Geodesic Computations on Surfaces

Karthik Krishnan

June 27, 2013

Abstract

The computation of geodesic distances on a triangle mesh has many applications in geometry processing. The fast marching method provides an approximation of the true geodesic distance field. We provide VTK classes to compute geodesics on triangulated surface meshes. This includes classes for computing the geodesic distance field from a set of seeds and to compute the geodesic curve between source and destination point(s) by back-tracking along the gradient of the distance field. The fast marching toolkit [3] is internally used. A variety of options are exposed to guide front propagation including the ability to specify propagation weights, constrain to a region, specify exclusion regions, and distance based termination criteria. Interpolators that plug into a contour widget, are provided to enable interactive tracing of paths on meshes.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3415]

Contents

1 Theoretical background

The computation of geodesic distances on a triangle mesh has many applications in geometry processing. The fast marching method provides an approximation of the true geodesic distance field. This geodesic distance to a collection of points satisfies a non-linear differential equation, namely the Eikonal equation. The viscosity solution $\phi(x,y)$ of the Eikonal equation given by

$$||\nabla\phi|| = F \tag{1}$$

is a weighted distance map from a set of initial (seed) points, where the value of ϕ is provided, or typically assumed to be 0. The weights are given by the function $F(x,y)$ which is a scalar positive function. In the special case where the function $F(x,y)$ is a constant, ϕ can be interpreted as the distance function to the set of seeds. The level set curve, C_t, defined as points on the front of the function ϕ at time t propagates following the evolution equation

$$\frac{\delta}{\delta t}C_t(x,y) = \frac{n_{xy}}{P(x,y)} \tag{2}$$

where n_{xy} is the exterior unit normal vector to the curve at the point (x,y). The function $F(x,y) = 1/P(x,y)$ is the propagation speed of the front, C_t.

Efficient numerical solutions were introduced by Sethian [4] and adapted by Kimmel [1] on triangulated manifolds:

$$||\nabla_M\phi|| = F \tag{3}$$

where $\nabla_M\phi$ is the gradient on the manifold, M. For further details, the reader is referred to [4, 1, 3].

A geodesic path between two points may be computed from the distance map, ϕ to one of the points, by performing a gradient descent on the function ϕ.

2 VTK Interface

The VTK classes utilize the fast marching library [2] from Gabriel Peyre provided under a BSD license. The VTK classes act as a wrapper around the library providing the bridge between datastructures and callbacks in both libraries. This section serves as a user guide for the VTK classes.

2.1 Geodesic distances - vtkFastMarchingGeodesicDistance

In the simplest use case, `vtkFastMarchingGeodesicDistance` takes in a `vtkPolyData` reprsentative of a triangle mesh and produces as output the same polydata with point attributes containing the distance field from a seed, or set of seeds. It derives from an abstract base class `vtkPolyDataGeodesicDistance`, which inturn derives from `vtkPolyDataAlgorithm`.

Propagation weights

By default the filter assumes a constant propagation speed. However propagation weights may be explicitly specified by the user using `SetPropagationWeights`. This takes as argument, a float/double array, with as many entries as the number of vertices on the mesh. For instance the the mesh curvature may be used to propagate quickly in areas of low curvature and slowly in areas of high curvature. Note that the propagation weights specified must be strictly positive for fast marching to be numerically stable.

Exclusion Regions

Optionally, exclusion regions may be specified via `SetExclusionPointIds(vtkIdList*)`. Vertices with ids that are in the exclusion list are ommitted from inclusion in the fast marching front. This can be used to prevent bleeding into certain regions. If the specified point ids form a closed, fully connected loop, this effectively serves as an exclusion region boundary. Conversely, it can be used to confine fast marching to a specific region.

Termination Criteria

The fast marching may be prematurely terminated via any of the following optional stopping criteria. These are:

- *Distance stop criterion*: Propagation stops if any portion of the front has traversed more than the specified distance from the seed(s). See `SetDistanceStopCriterion(float)`.
- *Destination vertex stop criterion*: Propagation stops if any point in the front arrives at the user supplied destination vertex id(s). See `SetDestinationVertexStopCriterion(vtkIdList*)`. This strategy is used when computing the geodesic path interactively using the contour line interpolator (described later).
- *Boundaries*: If a boundary is specified via `SetExclusionPointIds` forming a closed boundary around the seeds, propagation terminates when all vertices contained within the boundary have been visited.

