Parallelised Differentiable Straightest Geodesics for 3D Meshes

Supplementary Material

A. Appendix

This document supplements our paper entitled **Parallelised Differentiable Straightest Geodesics for 3D Meshes** by providing further information on our core contributions, limitations and future directions, as well as additional implementation details for our differentiable straightest geodesics algorithm. We then describe the projection integration method we re-implement and show how our differentiation schemes can be used to improve their gradient estimation. We also provide a derivation of the closed-forms we use as benchmarks. Finally, for all our applications (i.e., AGC, MeshFlow, and Mesh-LBFGS) we provide more details and experimental results.

Narrative description of our core contributions. Most modern machine learning tools struggle to correctly operate on surfaces discretised as meshes because the math behind them is slow and hard to differentiate. We bridge this gap by introducing a way to calculate Exponential maps on 3D meshes that is fully compatible with modern machine learning (Sec. 4). We first moved these straightest geodesics algorithm to the GPU, making it up to three orders of magnitude faster than current methods. More importantly, we created two new ways to let gradients flow through the Exponential map: a fast approximation and a high-accuracy version. For the first time, a machine learning model can learn the best way to move across a surface.

To show what these tools can actually do, we applied them to three real-world problems that have traditionally been difficult to solve on 3D surfaces. First, we built Adaptive Geodesic Convolutions (AGC) (Sec. 5.2), which use our differentiable Exponential map to learn exactly how wide or narrow its receptive field should be for every single layer, allowing it to segment complex shapes like human body parts with much better accuracy than models with fixed kernel sizes. Next, we developed MeshFlow (Fig. A.9), a generative model that can transform a learn how to flow distributions defined on a mesh through a single static velocity field. MeshFlow moves sampled points natively along the mesh and is incredibly efficient, being its work roughly $16,000$ times faster and using 97% less memory than Riemannian Flow Matching (RFM) [15]. Finally, we introduced the Mesh-LBFGS optimiser (Sec. 5.4), a high-speed second-order optimiser for meshes. Our optimiser uses the surface's curvature to take much smarter, larger leaps, reaching a perfect distribution of points in far fewer steps.

Limitations & Future Work. In this work, we propose two differentiation schemes for the exponential map of 3D meshes, enabling backpropagation through this operator for the first time on meshes. Both methods are completely independent of the implementation of the exponential map, but each suffers from different limitations. The EP scheme is faster, but can be used only to approximate the Jacobian with respect to the initial vector $\mathbf{v}$, which is still less accurate than GFD. The GFD scheme can accurately differentiate with respect to both $\mathbf{p}$ and $\mathbf{v}$ at the expense of a slightly increased computational cost, caused by the requirement to trace multiple geodesics. While this is not a problem for our parallelised GPU implementation of the geodesic tracing algorithm, it may limit the applicability of alternative exponential maps in a learning framework (e.g., see our experiments with PI in Fig. A.2). Future work shall seek a differentiation scheme capable of achieving computational speeds comparable to EP, while retaining the accuracy of GFD.

Note on differences with Eikonal solvers. The main difference between our method and Eikonal solvers [38, 47] is that while we solve an initial value problem, they solve boundary value problems. Eikonal solvers, solving Hamilton–Jacobi equations, explicitly work on minimising the distance (gradient descent after obtaining the distance field). Distance fields are scalar PDE objects not directly suited for computing exponential maps or parallel transport. Our approach does not provide a global distance field, but directly gives the discrete analogues of the Riemannian operators. Eikonal solvers such as the Deep Eikonal Solvers (DES) compute "shortest" paths whereas we compute "straightest" paths. While on smooth manifolds these coincide (0 covariant acceleration = length of the curve that minimises the path length), on discrete surfaces they do not. This is a geometric difference, which will naturally lead to algorithms computing a different quantity. At first, they look similar, but this can create very different behaviours when examined closely, especially on meshes with non-uniform triangle sizes.

Moreover, naive fast marching (used for Eikonal solvers) is also non-differentiable and several works seek to make it so (like DES - deep eikonal solvers). They are also harder to parallelise as front ordering is naturally sequential (pop-min from wavefront) - ours is embarrassingly parallel (per traced geodesic).

