GEODESIC DISTANCE from HEAT METHOD

By - Vandita Shukla
Student Number - 17141590

(*Disclaimer:* This report was initially submitted as this module's project and is being resubmitted as per the LSA brief. This report is entirely my own work and an interpretation of the Heat Method paper chosen for the project.)

1 Introduction

In this project, I have worked on the paper "The Heat Method for Distance Computation" by Keenan Crane, Clarisse Weischedel, and Max Wardetzky. The paper explores a very interesting approach to the calculation of geodesic distances on mesh surfaces using a very elegant pipeline. The results are shown in images and I implemented this algorithm in Python using Trimesh objects of triangle meshes. My partner in this project Ms. Furat Aljishi plans to implement the Fast Marching for the Geodesic Distance computation (hasn't contributed to this report or project implementation due to a different working timeline owing to SoRA). I have done this project individually.

2 Paper Summary & Heat Flow Algorithm

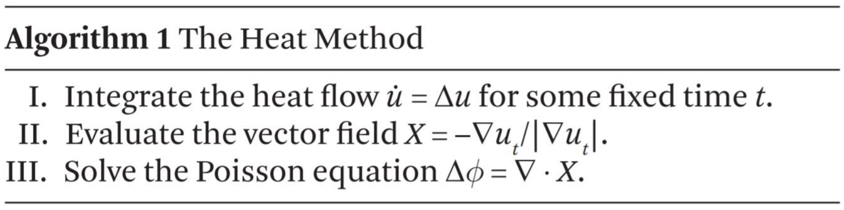
The beauty of the application of heat flow in geodesic distance computation lies in the fact that it is applicable to different kinds of meshes including polyhedral meshes, triangular meshes, point clouds et cetera as long as the gradient, divergence, and the Laplace operator can be applied. The functionality of this algorithm is extensive as factorization can be applied to pre-calculated matrices and spatial discretization like laplacian discretization can be applied as well. I will now go on to expand on the algorithm.

The main advantage that the heat method offers is the high speed without any compromise on the accuracy of the system. The heat method can work with any form of meshes, grids, or point clouds due to its applicability to all discretizations. Earlier methods like fast marching for geodesic distance calculations required the expensive characteristically serial computation on every new point or gamma subset. Varadhan's formula which provides a mathematical formula

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

Fig 1: Varadhan formula for the geodesic distance between x and y and $k_{t,x}$ (y) is the heat kernel

for the association of heat kernel and geodesic distance is interpreted in a generalized way for applicability. The formula states that on a Riemannian manifold, the distance between two points can be calculated using 'point transformation' of the heat kernel, stemming from the science of heat travel on a surface. However, this point transformation at individual heat points is complicated to recover, so we work with the gradients, any function parallel to the gradient instead of the actual geodesic distance.



Algorithm 1 The Heat Method

```
I.   Integrate the heat flow $\dot{u} = \Delta u$ for some fixed time $t$.
II.  Evaluate the vector field $X = -\nabla u_t / |\nabla u_t|$.
III. Solve the Poisson equation $\Delta \phi = \nabla \cdot X$.
```

Fig 2: The Heat Method (Crane, Weischedel and Wardetzky, 2017)

STEP 1:

The heat value at any surface point on the mesh is considered as 1 (source) and lies between 0 & 1 as it moves away from the source vertex.

Using time step *t* we solve for u at time t on triangle mesh using the formula-

$$(A - t\,Lc)\,u \;=\; u_0$$

A = Diagonal vertex area matrix

Lc = $A^{-1}Lc$ is the discrete Laplace Beltrami

u_0 = The value of this vector is 1 at the origin vertex.

STEP 2:

The vector field X is evaluated by finding the gradient operator on u and normalizing. The method for calculating X on the mesh is given as -

$$X = -\nabla u_t / |\nabla u_t|$$

To calculate gradient on u-

$$\nabla u = \frac{1}{2A_f} \sum_i u_i \left(N \times e_i \right),$$

Here

A_f = triangle area

N = normal of the face

u_i = value of u at vertex opposite to the edge in use

e_i = vector of edge (anti-clockwise)

Only the orientation of the gradient is of importance hence the normalization for X.