Events

The filter reports IterationEvents. It does not report progress events, since its not possible to determine when the front might terminate.

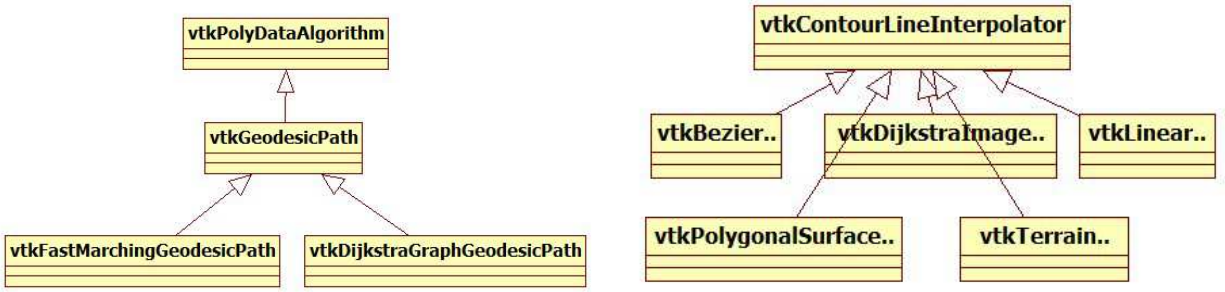
Miscellaneous

Other options include (a) querying the maximum distance marched by the front (b) setting the outside value (for vertices that may not been visited) (c) specifying the distance field name and the ability to turn off its generation (for instance computing the path need not allocate storage for a field on each vertex)

2.2 Tracing geodesic paths

The class `vtkFastMarchingGeodesicPath` (fig. 1(a)) computes geodesic paths on triangle meshes. It takes as input a triangle mesh, a set of destination vertex ids and a source vertex id. The geodesic path is generated by traversing along the gradient of the distance field to arrive at a destination vertex.

Internally an instance of `vtkFastMarchingGeodesicDistance` is used, and the same options that guide/control front propagation can be used when computing the path. The distance field computation is performed such that it terminates when the target seed has been reached so that it is localized and fast enough for interactive tracing. The path extraction alone has an $O(N)$ complexity, where N is the number of vertices in the path. The front propagation takes $O(mlogm)$ where m is the number of vertices contained in a circular patch of radius spanning the distance between the source and



(a) Geodesic paths may be computed from a single source to destination vertices using the Dijkstra algorithm or by gradient backtracking of the distance field generated by fast marching.

(b) The vtkPolygonalSurfaceContourLineInterpolator2 is an interpolator that can be used by a contour widget to trace paths on surfaces. These can use either instance of the geodesic path shown on the left.

Figure 1

destination vertex. When the source and destination points are no more than a few thousand vertices apart the path extraction is done in milliseconds and is suitable for real time tracing and editing of paths on meshes.

Paths may be generated so that they traverse between mesh vertices (linear interpolation) or may be constrained on mesh vertices. The method `SetInterpolationOrder` allows one set the path interpolation order. A zeroth order path passes through vertices of the mesh. A first order path passes in between vertices (linear interpolation is performed on the triangle edges). Each point in the first order path is guarenteed to lie on a triangle edge.

A maximum path length may optionally be specified, in which case, the gradient based back-tracking may stop prematurely (once the specified length is exceeded).

The point ids of the vertices on the mesh (in case of zeroth order interpolation) or closest vertices on the mesh (in case of first order interpolation) that connect the source to the destination point may be queried. This can be used for interactive surface clipping.

The class `vtkPolygonalSurfaceContourLineInterpolator2` enables interactive tracing on polygonal surfaces. It is one of a growing family (see fig. 1(b)) of contour line interpolators that make the `vtkContourWidget` a powerful tool enabling contouring subject to a variety of constraints. It can replace `vtkPolygonalSurfaceContourLineInterpolator`, since it retains its public API but provides added functionality.

This interpolator is meant to be used in conjunction with a `vtkPolygonalSurfacePointPlacer` which constrains nodes dropped using a contour widget to vertices that lie on the surface of the mesh. As points are interactively placed using the widget, it computes the path joining these nodes on the 2D manifold. There is a weak coupling between the classes allowing the picked point id to be queried by the interpolator from the placer, so that picking (which for large meshes can be expensive) may be performed only once. The path may internally be generated using the Dijkstra algorithm, or by fast marching as described above. Fast marching with first order interpolation, unlike Dijkstra, allows the generation of smooth paths. Weighted constraints (as described above) may be used to preferentially displace the path based on features on the surface.