Finally, (i) we directly discretise the geodesic equation itself, whereas Eikonal solvers do so indirectly - as a result, arguably, ours remain closer to the true notion of a "geodesic" whatever it might mean in the discrete setting; (ii) our work also gives a natural connection therefore a parallel transport (which we use in LBFGS); these are harder

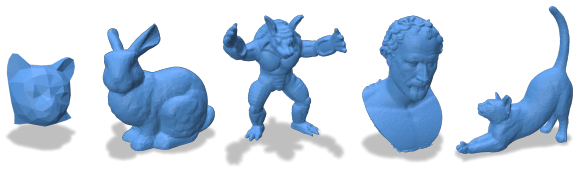


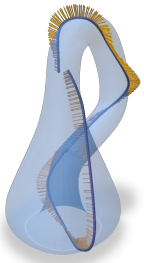
Figure A.1. Meshes used to evaluate our method in Sec. 5.1: cat head (248 faces), bunny (4,968 faces), armadillo (29,170 faces), Demosthenes (240,293 faces), and cat (1,698,248 faces) [79]

to obtain for DES-like methods.

A.1. Additional Details on Our Method

Implementation of the Geodesic Step. While straightest geodesics can be formulated from a purely intrinsic perspective, as mentioned in Sec. 4, by using extrinsic 3D normals and rotations, we can replicate the intrinsic logic with more efficient tensor operations. The geodesic step reported in Alg. 1 and detailed in Alg. A.1 still distinguishes between three possible traces: on a face, across an edge, or across a vertex. On a face, the method uses barycentric coordinates to trace a straight line that terminates within the face when reaching the maximum length or continues until it intersects either a vertex or an edge. On edges, the angle preservation criteria corresponding to the unfolding of the triangles adjacent to the edge is replaced by computing the 3D dihedral angle between the face normals and extrinsically rotating the incoming vector around the common edge. At vertices, after identifying the correct outgoing edge by accumulating intrinsic angles to satisfy $\theta_l = \theta_r$, we determine the outgoing vector by applying a 3D rotation to that edge's vector on the new 3D face plane.

Main differences with SotA implementation. Geometry central processes meshes using a half-edge representation, thus requiring meshes to be orientable. However, in our implementation, we only store triangle adjacencies to trace geodesics, making it compatible with non-orientable meshes, like a Möbius strip or Klein bottle. We here report a geodesic trace on the Klein bottle computed with our implementation while showing also the normals along the geodesic.



Geometry central also does not handle traces crossing vertices, and only considers that the trace will cross edges. This can lead to some imprecisions when using high resolution meshes or low precision floats.

An additional distinction, previously discussed in Sec. 4 lies in the behaviour when encountering a mesh boundary. Instead of terminating the geodesic, we continue tracing along the boundary until the original tracing direction can be resumed. We highlight the essential modifications in blue in Alg. A.2 and Alg. A.3. We show how this can improve convergence performance on meshes with holes in Sec. A.7. In Alg. A.2, the transport error depends on two factors: it measures how far the direction deviates from the target face, and it ensures that the projected direction keeps the point inside that face. If no faces meet this criterion, we select the face adjacent to the hole and trace the geodesic along the edge.

Non-auto-differentiability of straightest geodesics. As mentioned in Sec. 4, proven in [14], and confirmed by our experiments where we let `autograd` automatically differentiate through the exponential map of [51] (Fig. 4), automatic differentiation is not suitable in non-Euclidean settings. Even if we pretend to ignore this fact, all implementations of straightest geodesics are algorithms characterised by discrete combinatorial choices. As such, automatic differentiation cannot even be applied in practice. In Alg. A.1 to A.3 we highlight in red the non-differentiable operations that PyTorch's `autograd` cannot handle automatically. In particular, searching for specific edges/faces and for indices that meet certain conditions would all require

