Distances

What is the distance between two points? If you are in $\mathbb{R}^d$, then this is a trivial question. The distance between $x \in \mathbb{R}^d$ and $y \in \mathbb{R}^d$ is

$$\|x - y\| = \sqrt{(x-y) \cdot (x-y)}\,. \tag{1}$$

But what if you are in a different space. What if there is an obstacle between x and y? Or what if x and y are on a surface $\Omega \subseteq \mathbb{R}^d$, and we are looking for their distance within Ω? In this scenario, even the definition of distances becomes difficult.

Shortest path

A roundabout way to define the distance between two points is to define it as the length of the shortest path between them.

Definition 1: Path

A path in $\mathbb{R}^d$ is a C^1 (continuous and differentiable) function

$$\gamma : [0,1] \to \mathbb{R}^d\,. \tag{2}$$

The length of γ is given by

$$L(\gamma) = \int_0^1 \left\| \frac{\partial \gamma}{\partial t}(t) \right\| dt\,. \tag{3}$$

We can use paths to define the distances between two points

Definition 2: Path distance

Let $x, y \in \mathbb{R}^d$. The distance between x and y is defined as the length of the shortest path that connects them, i.e.,

$$d(x,y) = \min_{\substack{\gamma:[0,1]\to\mathbb{R}^d \text{ path} \\ \gamma(0)=x,\gamma(1)=y}} L(\gamma)\,. \tag{4}$$

Theorem 1: Equivalence of Euclidean and shortest-path distance

For two points $x, y \in \mathbb{R}^d$, the shortest-path distance of Definition 2 is equivalent to the Euclidean distance of (1).

Why do we care about defining the distance via paths? Paths easily generalize to many situations where the naive distance from (1) does not work.

Obstacles in $\mathbb{R}^d$

Consider now the case of $\mathbb{R}^d \setminus O$, where $O \subseteq$ is some obstacle. What is the distance between two points $x, y \in \mathbb{R}^d$ now? Figure 1 shows that we can not just use $\|x - y\|$ again. $\|x - y\|$ is the length of the line passing through the obstacle. That is not a useful notion of distance, since people walking in this space cannot actually pass through the obstacle.

In this scenario, we have to restrict ourselves to paths in $\mathbb{R}^d \setminus O$ and compute distances via shortest paths only.

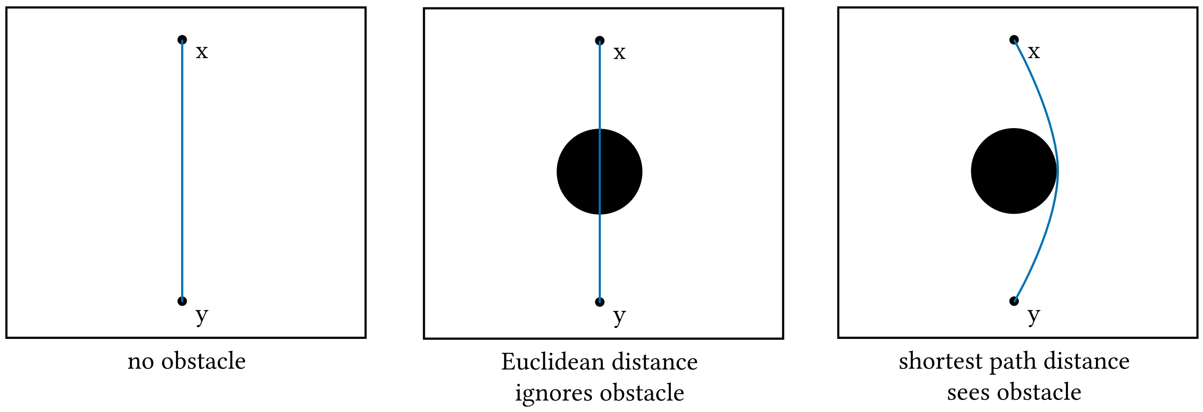

Figure 1: In the presence of obstacles, the distance $\|x - y\|$ fails to measure what we intuitively think distance is - the length of the shortest path between two points.