2.3 Computation time

The complexity of this method is $O(NlogN)$, where N is the number of mesh vertices. Hoppes et. al. compare, the runtime performance of this implementation with their and other implementations. The results are available in Table 1 [5]. On a mesh with 10000 vertices, fast marching using this method is reported to take 3.5s on a Pentium M 1.6GHz with 1GB RAM.

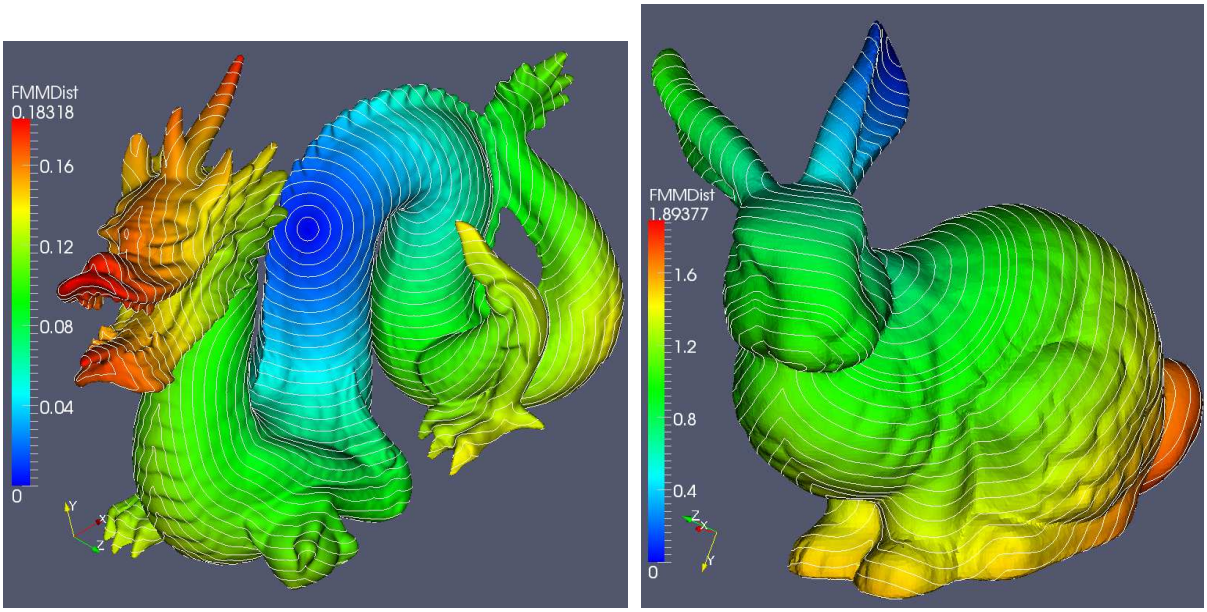


Figure 2: Geodesic distance using fast marching.

3 Examples

A subset of the scenarios discussed above are illustrated below with examples, included with this package. The Stanford Bunny (provided with this package) is used for all examples.

3.1 Computing Geodesic distance

The program `GeodesicDistanceExample.cxx` uses the `vtkFastMarchingGeodesicDistance` to compute a geodesic distance field on a surface. Run it with the arguments

```
Bunny.vtp BunnyWithDistField.vtp
```

Pick any point on the displayed surface. The resulting image is the geodesic distance from the chosen point (see Fig. 2) colored by the distance from the chosen point. The point data array `FMMDist` is populated with the distances.

```
vtkFastMarchingGeodesicDistance *gd = vtkFastMarchingGeodesicDistance::New();
gd->SetInputConnection(polydata);
gd->SetFieldDataName("FMMDist");
```

3.2 Specifying distance termination criteria

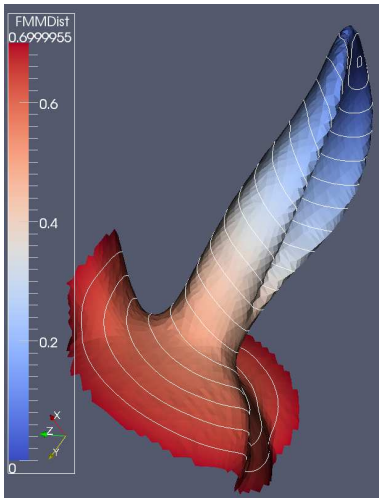
Run the example as before but with the arguments

```
Bunny.vtp out.vtp --maxDist 0.7
```