Algorithm A.1 `geodesic_step`

In: mesh $\mathscr{M}$, surface point $\mathbf{p}$, direction $\mathbf{v}$

$(\mathrm{f}, \mathbf{b}) \leftarrow \mathbf{p}$
if $\exists k$ such that $\mathbf{b}_k = 1$ **then**
  return `transport_over_vertex`$(\mathscr{M}, \mathrm{f}, \mathbf{b}, \mathbf{v})$
else if $\exists k$ such that $\mathbf{b}_k = 0$ **then**
  return `transport_over_edge`$(\mathscr{M}, \mathrm{f}, \mathbf{b}, \mathbf{v})$
else
  Let $\mathbf{b}^{\mathbf{v}}$ the barycentric representation of $\mathbf{v}$ in f
  $\lambda \leftarrow \min \left\{ -\frac{\mathbf{b}_i}{\mathbf{b}_i^{\mathbf{v}}} \;\middle|\; i \in [0,2],\ -\frac{\mathbf{b}_i}{\mathbf{b}_i^{\mathbf{v}}} > 0 \right\}$
  $\mathbf{b} \leftarrow \mathbf{b} + \lambda \mathbf{b}^{\mathbf{v}}$
  $\mathbf{p} \leftarrow (\mathrm{f}, \mathbf{b})$
  $\Delta\mathbf{v} \leftarrow \lambda\,(\mathbf{b}_v^{\mathbf{v}}(\mathbf{x}_1 - \mathbf{x}_0) + \mathbf{b}_w^{\mathbf{v}}(\mathbf{x}_2 - \mathbf{x}_0))$
  return $\mathbf{p}, \mathbf{v}, ||\Delta\mathbf{v}||$
end if

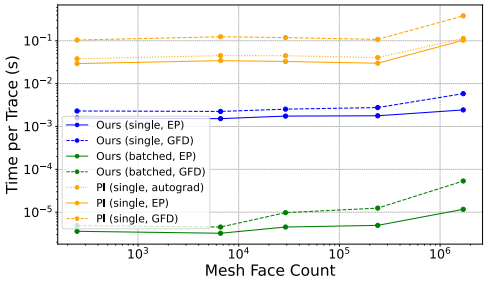


Figure A.2. Mean time per trace for forward and backward pass using our method and PI with different differentiation schemes.

Algorithm A.2 `transport_over_vertex`. (we highlight in pink the non differentiable steps and in blue the hole avoidance modifications).

In: mesh $\mathscr{M}$, face f, barycentric coordinates $\mathbf{b}$, direction $\mathbf{v}$

$(\mathbf{X}, \mathbf{F}) \leftarrow \mathscr{M}$
$\mathrm{x}_0 \leftarrow \mathbf{F}[\mathrm{f}, \mathrm{argmax}_k \mathbf{b}_k]$
if $\mathbf{p}$ is adjacent to a hole **then**
  $\mathrm{f}' \leftarrow \mathrm{argmin}_f$ `transport_error`$(\mathscr{M}, \mathbf{v}, f)$
  $\mathbf{b}' \leftarrow (0,0,0)$
  $k \leftarrow \mathrm{argwhere}_i \mathbf{F}[\mathrm{f}, i] = \mathrm{x}_0$
  $\mathbf{b}'_k \leftarrow 1$
  $\mathbf{p}' \leftarrow (\mathrm{f}', \mathbf{b}')$
  $\mathbf{n}' \leftarrow$ `get_face_normal`$(\mathscr{M}, \mathrm{f}')$
  $\mathbf{v}' \leftarrow \mathbf{v} - \langle \mathbf{v}, \mathbf{n}' \rangle \mathbf{n}'$
  $\mathbf{v}' \leftarrow \frac{\mathbf{v}'}{||\mathbf{v}'||}$
  return $\mathbf{p}', \mathbf{v}', 0$
end if
$\mathrm{x}_1, \mathrm{x}_2 \leftarrow \{\mathbf{F}[\mathrm{f}, \mathbf{b}_i] \mid i \in [0,2],\ i \neq \mathrm{argmax}_k \mathbf{b}_k\}$
$\mathbf{n} \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f})$
$\alpha \leftarrow |$`signed_angle`$(-\mathbf{v}, \mathbf{X}[\mathrm{x}_1] - \mathbf{X}[\mathrm{x}_0], \mathbf{n})|$
if $\alpha > \theta(\mathrm{x}_0)/2$ **then**
  $\alpha \leftarrow |$`signed_angle`$(\mathbf{X}[\mathrm{x}_2] - \mathbf{X}[\mathrm{x}_0], \mathbf{v}, \mathbf{n})|$
  $\mathrm{x}_1, \mathrm{x}_2 \leftarrow \mathrm{x}_2, \mathrm{x}_1$
end if
while $\alpha < \theta(\mathrm{x}_0)/2$ **do**
  if there is no adjacent triangle **then**
    $\mathbf{p} \leftarrow (\mathrm{f}, \mathbf{b})$
    return $\mathbf{p}, \mathbf{v}, 0$
  end if
  $\mathrm{f}' \leftarrow$ `connected_face`$(\mathscr{M}, \mathrm{f}, \mathrm{x}_0, \mathrm{x}_1)$
  $\mathrm{x}_2 \leftarrow$ `next_edge_vertex`$(\mathscr{M}, \mathrm{f}', \mathrm{x}_0, \mathrm{x}_1)$
  $\mathbf{e}_1 \leftarrow \mathbf{X}[\mathrm{x}_1] - \mathbf{X}[\mathrm{x}_0]$; $\mathbf{e}_2 \leftarrow \mathbf{X}[\mathrm{x}_2] - \mathbf{X}[\mathrm{x}_0]$
  $\alpha \leftarrow \alpha + |$`signed_angle`$(\mathbf{e}_1, \mathbf{e}_2, \mathbf{n})|$
  $\mathrm{f} \leftarrow \mathrm{f}'$; $\mathrm{x}_1 \leftarrow \mathrm{x}_2$
end while
$\beta \leftarrow \alpha - \theta(\mathrm{x}_0)/2$
$\mathbf{n} \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f})$
$\mathbf{v} \leftarrow$ `rotate_vector`$(\mathbf{v}, \mathbf{n}, \beta)$
$\mathbf{b} \leftarrow (0,0,0)$
$k \leftarrow \mathrm{argwhere}_i\ \mathbf{F}[\mathrm{f}, i] = \mathrm{x}_0$
$\mathbf{b}_k \leftarrow 1$
$\mathbf{p} \leftarrow (\mathrm{f}, \mathbf{b})$
return $\mathbf{p}, \mathbf{v}, 0$