STEP 3:
We solve for phi, which contains the approx distance to the source vertices, on the last step by calculating a vector b which contains the integrated divergences and using pre-factored Lc with the formula -

$$L_C\phi = b$$

Note:
According to the paper, an important consideration regarding this whole process is the time step t. Time t for different meshes can be a different complex analysis, therefore, a general formula for time step as suggested in the paper is

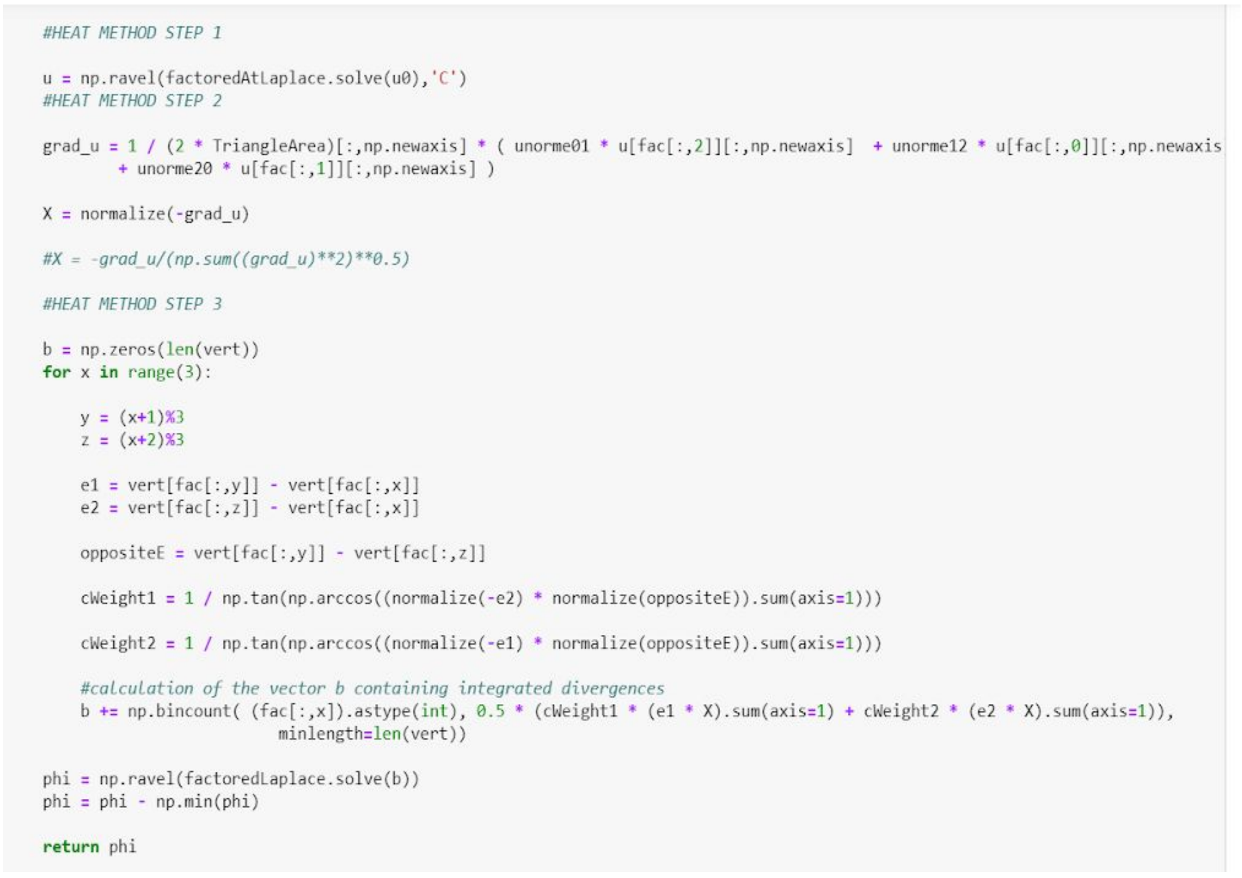
$$t = mh^2$$

Where m is always greater than 0 and h in the implementation is calculated to be the average edge length.