Definition 3: Path distance in an arbitrary space

Let $U \subseteq \mathbb{R}^d$ (in the case of our domain with an obstacle, $U = \mathbb{R}^d \setminus O$). Let $x, y \in U$. The distance between x and y is defined as the length of the shortest path *entirely within* U that connects them, i.e.,

$$d(x,y) = \min_{\substack{\gamma:[0,1]\to U \text{ path} \\ \gamma(0)=x,\gamma(1)=y}} L(\gamma)\,. \tag{5}$$

Dijkstra's method

Computing the Euclidean distance (1) is easy. Computing the shortest-path distance in the absence of any obstacles is also easy. But how does one compute the shortest-path distance in the presence of obstacles? There are infinitely many possible paths $\gamma : [0,1] \to U$. A classical way to solve this dilemma is to discretize the shape, and then use a *shortest-path algorithm.*

We can discretize the space U by turning it into a graph G with vertices and edges, where every edge measures a distance between two vertices. This is very similar to turning a surface into a triangle or quad mesh, but slightly different - we do not assign any importance to the faces, and in fact we do not really require our graph to be manifold at all. A popular choice is to simply make a grid in space, discard all nodes covered by obstacles, and connect neighboring nodes with graph edges.

On this graph G we can now run *Dijkstra's algorithn* to compute the distance between one point and every other point. Dijkstra's algorithm is a classical greedy algorithm that computes the distance from a source to every other vertex in the graph by locally finding the shortest distance from every neighbor. It goes as following:

- Keep track of the shortest distance to every vertex in the graph. Start with this distance being ∞ for every vertex except for the source vertex.
- Successively visit every vertex, starting with the lowest-distance currently unvisited vector.
- For every vertex that is visited, update the distance of that vertex's neighbor to the minimum of their current distance and the distance of the visited vertex plus the edge that connects the visited vertex with the neighbor.

The result of this algorithm is a distance labeling for every vertex in the graph. The label of every vertex is the length of the shortest path to the source.

In Pseudocode form:

```
def distance(G, source):
  vertices_to_check = G.vertices
  dists = constant_vector(size=G.vertices.size, value=infinity)
  dists(source) = 0
  while vertices_to_check is not empty:
    v = get_vertex_to_check_with_smallest_dist(vertices_to_check, dists)
    vertices_to_check.remove(v)
    for e in edges(v):
      w = neighbor_along_edge(v,e)
      if e.dist + dists(v) < dists(w):
        dists(w) = e.dist + dists(v)
  return dists
```

The problem with Dijkstra's algorithm can be clearly seen in Figure 2. Dijkstra's method follows only edges of the graph in determining the shortest path. If the shortest path is not along edges, then the distance that it computes can be very far off from the correct result.

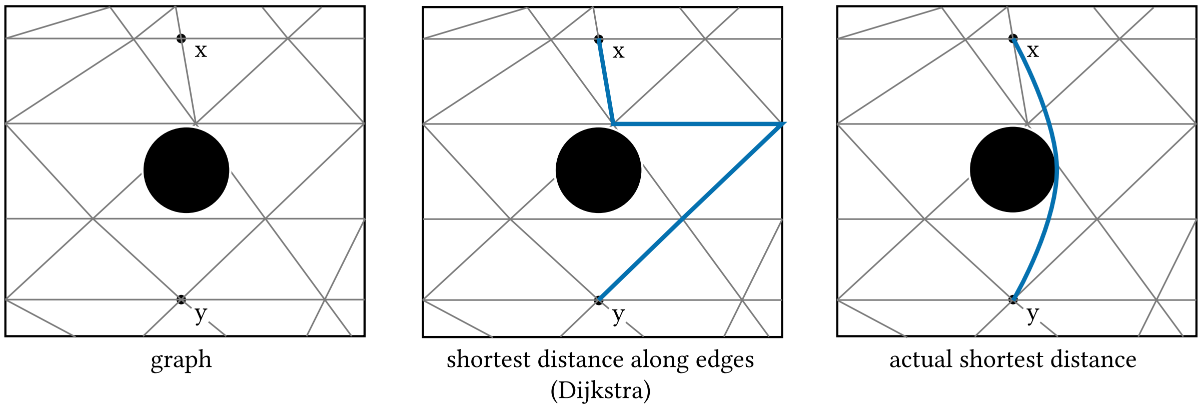

Figure 2: Dijkstra's algorithm finds the shortest path going only along the edges of the mesh / graph. If the edges are not well-aligned with the shortest path, Dijkstra will return a result that is far off from the exact result.