Algorithm A.3 `transport_over_edge`. (we highlight in pink the non differentiable steps and in blue the hole avoidance modifications).

In: mesh $\mathscr{M}$, face f, barycentric coordinates $\mathbf{b}$, direction $\mathbf{v}$

$(\mathbf{X}, \mathbf{F}) \leftarrow \mathscr{M}$
if there is no adjacent triangle **then**
  Select x' one of the vertices on the edge
  $\mathbf{b}' \leftarrow (0,0,0)$
  $k \leftarrow \mathrm{argwhere}_i \mathbf{F}[\mathrm{f}, i] = \mathrm{x}'$
  $\mathbf{b}'_k \leftarrow 1$
  $\mathbf{p}' \leftarrow (\mathrm{f}, \mathbf{b}')$
  return $\mathbf{p}', \mathbf{v}, ||\mathbf{p}' - \mathbf{p}||$
end if
$\mathrm{x}_0, \mathrm{x}_1 = \{\mathbf{X}[\mathbf{F}[\mathrm{f}, \mathbf{b}_i]] \mid i \in [0,2],\ \mathbf{b}_i \neq 0\}$
$\mathrm{f}' \leftarrow$ `connected_face`$(\mathscr{M}, \mathrm{f}, \mathrm{x}_0, \mathrm{x}_1)$
$\mathbf{n} \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f})$
$\mathbf{n}' \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f}')$
$\mathbf{e} \leftarrow (\mathbf{X}[\mathrm{x}_1] - \mathbf{X}[\mathrm{x}_0])/||\mathbf{X}[\mathrm{x}_1] - \mathbf{X}[\mathrm{x}_0]||$
$\alpha \leftarrow$ `signed_angle`$(\mathbf{n}, \mathbf{n}', \mathbf{e})$
$\mathbf{v}' \leftarrow$ `rotate_vector`$(\mathbf{v}, \mathbf{e}, \alpha)$
Let $\mathbf{b}'$ the new barycentric coordinates of $\mathbf{p}$ in f'
$\mathbf{p}' \leftarrow (\mathrm{f}', \mathbf{b}')$
return $\mathbf{p}', \mathbf{v}', 0$

computing gradients with respect to integer indices, which cannot be handled automatically. In addition, moving points intrinsically defined by their barycentric coordinates, which change arbitrarily from face to face, requires hard resets and per-face computations. Another issue is differentiating with respect to $\mathbf{v}$ as the number of iterations is a dynamic function of $\|\mathbf{v}\|$ and autograd can't differentiate with respect to the number of times a loop runs. Finally, a native autograd approach is also prohibitively inefficient, as it would require storing intermediate states, which would increase the time and memory required to compute the gradient. To evaluate these issues, we compare autograd against our differentiable scheme using both our straightest-geodesic implementation and the PI exponential map. We compute geodesic traces on various meshes and optimise for an arbitrary target point using the Euclidean distance; this choice isolates the performance of the differentiation schemes themselves, without introducing additional complexity from using a geodesic metric such as the biharmonic distance. The results, shown in Fig. A.2, report the computational performance for our method with both differentiation schemes (EP and GFD), as well as for PI using EP, GFD, and autograd. As discussed above, autograd is substantially slower because it must store intermediate values, whereas the EP scheme requires only the final state, making it far more efficient.

