IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

March 7, 2017

1 EIKONAL EQUATION (FAST MARCHING ALGORITHM)

1.1 THE EIKONAL EQUATION

Interface problems are very common in science and engineering, ranging from multiphase fluid problems to medical imaging. Interfaces, fronts and edges are often represented by an indicator function that measures the minimal distance from any point in space to the interface. The interface is then defined as the zero-isosurface of the higher-dimensional distance function.

If we define a propagating interface implicitly as $u(\mathbf{x}(t), t) = 0$, the chain rule produces a propagation equation for u according to

$$\frac{\partial u}{\partial t} + \nabla u \cdot \frac{d\mathbf{x}}{dt} = 0. \tag{1.1}$$

We assume that the front propagates in the direction of the local normal $\mathbf{n}$ with a speed F which yields

$$\frac{d\mathbf{x}}{dt} \cdot \mathbf{n} = F \qquad\qquad \mathbf{n} = \frac{\nabla u}{|\nabla u|}. \tag{1.2}$$

Substituting this expression into the evolution equation we arrive at

$$\frac{\partial u}{\partial t} + F|\nabla u| = 0. \tag{1.3}$$

The steady-state of the above equation then produces the eikonal equation

$$F|\nabla u| = 1 \tag{1.4}$$

which describes an equation for the travel time of information through a "speed" field F. The speed field represents the local traveling-wave speed and influences the time of travel from any point to any other point. The travel time information, can be used in a multitude of application, among them seismic exploration, optimal path planning and robotics. Constant values of u represent surfaces of constant phase information or the wave fronts; the normals to the same surfaces represent wave rays.
For a special case of $F = 1$, the eikonal equation reduces to a governing equation for a distance function

$$|\nabla u| = 1 \tag{1.5}$$

which states that the absolute value of the gradient is constant. Note that the absolute value is necessary to express that distance from the interface is independent of the direction we march.

The eikonal equation constitutes a nonlinear first-order partial differential equation, which can be solved by the method of characteristics or a wave-propagation approach. Special emphasis has to be drawn to crossing characteristics, in which case, we are interested in a regularized solution or a viscosity solution.

Typically, the eikonal equation is solved starting from an initial condition $u(x) = 0$ for $x \in \mathscr{I}$ which stated that along a given set $\mathscr{I}$ the travel time is zero. From there, we propagate local wave solutions omnidirectionally and thus advance a front of constant travel time. This propagation is contingent on the presence of obstacles, other geometric constraints and the speed field (which sets the rate of front propagation). The final solution then presents the minimal travel time from any point on the initial set $\mathscr{I}$ to any point in the rest of the domain.

This travel time information can be used in multiple ways: (i) in seismic exploration it is used to reconstruct the material layering in the earth's crust from localized measurements; (ii) in medical applications, it is used to segment medical images to isolate abnormal from normal tissue; (iii) in robotics, it is used to solve problems of optimal path and motion planning. In the latter case, the travel time field is used to determine geodesic lines, i.e., lines of minimal distance between two points on a general manifold and/or under a varying propagation speed. The geodesic lines can simply be extracted from the computed travel-time field u according to

$$\frac{d\mathbf{x}}{dt} = -\frac{\nabla u}{|\nabla u|}. \tag{1.6}$$

This states that, starting with the end point of the geodesic line, we integrate backwards along a path of minimal travel time to the starting time, i.e., a point in the set $\mathscr{I}$. In robotics, the vector $\mathbf{x}$ can contain not only Cartesian coordinates, but also rotational angles. For the motion of a multi-degree-of-freedom system (for example, a complex robot arm), the eikonal equation to be solved and subsequently integrated can be rather high-dimensional, and special numerical techniques have to be applied.

In this section, we will concentrate on the two-dimensional Cartesian case, but will structure our `python`-program to allow a straightforward extension to the higher-dimensional case.