Distances on surfaces

Let us now move on from computing distances in Euclidean space $\mathbb{R}^d$ (or Euclidean space with obstacles) towards computing distances on surfaces. As with $\mathbb{R}^d$, there are two different questions that we will tackle successively:

- How do we decine an exact distance on surfaces $\Omega \subseteq \mathbb{R}^k$?
- How do we compute this distance on a computer?

Computing such distances is very common in practice. Think, e.g., about one of the most common surfaces you see every day in your life: the surface of the earth. Planning an airplane flight path on earth is an exercise in finding the shortest path between two points on a surface $\Omega \subseteq \mathbb{R}^3$. The earth's curvature is important here: Routest that look like the shortest path on a map are not actually the shortest path between two points. As Figure 3 shows, shortest paths in a projection that renders latitudes as straight lines differ significantly from the real shortest paths on the globe.

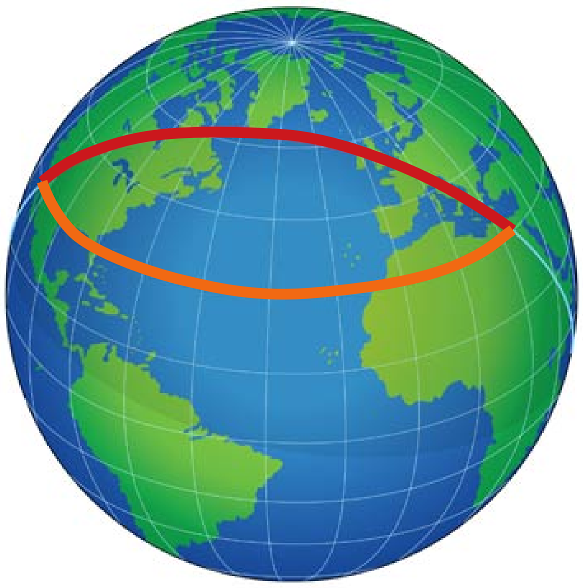

Figure 3: The shortest path between two points on the globa *(red)* compared to what looks like a straight line on a flat map that renders the longitudinal and latitudinal circles as straight lines [3].

Paths on surfaces

We define the distance on a surface via shortest paths, just like we did for Euclidean space with obstacles. The main difference is that we now confine our paths γ to be entirely contained in the surface Ω. This is very similar Definition 3, just that now $U = \Omega$, a surface in $\mathbb{R}^k$.

Definition 4: Path distance on a surface

Let $\Omega \subseteq \mathbb{R}^k$ be a d-dimensional surface in $\mathbb{R}^k$. Let $x, y \in \Omega$. The distance between x and y is defined as the length of the shortest path *entirely within* U that connects them, i.e.,

$$d(x, y) = \min_{\substack{\gamma:[0,1]\to\Omega \text{ path} \\ \gamma(0)=x,\gamma(1)=y}} L(\gamma)\,. \tag{6}$$

Geodesics

Surfaces are trickier than Euclidean space with obstacles on multiple levels. While for obstacles, the *global* problem of finding the shortest path becomes hard, the local problem is still easy - the shortest path between two very close points is still a short line (as long as they are not on the boundary of the obstacle). For surfaces, this is no longer true - the shortest path connecting two infinitesimally close points is no longer an Euclidean straight line (see Figure 4). This is because the path definition from Definition 4 restricts the path to lie entirely on the surface, and this shortest path might not be a line anywhere.