Speed-up GFD and EP through batching. The computational speed of our GFD differentiation scheme is primarily influenced by the number of Exp that need to be computed. Nevertheless, the number of Exp calls can be significantly reduced by batching. In fact, $\mathrm{Exp}(\mathbf{p}, \mathbf{v})$, $\mathrm{Exp}(\mathbf{p}, \mathbf{v} + \varepsilon_{\mathbf{v}} \hat{\mathbf{e}}_\perp)$, $\mathrm{Exp}(\mathbf{p}, \varepsilon_{\mathbf{p}} \hat{\mathbf{e}}_u)$, $\mathrm{Exp}(\mathbf{p}, \varepsilon_{\mathbf{p}} \hat{\mathbf{e}}_v)$ can all be combined in a single batched tracing operation by stacking the input arguments. A second execution of the trac-

ing algorithm can be computed by combining the following exponential map computations: $\mathrm{Exp}\big(\mathbf{p}', \prod_{\mathbf{p}}^{\mathbf{p}'}(\varepsilon_{\mathbf{v}}\hat{\mathbf{e}}_{\|})\big)$, $\mathrm{Exp}(\mathbf{p}_u, \mathbf{v}_u)$, and $\mathrm{Exp}(\mathbf{p}_v, \mathbf{v}_v)$. As we can see in Fig. A.2, batching provides a significant reduction in computational time. On the other hand, EP does not require the additional tracing, but can easily be batched using batched tensor operations.

A.2. Projection Integration Exp Map

Algorithm A.4 Projection Integration (PI) Exp map [51]

In: Mesh $\mathscr{M}$, point $\mathbf{p}$, tangent vector $\mathbf{v}$, step size s

$\mathrm{f}, \mathbf{b} \leftarrow \mathbf{p}$
$l \leftarrow 0; L \leftarrow ||\mathbf{v}||$
while $l < L$ **do**
  $\mathbf{p}' \leftarrow \mathbf{p} + s\mathbf{v}$
  $\mathbf{p}', \mathrm{f}' \leftarrow$ `project_to_mesh`$(\mathscr{M}, \mathbf{p}')$
  $\mathbf{n} \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f})$
  $\mathbf{n}' \leftarrow$ `face_normal`$(\mathscr{M}, \mathrm{f}')$
  $\alpha \leftarrow \cos^{-1}(\mathbf{n} \cdot \mathbf{n}')$
  $\mathbf{v} \leftarrow$ `rotate_vector`$(\mathbf{v}, \mathbf{n} \times \mathbf{n}', \alpha)$
  $l \leftarrow l + ||\mathbf{p}' - \mathbf{p}||$
  $\mathbf{p} \leftarrow \mathbf{p}'; \mathrm{f} \leftarrow \mathrm{f}'$
end while
return $\mathbf{x}$

We re-implemented the Projection Integration (PI) exponential map following the formulation in [51] and compared it with our method as well as the implementation provided by Geometry Central [74]. Our implementation is detailed in Alg. A.4. The projection step computes the projection of the point onto every face of the mesh, identifies the faces that contain the point, and selects the one that minimises the distance. This operation is computationally expensive and must be performed at every step. While the number of steps is independent of the mesh resolution (unlike in straightest geodesic implementations), the projection step scales with mesh resolution, thereby increasing the overall computational cost of tracing. Implementing the projection on the GPU is relatively straightforward, as it primarily involves tensor operations that are easily parallelisable. However, even with GPU acceleration, the computation of this Exp map remains substantially slower than the other methods.

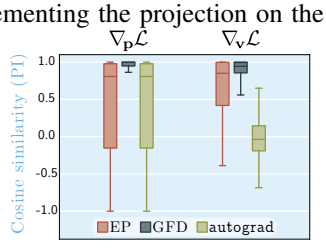


Figure A.3. Differentiation correctness of PI across differentiation schemes.

In Fig. A.3 we evaluate the differentiation correctness of PI when using autograd as well as our differentiation schemes. Cosine similarity results on PI remain consistent with previous findings leveraging our implementation of the straightest geodesics to compute the exponential map. This result is expected as the accuracy of both exponential maps was also comparable thanks to the choice of a small step size in PI (see Tab. 1).