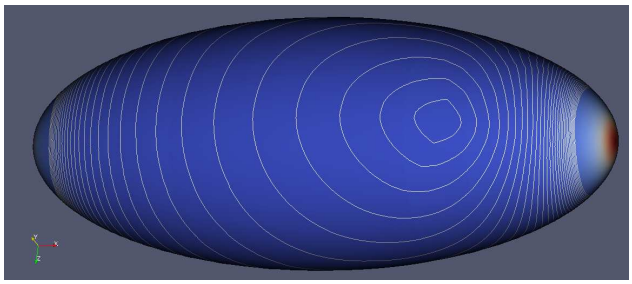
This specifies distance threshold to terminate the front propagation by adding a line to the effect of

```
gd->SetDistanceStopCriterion(0.7);
```

The result is as shown in Fig. 3(a). The strategy can also be used to extract the surface within a given distance of the seed(s).



(a) Fast marching upto a distance threshold of 0.7 mm



(b) Propagation weights are specified as the curvature of the point ids. This is observed in the assymetry in the iso-lines, which travel faster to the shorter axis of the point of least curvature on the ellipsoid.

Figure 3

3.3 Propagation Weights

Propagation weights may be specified as a data array. For instance one can pass the mesh through the `vtkCurvatures` filter and specify the curvature data array as the propagation weight for the vertices.

```
gd->SetPropagationWeights(dataArray);
```

Run the same example with the arguments

```
Ellipsoid.vtp output.vtp --propagationWts CurvWt
```

This result is shown in Fig. 3(b). Note that, as expected, the field traverses faster towards the left than towards the right.

3.4 Exclusion Regions / Boundaries

Exclusion regions may be specified via a list of PointIds comprising the exclusion regions. Note that if the exclusion regions, comprise a closed loop, around the seeds, this effectively becomes a boundary for the front. The following example demonstrates this. We run the same example with the arguments

```
Bunny.vtp output.vtp --exclusionContour
```

Once the render window comes up, drop points on the surface to trace a contour, looping back t to the first one, so as to close the contour. After, the contour is closed, pick any point on the surface within the contour. Here we extract the bunny's head (see Fig. 4 by tracing a contour around its neck and then pass these point ids as the exclusion region :

```
// Query the traced contour's point ids
dijkstraContourInterp->GetContourPointIds(contourRepresentation, exclusionPtIds);
gd->SetExclusionPointIds(ids); // Provide the ids as an exclusion region
```

You may peruse the example which also demonstates the use of a `vtkContourWidget` in conjunction with a `vtkDijkstraContourLineInterpolator` to trace the boundary contour.

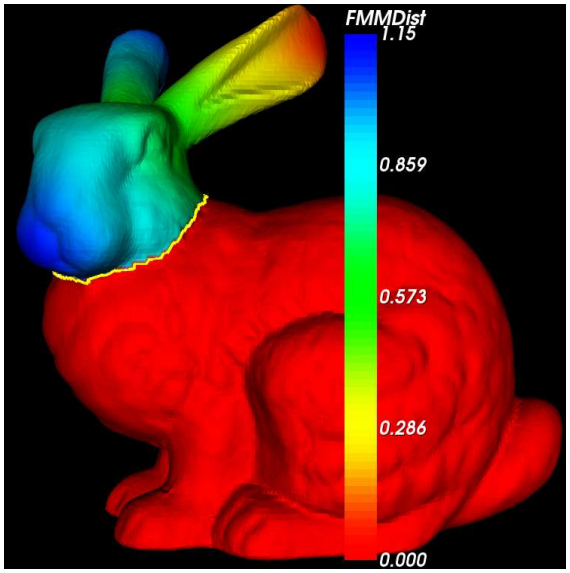


Figure 4: Exclusion regions or boundaries may be specified.

3.5 Tracing geodesic paths

The program `GeodesicPathExample.cxx` demonstrates tracing a path interactively using a `vtkContourWidget` on a surface mesh. Its arguments are

```
surface.vtp [Method (0=Dijkstra,1=FastMarching)] [InterpOrder (0=NN,1=Linear)] [heightOffset]
```

Fig. 5 shows the paths generated using various methods. Typically, the path computed using linear interpolation with fast marching is shorter than that computed with zeroth order interpolation using Dijkstra or with fast marching. Note that the zeroth order paths using both methods (Dijkstra or FastMarching) need not be identical (as may also be observed in the figure). The Dijkstra method provides the true shortest zeroth order path, while that obtained from fast marching is (a) an approximation (b) clamps the first order path to its closest vertices and could be longer. The computation times of both methods are nearly the same. The heightOffset argument allows one to displace the path in the direction of the surface normal.