1.2 Discretization

We use a highly efficient numerical method to solve the eikonal equation, the Fast Marching Method (FMM). In its core, it is closely related to Dijkstra's method, albeit applied to a Cartesian mesh rather than a graph. We exploit the nature of the eikonal equation and solve it by propagating a front (from one or multiple starting positions) while observing the governing equation. The method calculates the time u it takes for a wave to reach every point in our computational domain. We assume that the front propagates normal to the front.

The start with a Cartesian grid, with i, j denoting the i-th grid point in the x-direction and j-th grid point in the y-direction. The notation u_{ij} then stands for the value of u at the grid point (i, j). We next introduce the first-order gradient approximations

$$\begin{aligned}
D_{ij}^{-x} u &= \frac{u_{ij} - u_{i-1,j}}{\Delta x}, && (1.7a)\\
D_{ij}^{+x} u &= \frac{u_{i+1,j} - u_{ij}}{\Delta x}, && (1.7b)\\
D_{ij}^{-y} u &= \frac{u_{ij} - u_{i,j-1}}{\Delta y}, && (1.7c)\\
D_{ij}^{+y} u &= \frac{u_{i,j+1} - u_{ij}}{\Delta y}, && (1.7d)
\end{aligned}$$

where Δx and Δy stand for the grid spacing in the x- and y-direction, respectively. Due to the absolute value in the governing equation and the fact that we can have discontinuous gradients in the solution, it is advantageous and necessary to slope-limit the gradients based on the options above. In effect, we have to take the larger of two gradients in each direction, while avoiding negative gradients altogether. The avoidance of negative gradients stems from the fact that travel time away from the initial set $\mathscr{I}$ can only increase. We choose as our implemented discretization the scheme

$$\max\left(D_{ij}^{-x} u, -D_{ij}^{+x} u, 0\right)^2 + \max\left(D_{ij}^{-y} u, -D_{ij}^{+y} u, 0\right)^2 = 1/F_{ij}^2. \qquad (1.8)$$

After a bit of algebra, we arrive at

$$\max\left(\frac{u - u_1}{\Delta x}, 0\right)^2 + \max\left(\frac{u - u_2}{\Delta y}, 0\right)^2 = 1/F_{ij}^2 \qquad (1.9)$$

with the definitions

$$\begin{aligned}
u &= u_{ij} && (1.10a)\\
u_1 &= \min(u_{i-1,j}, u_{i+1,j}) && (1.10b)\\
u_2 &= \min(u_{i,j-1}, u_{i,j+1}) && (1.10c)
\end{aligned}$$

The above equation (1.9) leads to a quadratic equation for u, the solution at the new point. We have

$$\left(\frac{u-u_1}{\Delta x}\right)^2 + \left(\frac{u-u_2}{\Delta y}\right)^2 = \frac{1}{F_{ij}^2} \tag{1.11}$$

with the solution

$$u = -\frac{\alpha}{2} + \frac{1}{2}\sqrt{D} \tag{1.12}$$

and

$$\alpha = -2\frac{\Delta y^2 u_1 + \Delta x^2 u_2}{\Delta x^2 + \Delta y^2}, \tag{1.13a}$$

$$\beta = \frac{\Delta y^2 u_1^2 + \Delta x^2 u_2^2 - \Delta x^2 \Delta y^2 / F_{ij}^2}{\Delta x^2 + \Delta y^2}, \tag{1.13b}$$

$$D = \alpha^2 - 4\beta. \tag{1.13c}$$

It depends on the distribution of u-values in the immediate neighborhood of the grid point (i, j). Special treatment includes the case where one of the terms in (1.9) vanishes, in which case we have a simple linear equation (and a corresponding planar wave propagation):

$$u = u_1 + \frac{\Delta x}{F_{ij}} \qquad \text{or} \qquad u = u_2 + \frac{\Delta y}{F_{ij}}. \tag{1.14}$$

For the algorithm, we propagate a wave front, starting from an inital location, by solving (1.9). For this we divide the grid points into `Accepted`, `NarrowBand` and `UnVisited`. The `Accepted` points are grid points where the travel time u_{ij} has converged to the final value. Initially, only the starting point is in the `Accepted` group, with a value of $u_{ij} = 0$. The `UnVisited` group are grid points that still need to be computed. In the beginning, all but the initial grid point are `UnVisited`. The `NarrowBand` category is among the `UnVisited` group, but is in the neighborhood of the `Accepted` points. The `NarrowBand` points form a thin layer around the `Accepted` points.