A.3. Exp map on the Sphere

Given a starting point $\mathbf{p} \in \mathcal{S}^2$ and direction $\mathbf{v} \in \mathcal{T}_{\mathbf{p}}\mathcal{M}$. We assume these are given in Cartesian coordinates. The exponential map on the sphere is then given by [6, 15]:

$$\mathrm{Exp}_{\mathbf{p}}(\mathbf{v}) = \cos(||\mathbf{v}||)\mathbf{p} + \sin(||\mathbf{v}||)\frac{\mathbf{v}}{||\mathbf{v}||}$$

The Jacobian with respect to $\mathbf{p}$ can be computed differentiating with respect to $\mathbf{p}$:

$$\mathbf{J}_p = \cos(\|\mathbf{v}\|)\mathbf{I} \tag{A.1}$$

Computing the Jacobian with respect to $\mathbf{v}$ is slightly more cumbersome, but still achievable via simple differentiation. Thus, we have:

$$\mathbf{J}_{\mathbf{v}} = \frac{\partial}{\partial \mathbf{v}}\Big[\mathbf{p}\cos(\|\mathbf{v}\|)\Big] + \frac{\partial}{\partial \mathbf{v}}\left[\frac{\mathbf{v}}{\|\mathbf{v}\|}\sin(\|\mathbf{v}\|)\right]. \tag{A.2}$$

We now differentiate the two terms separately, while knowing that $\frac{\partial}{\partial \mathbf{v}}\|\mathbf{v}\| = \frac{\mathbf{v}^\top}{\|\mathbf{v}\|}$. The first term of Eq. (A.2) is:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{v}}\Big[\mathbf{p}\cos(\|\mathbf{v}\|)\Big] &= \mathbf{p}\frac{\partial}{\partial \mathbf{v}}\Big[\cos(\|\mathbf{v}\|)\Big] = -\mathbf{p}\sin(\|\mathbf{v}\|)\frac{\partial}{\partial \mathbf{v}}\|\mathbf{v}\| \\ &= -\mathbf{p}\sin(\|\mathbf{v}\|)\frac{\mathbf{v}^\top}{\|\mathbf{v}\|}\end{aligned} \tag{A.3}$$

The second term of Eq. (A.2) is:

$$\frac{\partial}{\partial \mathbf{v}}\left[\frac{\mathbf{v}}{\|\mathbf{v}\|}\sin(\|\mathbf{v}\|)\right] = \frac{\partial}{\partial \mathbf{v}}\left[\frac{\mathbf{v}}{\|\mathbf{v}\|}\right]\sin(\|\mathbf{v}\|) + \frac{\mathbf{v}}{\|\mathbf{v}\|}\frac{\partial}{\partial \mathbf{v}}\sin(\|\mathbf{v}\|).$$

If we first solve

$$\frac{\partial}{\partial \mathbf{v}}\left[\frac{\mathbf{v}}{\|\mathbf{v}\|}\right] = \frac{\|\mathbf{v}\|\mathbf{I} - \mathbf{v}\frac{\mathbf{v}^\top}{\|v\|}}{\|\mathbf{v}\|^2} = \frac{\|\mathbf{v}\|^2\mathbf{I} - \mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^3} = \frac{\mathbf{I}}{\|\mathbf{v}\|} - \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^3}$$

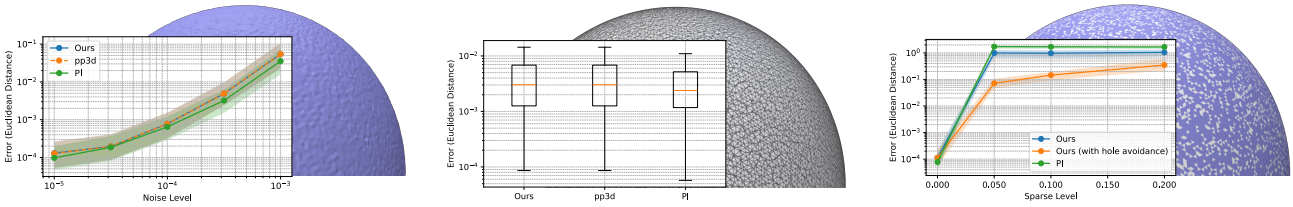
we can then finish differentiating the second term of Eq. (A.2):

$$\frac{\partial}{\partial \mathbf{v}}\left[\frac{\mathbf{v}}{\|\mathbf{v}\|}\sin(\|\mathbf{v}\|)\right] = \left[\frac{\mathbf{I}}{\|\mathbf{v}\|} - \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^3}\right]\sin(\|\mathbf{v}\|) + \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^2}\cos(\|v\|)$$