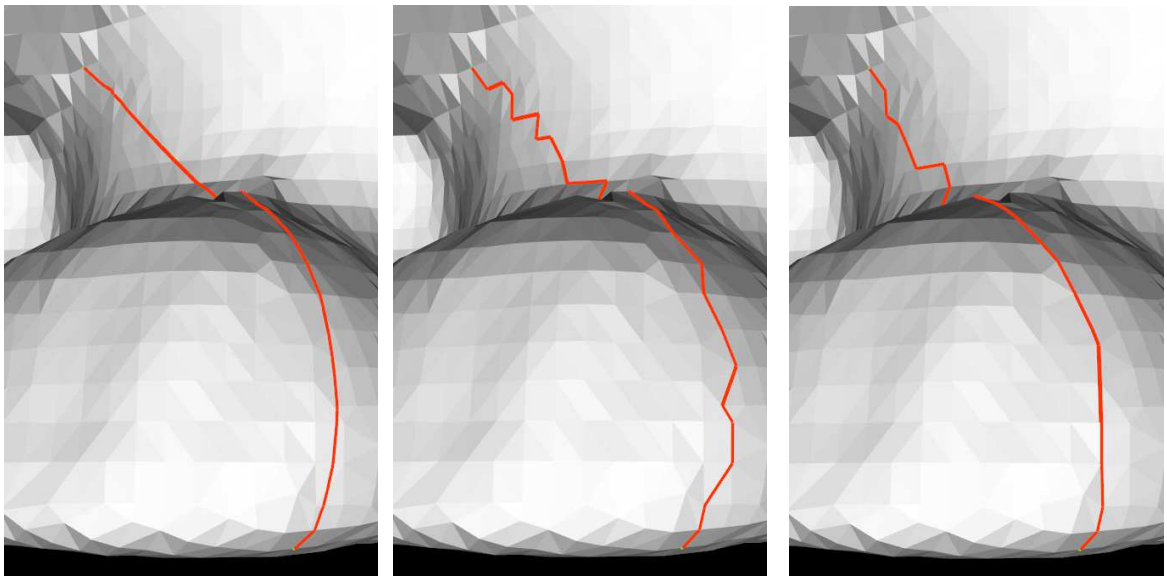
```
vtkNew<vtkPolygonalSurfaceContourLineInterpolator2> interpolator;

// Add all the surfaces to which the path is to be constrained
interpolator->GetPolys()->AddItem( surfaceMesh );

// Set the method to compute the geodesic (fast marching or dijkstra)
interpolator->SetGeodesicMethodToFastMarching();
interpolator->SetInterpolationOrder(1); // use first order interp
rep->SetLineInterpolator(interpolator.GetPointer());

// Setup the point placer used to constrain picking on the surface
vtkNew<vtkPolygonalSurfacePointPlacer> pointPlacer;
pointPlacer->AddProp(actor.GetPointer());
pointPlacer->GetPolys()->AddItem( surfaceMesh );
rep->SetPointPlacer(pointPlacer.GetPointer());

// Snap the contour nodes to the closest vertices on the mesh
pointPlacer->SnapToClosestPointOn();
```



(a) Fast marching first order interpolation

(b) Fast marching zeroth order interpolation

(c) Dijkstra (zeroth order interpolation)

Figure 5: Geodesic paths generated using various methods

References

[1] R. Kimmel and J. Sethian. Computing geodesic paths on manifolds. In *Procedings of the National Academy of Sciences*, volume 95, July 1998. 1, 1

[2] G. Peyre. `http://geodesics4meshes.googlecode.com`. 2

[3] Gabriel Peyre and Laurent D. Cohen. Geodesic remeshing using front propagation. *International Journal of Computer Vision*, pages 145–156, 2006. (document), 1

[4] J. A. Sethian. Fast marching methods. *SIAM Review*, 41:199–235, 1998. 1, 1

[5] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. Gortler, and H. Hoppe. Computing geodesic paths on manifolds. 24, 2005. `http://research.microsoft.com/en-us/um/people/hoppe/geodesics.pdf`. 2.3