With these definitions, the algorithm to solve the above equation proceeds in the steps listed below (following sketch outlined in figure 1.1).

`step0` Initially, see figure 1.1(a), the only `Accepted` point is the initial point, with $u_{ij} = 0$. The u-value for all other grid points is set to a very large value.

`step1` We determine the neighborhood of the `Accepted` points, see the blue points in figure 1.1(a), and add them to the `NarrowBand` list. For each point in the `NarrowBand` list we solve the eikonal equation (i.e., the quadratic or degenerate linear equation). We choose the grid point in the `NarrowBand`-list with the smallest travel time u (indicated by the circles symbol in figure 1.1(a)) and transfer the corresponding grid point from the `NarrowBand`- to the `Accepted`-list. The associated value of the travel time is the final value.

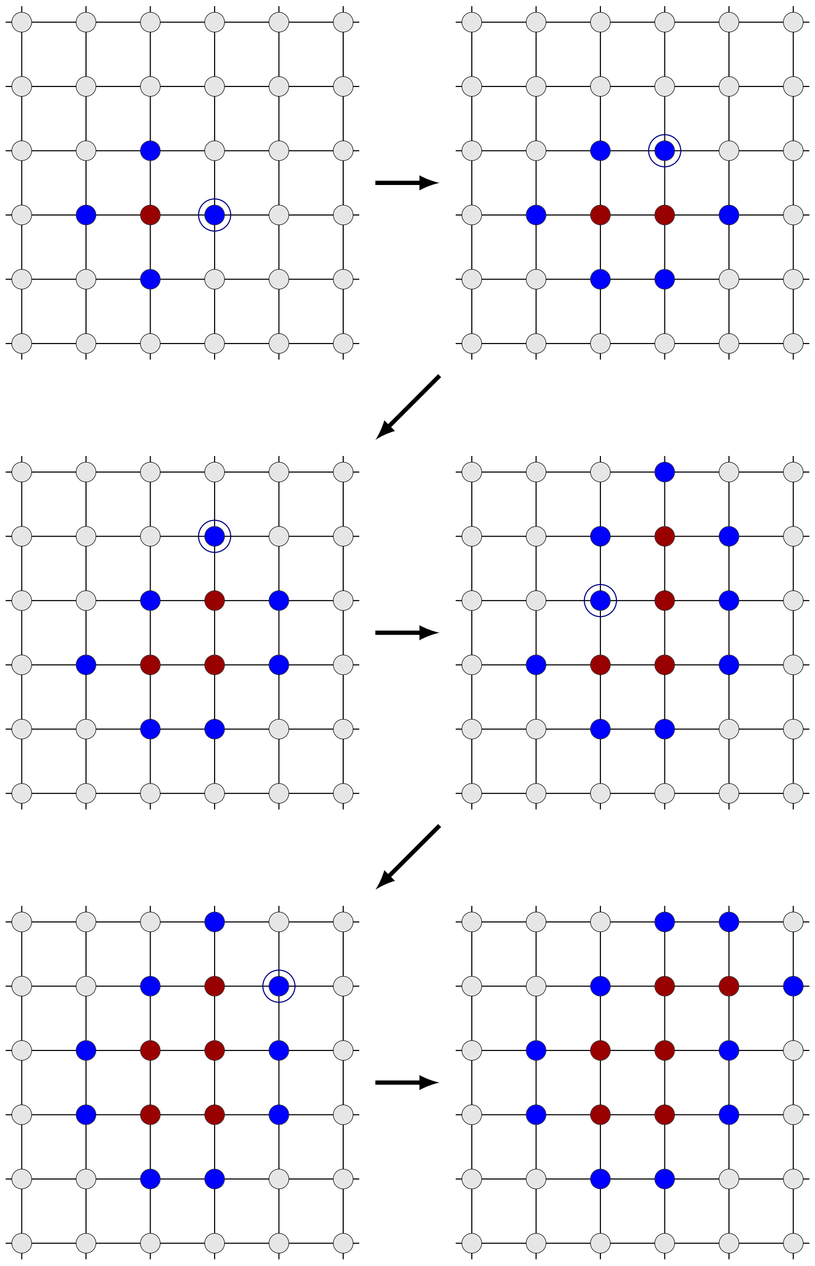


Figure 1.1: Sketch of marching a distance-function front from an initial condition based on the eikonal equation. Color code: (gray) UnVisited grid points, (red) visited and Accepted grid points, (blue) grid points in the NarrowBand area around Accepted grid points, (blue circled) NarrowBand-point with the minimum distance-function value (will be Accepted point in the next step).

step2 We augment the NarrowBand-list by the neighboring point of the newly added grid point, compute the eikonal value u for the new neighbors and continue with the algorithm above (see figure 1.1(b-f)).

step3 The algorithm terminates, when no neighboring points of the Accepted grid points can be found.

The code listing for the python-implementation is listed below. It consists of three functions: the first develops an index list between every point of the computational domain and its Northern, Southern, Western and Eastern neighbors; the second solves the local eikonal equation for a five-point stencil consisting of the central and the Northern, Southern, Western and Eastern points; the third function advances the NarrowBand-front from an initial condition to its conclusion.