3 MESHES and ALGORITHM execution

I used the sample meshes provided in the tutorials for implementing the algorithm in Python and used Trimesh, Numpy, and Scipy. The steps for running the algorithm on a mesh of choice are -

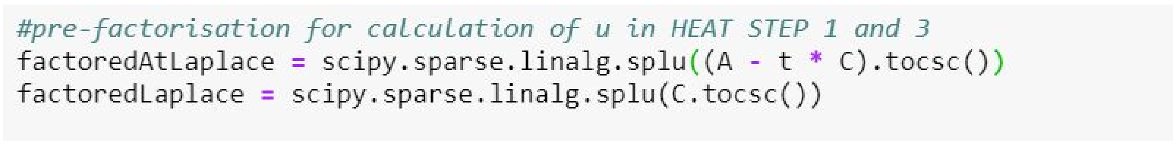
- Load mesh of choice using **trimesh.load()**
- Lc and A are calculated using the function **cotanlaplacebeltrami(**mesh**)**
- Run the heat flow algorithm to calculate phi using **geodesicdistance(**mesh, Lc, A, constant factor for time step, source index**)**



```python
#HEAT METHOD STEP 1

u = np.ravel(factoredAtLaplace.solve(u0),'C')
#HEAT METHOD STEP 2

grad_u = 1 / (2 * TriangleArea)[:,np.newaxis] * ( unorme01 * u[fac[:,2]][:,np.newaxis]  + unorme12 * u[fac[:,0]][:,np.newaxis
        + unorme20 * u[fac[:,1]][:,np.newaxis] )

X = normalize(-grad_u)

#X = -grad_u/(np.sum((grad_u)**2)**0.5)

#HEAT METHOD STEP 3

b = np.zeros(len(vert))
for x in range(3):

    y = (x+1)%3
    z = (x+2)%3

    e1 = vert[fac[:,y]] - vert[fac[:,x]]
    e2 = vert[fac[:,z]] - vert[fac[:,x]]

    oppositeE = vert[fac[:,y]] - vert[fac[:,z]]

    cWeight1 = 1 / np.tan(np.arccos((normalize(-e2) * normalize(oppositeE)).sum(axis=1)))

    cWeight2 = 1 / np.tan(np.arccos((normalize(-e1) * normalize(oppositeE)).sum(axis=1)))

    #calculation of the vector b containing integrated divergences
    b += np.bincount( (fac[:,x]).astype(int), 0.5 * (cWeight1 * (e1 * X).sum(axis=1) + cWeight2 * (e2 * X).sum(axis=1)),
                     minlength=len(vert))

phi = np.ravel(factoredLaplace.solve(b))
phi = phi - np.min(phi)

return phi
```

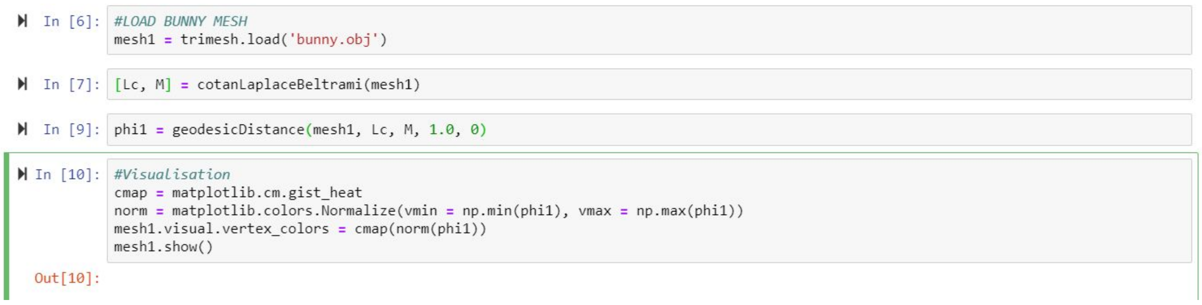
Fig 3: Main steps for Heat Method

- For optimization of the code, I used pre-factored matrices A - tLc and Lc for the first and third step of the heat flow algorithm. Cholesky decomposition of matrices into upper and lower triangular of LL^T makes the solving easier and faster while reducing the errors of the system. For big meshes like dragon, however, it was observed that a slight epsilon addition to the diagonal helps keep the matrix strictly positive definite.



```python
#pre-factorisation for calculation of u in HEAT STEP 1 and 3
factoredAtLaplace = scipy.sparse.linalg.splu((A - t * C).tocsc())
factoredLaplace = scipy.sparse.linalg.splu(C.tocsc())
```