Putting everything back together in Eq. (A.2), we have:

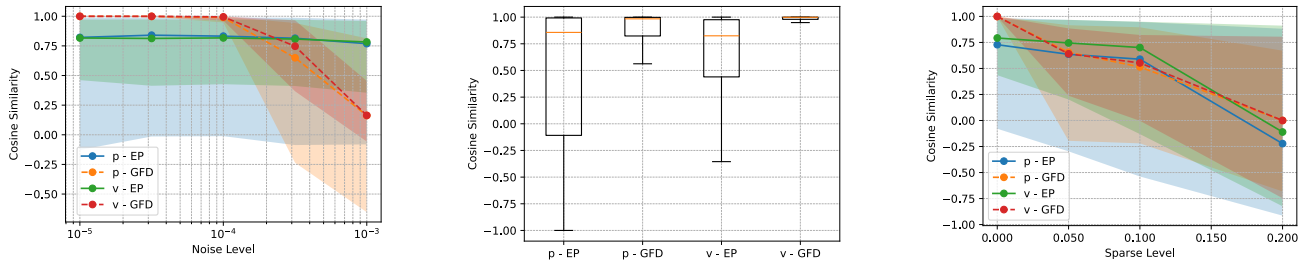
$$\mathbf{J}_{\mathbf{v}} = \left[\frac{\mathbf{I} - \mathbf{p}\mathbf{v}^\top}{\|\mathbf{v}\|} - \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^3}\right]\sin(\|\mathbf{v}\|) + \frac{\mathbf{v}\mathbf{v}^\top}{\|\mathbf{v}\|^2}\cos(\|v\|) \tag{A.4}$$

Then, the gradients of the loss $\mathcal{L}$ become $\nabla_{\mathbf{p}}\mathcal{L} = \nabla_{\mathbf{q}}\mathcal{L} \cdot \mathbf{J}_p$ and $\nabla_{\mathbf{v}}\mathcal{L} = \nabla_{\mathbf{q}}\mathcal{L} \cdot \mathbf{J}_v$



(a) Sphere with noise (b) Sphere with slim triangles (c) Sphere with holes

Figure A.4. Exponential map accuracy on sphere meshes with different types and amounts of corruption.



(a) Sphere with noise (b) Sphere with slim triangles (c) Sphere with holes

Figure A.5. Cosine similarity of the gradients using our method on spheres (using the GT exp map) and other meshes (using ODEs).

A.4. IVP on Torus

We consider a torus embedded in $\mathbb{R}^3$, with major and minor radius $R > r > 0$. Its smooth parametrisation is given by:

$$F(\alpha, \beta) = \begin{pmatrix} (R + r\cos\beta)\cos\alpha \\ (R + r\cos\beta)\sin\alpha \\ r\sin\beta \end{pmatrix}.$$

The tangent vectors are then given by:

$$F_\alpha = \frac{\partial F}{\partial \alpha} = \begin{pmatrix} -(R + r\cos\beta)\sin\alpha \\ (R + r\cos\beta)\cos\alpha \\ 0 \end{pmatrix},$$

$$F_\beta = \frac{\partial F}{\partial \beta} = \begin{pmatrix} -r\sin\beta\cos\alpha \\ -r\sin\beta\sin\alpha \\ r\cos\beta \end{pmatrix}.$$

Given the smooth parametrisation of the torus, F, we can compute the metric tensor, which defines the local curvature,

$$\begin{aligned} g &= \begin{pmatrix} \langle F_\alpha, F_\alpha\rangle & \langle F_\alpha, F_\beta\rangle \\ \langle F_\beta, F_\alpha\rangle & \langle F_\beta, F_\beta\rangle \end{pmatrix} \\ &= \begin{pmatrix} (R + r\cos\beta)^2 & 0 \\ 0 & r^2 \end{pmatrix} \end{aligned} \tag{A.5}$$