```python
import numpy as np
import scipy as sp
import time

import matplotlib
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

def makeIndexList(N,M):
    # running index of grid points
    C   = np.arange(1,N*M+1).reshape(N,M)
    # zero-padding
    CN  = np.zeros((N+2,M+2),dtype=np.int)
    CN[1:1+N,1:1+M] = C
    # shift to get N,S,W,E neighbors
    C_N = CN[1:1+N,2:M+2].flatten()
    C_S = CN[1:1+N,0:M].flatten()
    C_E = CN[2:N+2,1:M+1].flatten()
    C_W = CN[0:N,1:M+1].flatten()
    # rearrange into neighbor list
    CN  = np.array(zip(C_N,C_S,C_E,C_W))
    C0  = C.flatten()
    # initialize
    iA  = np.ones(N*M, dtype=int)  # accepted points = 0, not accepted =
    iC  = np.zeros(N*M, dtype=int) # close points = 1, not close = 0
    return C0,CN,iA,iC
```

This function establishes the four neighbors of a given point in the grid. It is compiled into a zip-list containing the grid indices of the four neighbors. This way, it simplifies the code when looking for the neighbors of a given point.

```python
def solveEik2 (uN,uS,uE,uW,dx,dy,f):
    # initialize
    u1   = 1.e38
    u2   = 1.e38
    u3   = 1.e38
    # grid parameters
    dx2 = dx*dx
    dy2 = dy*dy
    # minimum in each direction
    Tx   = min(uE,uW)
    Ty   = min(uN,uS)
    # case 1: no x-neighbor
    if (Tx > 1.e37):
        u1 = Ty + dy/f
    # case 2: no y-neighbor
    if (Ty > 1.e37):
        u2 = Tx + dx/f
    # case 3: both x-neighbor and y-neighbor are Alive
    if ( (Tx < 1.e38) and (Ty < 1.e38) ):
        alpha = -2*(dy2*Tx + dx2*Ty)/(dx2 + dy2)
        beta  = (dy2*Tx*Tx + dx2*Ty*Ty - dx2*dy2/(f*f))/(dx2 + dy2)
        discri = alpha*alpha - 4*beta
        if (discri > 0):
            u3 = (-alpha + np.sqrt(discri))/2
    return min(u1,u2,u3)
```

This function solves the local eikonal equation, based on the quadratic equation above. For the special cases of no x-neighbor or no y-neighbor, a linear equation is substituted.