The next figure shows the visualization code -



```
In [6]:  #LOAD BUNNY MESH
         mesh1 = trimesh.load('bunny.obj')

In [7]:  [Lc, M] = cotanLaplaceBeltrami(mesh1)

In [9]:  phi1 = geodesicDistance(mesh1, Lc, M, 1.0, 0)

In [10]: #Visualisation
         cmap = matplotlib.cm.gist_heat
         norm = matplotlib.colors.Normalize(vmin = np.min(phi1), vmax = np.max(phi1))
         mesh1.visual.vertex_colors = cmap(norm(phi1))
         mesh1.show()

Out[10]:
```

For visualization, I used matplotlib and libigl for Python for constructing the isolines on the mesh.

Results-
The following results were obtained as viewed in terms of the heat diffusion obtained -

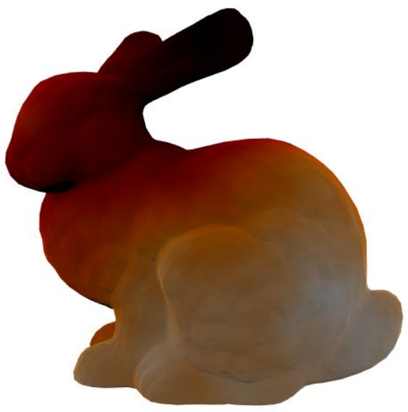


Fig 2 : Heat diffusion in Bunny.obj with m = 1.0 for time step

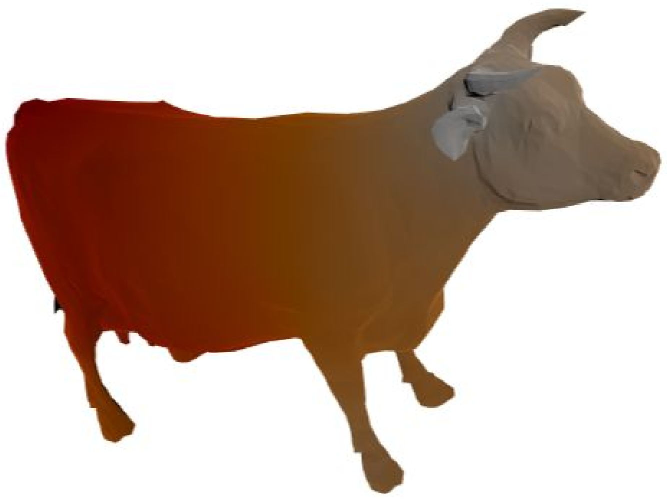


Fig 4: Heat diffusion in Cow_Manifold.obj with m =10 for time step

The visualisation of geodesic distances via Isolines was done through a sin function for phi (phi multiplied with a constant) and could be well observed with an appropriate constant.

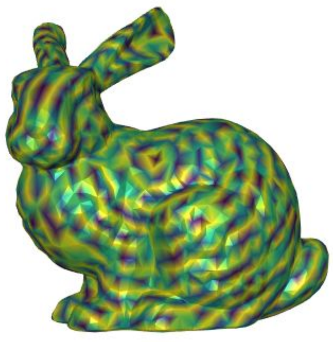


Fig 5: Libigl viewer for colormap of np.sin(phi*np.pi*d=30)

For bigger meshes, the phi required some starkly different arbitrary constant instead of being near d=30 for visualization of isolines. The cow manifold gave a mapping as below for the same function used in the bunny.

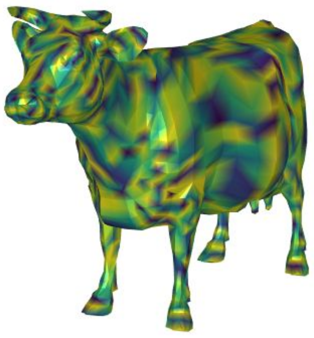


Fig 6: Libigl viewer for colormap of np.sin(phi*np.pi*d=30)