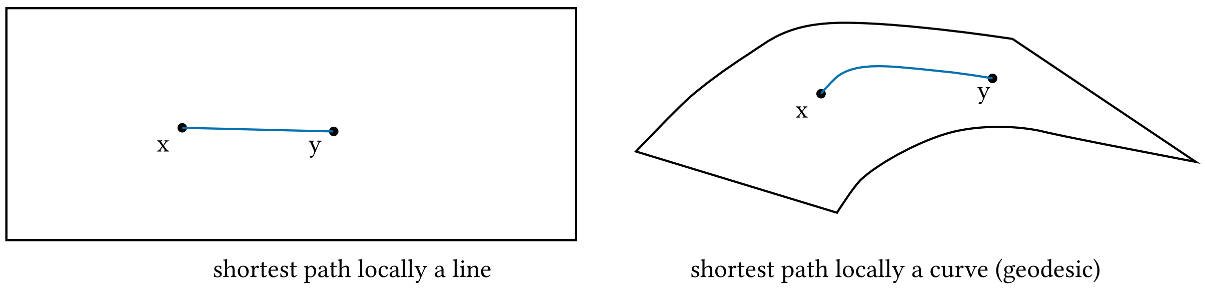
x
y
shortest path locally a line
x
y
shortest path locally a curve (geodesic)

Figure 4: In Euclidean space, the shortest path between two points is always a straight line, as long as the two points are not on any obstacle *(left)*. On a surface $\Omega \subseteq \mathbb{R}^k$, this is no longer true.

Where we have the line in Euclidean space, on surfaces we have the *geodesic*: the locally shortest path between two infinitesimally close points.

Definition 5: Geodesic

Let $\Omega \subseteq \mathbb{R}^k$ be a d-dimensional surface in $\mathbb{R}^k$.

A curve $\gamma : [0, 1] \to$ is a geodesic if any infinitesimal variation of the curve would increase its length.

Geodesics are the shortest path between two infinitesimally close points. This means that, for every point $x \in \Omega$ there is an open set $U \subseteq \Omega$, $x \in U$ such that, for all $y \in U$ a geodesic connecting x and y is the shortest path between x and y.

The fact that the locally shortest path between two points on a surface is called a geodesic lends a popular name to the path-distance on a surface (Definition 4): *geodesic distance* (the distance within Ω). This is to differentiate it from the *Euclidean distance*, the straight-line distance (in the surrounding Euclidean space $\mathbb{R}^d$).

Computing geodesic distances

Let us now return to the problem of computing the geodesic distance on an actual discrete surface. Recall the main problem with Dijkstra's algorithm (Figure 2): Its shortest paths have to align with edges. How can we compute shortest paths that do not have to follow edges exactly?

We will use a partial differential equation whose solutions are known to be geodesic distance functions: the *eikonal equation* (eikon is the Greek word for image - this reveals the rich history of this PDE in optical physics).

Definition 6: Eikonal equation

Let $\Omega \subseteq \mathbb{R}^k$ be a d-dimensional surface in $\mathbb{R}^k$.

The *Eikonal equation* is the partial differential equation

$$\|\nabla u(x)\| = 1 \qquad \forall x \in \Omega\,. \tag{7}$$

The one-dimensional version of the eikonal equation should give us some intuition for why its solutions are distance functions. If a function has constant derivative 1, then it changes its value by 1 whenever its argument advances by 1, and thus $u(y) - u(x)$ is the distance between x and y. This generalizes even to curved surfaces, where solutions to the eikonal equation measure the *geodesic* distance.

We can use the eikonal equation to measure the distance from a source point x to any other point by adding the following boundary condition to (7):

$$u(x) = 0, \qquad u(\infty) = \infty, \qquad u \geq 0\,. \tag{8}$$

The notation $u(\infty)$ is a small abuse of notation which means that the distance should go to ∞ as the point gets farther away from x.

Approximate geodesic distance

How do we solve (7)? One can employ the famous fast marching method [2]. It is a generalization of Dijkstra's method to compute shortest distances that do not have to go along edges by computing the shortest path along graph edges.

Fast marching is, however, both expensive as well as difficult to implement (which is why you will not have to implement it in the homework). Instead, we will resort to computing approximate geodesic distances using the heat method [1].

The heat method uses the so-called *heat kernel* to compute geodesic distances between points.

Definition 7: Heat kernel

Let $\Omega \subseteq \mathbb{R}^k$ be a d-dimensional surface in $\mathbb{R}^k$.