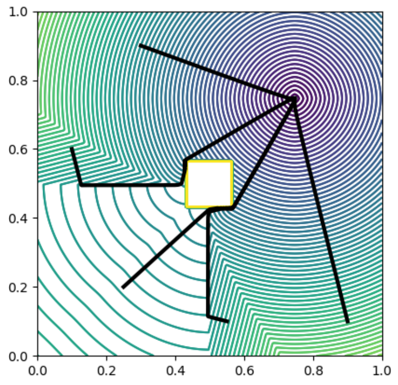
```
def eikonal(f,N,M,dx,dy,xS,yS):
    # make index list C0 and neighbor list CN
    C0,CN,iA,iC = makeIndexList(N,M)
    # initialize narrow-band arrays
    iNB       = np.empty(0, dtype=int)
    NB        = np.empty(0, dtype=float)
    # domain size
    Lx        = (N-1)*dx
    Ly        = (M-1)*dy
    # starting point
    nS        = int(np.floor(xS/Lx*N))
    mS        = int(np.floor(yS/Ly*M))
    kk        = (nS-1)*M + mS
    # initialization
    uBig      = 1.e38
    u         = np.ones(N*M)*uBig
    u[kk-1]   = 0
    iA[kk-1] = 0
    # declare all obstacles (with f=0) "accepted"
    # so they won't be updated
    iA       = iA*f
    iA       = np.array((iA!=0), dtype=int)

    # main loop for eikonal equation
    ic = 0
    while ((len(iNB)>0) or (ic==0)):
        # compute the non-accepted neighbors
        nei   = CN[kk-1]         # (1) get neighbors
        nei   = nei*iA[nei-1]    # (2) eliminate accepted points
        nei   = nei[nei!=0]      # (3) eliminate zeros
        neiC  = nei*iC[nei-1]    # (4) check for already close neighbors
        neiC  = neiC[neiC!=0]
        for i in range(len(neiC)): # (5) eliminate already close neighbor
            ii  = neiC[i]
            jj  = np.where(iNB==ii)[0][0] # search for already close neig
            iNB = np.delete(iNB,jj)       # eliminate them in iNB and NB
            NB  = np.delete(NB,jj)
        iC[nei-1] = 1            # flag new neighbors in close-array
        # compute the u for the indices in nei
        for i in range(len(nei)):
            ii  = nei[i]
            iNB = np.append(iNB,ii)
            [No,So,Ea,We] = CN[ii-1]
            uNo = (1.e37 if (No==0) else u[No-1])
            uSo = (1.e37 if (So==0) else u[So-1])
            uEa = (1.e37 if (Ea==0) else u[Ea-1])
            uWe = (1.e37 if (We==0) else u[We-1])
            ff  = f[ii-1]
            uu  = solveEik2(uNo,uSo,uEa,uWe,dx,dy,ff)
            NB  = np.append(NB,uu)
        # determine the minimum u-value in narrow band
        umin,imin = NB.min(),NB.argmin()
        # next index
        kk       = iNB[imin]
        # delete from narrow band
        NB       = np.delete(NB,imin)
        iNB      = np.delete(iNB,imin)
        # update u, iA and iC
        u[kk-1]  = umin
        iA[kk-1] = 0
        iC[kk-1] = 0
```

```python
        ic       += 1
    return u.reshape(N,M)
```

This routine is the core routine of the eikonal solver. It tracks multiple lists: the Accepted-point list `iA`, which is zero for accepted points and one for unaccepted, the Close-point list `iC`, which is one for points in the NarrowBand and zero otherwise, `iNB` for a list of indices in the NarrowBand, and `NB` for the corresponding u-value (a preliminary value that is not yet accepted). When determining the neighbors of new points, we have to eliminate zero indices (since they indicate either obstacles or the boundaries of the domain), but also previously Close points (since they have to be recomputed). Once the proper neighbor list has been compiled, the u-values are determined, and the local eikonal equation is solved (either in full quadratic form, or in one of its degenerate linear form). The minimum u-value in the `NB` is then determined: this value is then eliminated from the neighbor lists `iNB` and `NB` and transferred from the Close-list `iC` to the Accepted-list `iA`. This concludes one iteration. The iterations continue until no more neighbors can be found. In this case, all grid points have been updated and Accepted.

