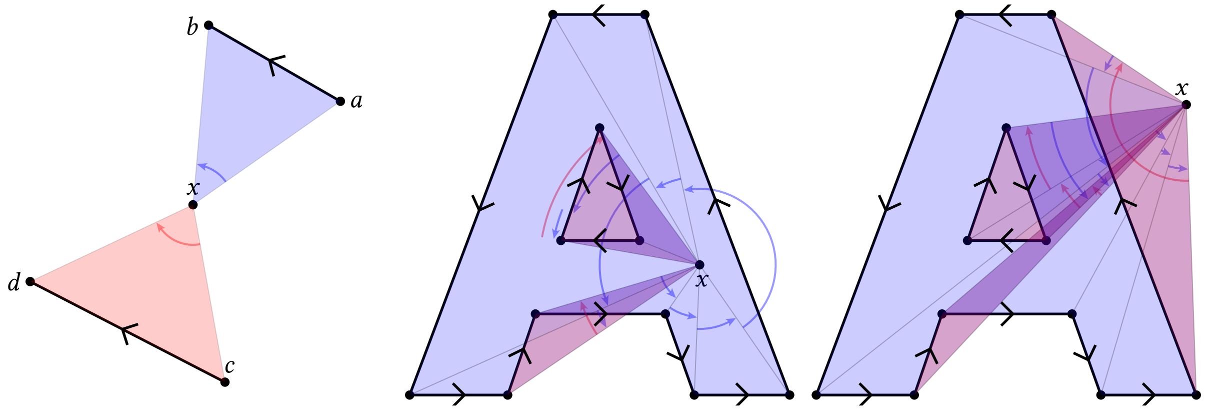

Figure 7: *Left:* the signed angle subtended by segments *ab* and *cd* are positive and negative (*resp.*) with respect to *x*. Summing these signed angles yields a total angle $+2\pi$ for any point *x* inside a polygon *(center)*, and a total angle of zero for points *x* outside the polygon *(right)*.

Acknowledgements

All diagrams in this document were made using Penrose [Ye et al., 2020]. Special thanks to Wode Ni, Sam Estep, Yiliang Liang, and the rest of the Penrose team for development support during production of this document.

References

David E Johnson and Elaine Cohen. Spatialized normal come hierarchies. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 129–134, 2001.

Mervin E Muller. Some continuous monte carlo methods for the dirichlet problem. *The Annals of Mathematical Statistics*, pages 569–589, 1956.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

Inigo Quilez. The sdf of a line segment, 2020. URL `https://www.youtube.com/watch?v=PMltMdi1Wzg`.

Rohan Sawhney, Bailey Miller, Ioannis Gkioulekas, and Keenan Crane. Walk on stars: A grid-free monte carlo method for pdes with neumann boundary conditions. *ACM Trans. Graph.*, XX(X), 2023.

Carsten Wächter and Nikolaus Binder. A fast and robust method for avoiding self-intersection. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pages 77–85, 2019.

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: From mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4), 2020.