The *heat kernel* is a function $k_{t,x} : \Omega \to \mathbb{R}$ that solves the following heat equation:

$$\begin{aligned} \frac{\partial k_{t,x}(y)}{\partial t} &= -\Delta_x k_{t,x}(y) \\ \lim_{t\to 0} k_{t,x}(y) &= \delta_x(y)\,, \end{aligned} \tag{9}$$

where δ_x is the Dirac delta function that is zero everywhere except at x, where it is ∞, and $\int_\Omega \delta_x(y)\,dy = 1$.

The heat kernel, historically, comes from the fact that one can use it so solve the heat equation $\frac{\partial u(t,x)}{\partial t} = -\Delta u(t,x)$ for any initial condition $u_0(x)$ via the simple integral $\int_\Omega k_{t,x}(y) u_0(y)\,dy$:

$$\begin{aligned} u(t,x) &= \int_\Omega k_{t,x}(y) u_0(y)\,dy \\ \frac{\partial u(t,x)}{\partial t} &= \int_\Omega u_0(y) \frac{\partial k_{t,x}(y)}{\partial t}\,dy = -\int_\Omega u_0(y) \Delta_x k_{t,x}(y)\,dy = -\Delta u(t,x) \\ u(0,x) &= \int_\Omega u_0(y) \delta_x(y)\,dy = u_0(x)\,. \end{aligned} \tag{10}$$

The heat kernel can, however, also be used to compute geodesic distances using *Varadhan's formula.*

Theorem 2: Varadhan's formula

Let Ω be a d-dimensional surface in $\mathbb{R}^k$. The geodesic distance between $x, y \in \Omega$ is given by the formula

$$d(x,y) = \lim_{t\to 0} \sqrt{-4t \log k_{t,x}(y)}\,. \tag{11}$$

It is tempting to directly compute Varadhan's formula for a small $t > 0$ and use it to simply compute distances between a source point $x \in \Omega$ and all other points of the surface. This does not work directly: As Crane et al. [1] show, this leads to numerical instabilities, since the heat kernel, for small times, involves the division of very small numbers by other very small numbers. They propose their heat geodesic algorithm as a workaround that features repeated renormalization of the result - it turns out that while the magnitude of the heat kernel is very sensitive to numerical noise, the direction of its gradient is not.

Pick a small $t > 0$ (it works best when t is about the edge length squared). Set an initial function u_0 on Ω that is 1 at the source point, and 0 everywhere else. Then:

1. Solve the heat equation $\frac{\partial u(t,x)}{\partial t} = -\Delta u(t,x)$ for time t and initial function u_0.
2. Evaluate the vector field $\boldsymbol{v} = -\frac{\nabla u}{\|\nabla u\|}$.
3. Solve the Poisson equation $-\Delta_y d_x(y) = \nabla \cdot \boldsymbol{v}$.

This heat geodesic algorithm is extremely simple, and can mostly be built with tools that we have already derived in previous chapters, such as the cotangent Laplacian or the finite element gradient. Figure 5 shows the simple heat geodesic algorithm in practice.

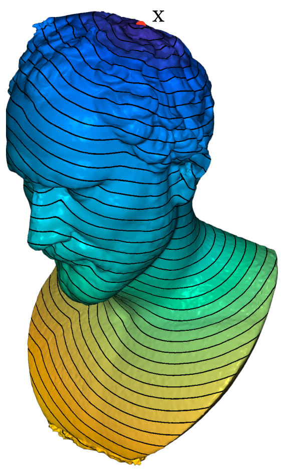

Figure 5: The heat geodesic algorithm can quickly and easily compute the distance between a source vertex $x \in \Omega$ and any other point on the mesh using standard finite element matrices.

Bibliography

[1] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. 2013. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.* 32, 5 (2013).

[2] Ron Kimmel and James A. Sethian. 1998. Computing geodesic paths on manifolds. *Proc. Natl. Acad. Sci. USA* 95, (1998), 8431–8435.

[3] Orion 8. 2013. Orthodrome auf der Erdkugel zwischen Los Angeles und London (https://de.wikipedia.org/wiki/Orthodrome#/media/Datei:Orthodromic_air_route.tif).