The main code is listed below. It tests the eikonal code on a variety of speed fields. In addition, an integration routine for tracing geodesic lines is included that allows us to determine minimal paths in the speed fields (case 0 for a test problem, cases 1 and 2 for a labyrinth).

```python
if __name__ == '__main__':

    kstride = 20                # points between parallel walls
    N   = 15*kstride+1          # number of rows
    M   = N                     # number of columns

    # domain parameters
    Lx = 1.                     # domain length in x
    Ly = 1.                     # domain length in y

    # mesh initialization
    dx   = Lx/(N-1)
    dy   = Ly/(M-1)
    x    = np.linspace(0,Lx,N)
    y    = np.linspace(0,Ly,M)
    Y,X  = np.meshgrid(y,x)

    # slowness field
    f0   = np.ones((N,M), dtype=float)
    f0[:int(N/2),:int(M/2)]  = 4.
    f0[130:170,130:170]      = 0.
    f     = f0.flatten()

    # starting points and other parameters
    xs     = 0.75
    ys     = 0.75
    cclip  = 1
    x0     = np.array([0.55, 0.9, 0.25, 0.1, 0.3])
    y0     = np.array([0.1,  0.1, 0.2,  0.6, 0.9])
    ncont  = 80
    nsteps = 500

    # solve eikonal equation
    t0 = time.time()
    u  = eikonal(f,N,M,dx,dy,xs,ys)
    t1 = time.time()
    print 'eikonal:  total time = ',t1-t0

    # graphics
    plt.figure()
    u1 = 0
    u  = np.clip(u,0.,cclip)
    u2 = u.max()
    du = (u2-u1)/ncont
    plt.contour(X,Y,u,levels=np.arange(u1,u2,du))
    CS = plt.contour(X,Y,u,levels=np.arange(u1,u2,du))

    for i in range(len(x0)):
        x00   = x0[i]
        y00   = y0[i]
        t0    = time.time()
        xT,yT = compGeodesics(u,dx,dy,x,y,x00,y00,nsteps)
        t1    = time.time()
        print 'geodesic: total time = ',t1-t0
        plt.plot(xT,yT,'k',linewidth=3)

    plt.axis('image')
    # plt.savefig('eik.png')
    plt.show()
```



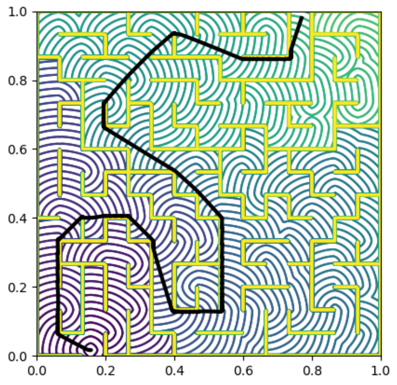


Figure 1.2: (left) Distance function: solution of the eikonal equation, using the Fast Marching Mathod, for a given speed field. Indicated are also geodesic lines in this speed fields, starting from different initial conditions and ending at the initial set $\mathscr{I}$. (right) Distance function and geodesic path for a maze.

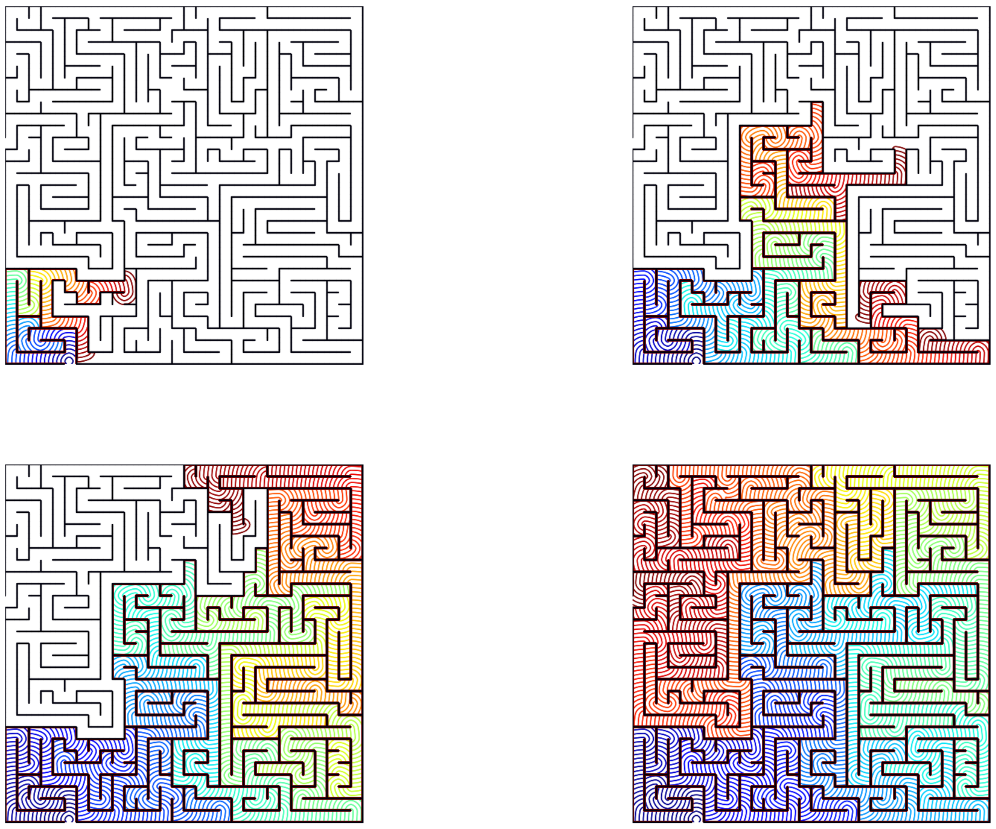


Figure 1.3: Propagation of distance function using the eikonal solver.

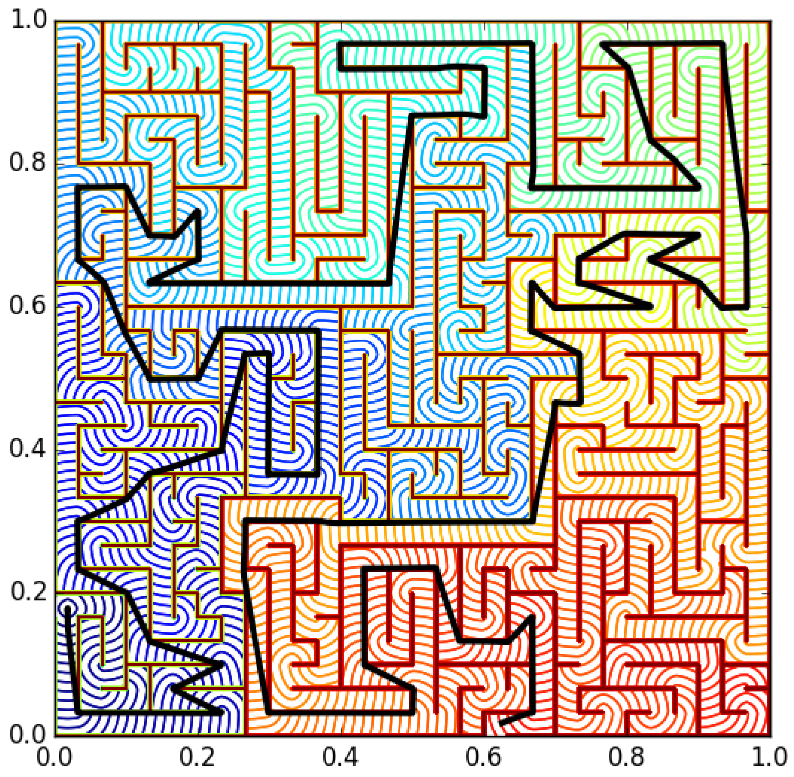


Figure 1.4: Distance function through a more complicated maze, together with a geodesic path.