The Christoffel symbols are defined as:

$$\Gamma^k_{ij} = \tfrac{1}{2}\, g^{k\ell}\left(\frac{\partial g_{\ell i}}{\partial x^j} + \frac{\partial g_{\ell j}}{\partial x^i} - \frac{\partial g_{ij}}{\partial x^\ell}\right)$$

For vector fields $\mathcal{V}, \mathcal{W} : \mathcal{M} \to \mathcal{TM}$, the Christoffel symbols act as the correction terms needed to express differentiation while staying consistent with the manifold's geometry,

$$(\nabla_{\mathcal{V}}\mathcal{W})^k = \sum_i \mathcal{V}^i \frac{\partial \mathcal{W}^k}{\partial x^i} + \sum_{i,j} \Gamma^k_{i,j}\mathcal{V}^i\mathcal{W}^j$$

Using Eq. (A.5), the non-zero Christoffel symbols can be computed as,

$$\Gamma^\alpha_{\alpha\beta} = \Gamma^\alpha_{\beta\alpha} = -\frac{r\sin\beta}{R + r\cos\beta}, \quad \Gamma^v_{\alpha\alpha} = \frac{(R + r\cos\beta)\sin\beta}{r}.$$

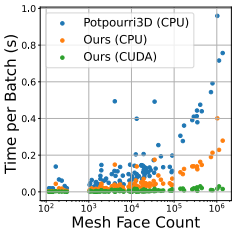
Injecting the Christoffel symbols inside the geodesic equation, $\nabla_{\dot\gamma(t)}\dot\gamma(t) = 0$, gives:

$$\begin{cases} \alpha'' - 2\dfrac{r\sin\beta}{R + r\cos\beta}\,\alpha'\beta' = 0 \\ \beta'' + \dfrac{(R + r\cos\beta)\sin\beta}{r}\,(\alpha')^2 = 0 \end{cases} \tag{A.6}$$

Given a starting point $\mathbf{p} \in \mathcal{M}$ and direction $\mathbf{v} \in \mathcal{T}_\mathbf{p}\mathcal{M}$, we can compute the initial conditions, $\alpha(0), \beta(0), \alpha'(0), \beta'(0)$ and use Eq. (A.6) to compute α'', β'' and integrate this differential equation to get $F(\alpha(||\mathbf{v}||), \beta(||\mathbf{v}||))$, the endpoint of the exponential map.

A.5. Additional Experiments on Our Method

Speed and accuracy comparison on more meshes. To further corroborate our findings of Fig. 3 we evaluate our method also on 100 meshes randomly sampled from the Thingi10K dataset [90]. This plot reported as insert figure closely follows the trend and timings of Fig. 3. Evaluating the accuracy against `pp3d` on the same set of 100 meshes from Thingi10K, we report mean errors in the order of 10^{-6}, thus closely aligning with Tab. 1.



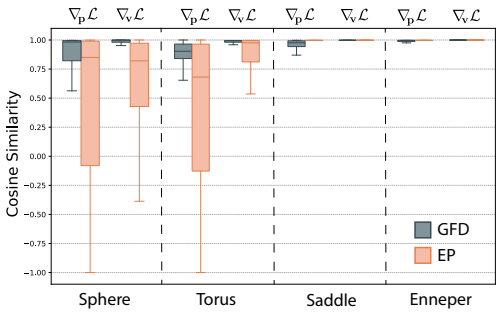


Figure A.6. Cosine similarity of the gradients using our method on different meshes

Robustness to mesh quality. We assess the robustness to mesh quality for both our forward and backward steps, reporting results in Figs. A.4 and A.5, respectively. Specifically, in Figs. A.4a and A.5a, we progressively add noise to vertex positions; in Figs. A.4b and A.5b, we deform approximately one third of the edges to produce slim triangles; in Figs. A.4c and A.5c, we progressively remove triangles. The *sparse level* on the x-axis of Figs. A.4c and A.5c denotes the percentage of triangles removed. In this setting, `pp3d` fails to compute geodesics as some vertices belong to multiple boundary loops, whereas our method remains applicable. While in Fig. A.4c we report results for both the default and hole-avoidance (Sec. A.1) versions of our method, gradients in Fig. A.5 are always evaluated using hole-avoidance.

Gradient evaluation on more meshes. We conducted additional validations on multiple, positive and negative curvature surfaces, as shown in Fig. A.6. We address the lack of ground-truth (GT) gradients, by comparing against those from the ODEs using `autograd` and compute the loss using an arbitrary point, $\mathcal{L} = ||\exp_{\mathbf{p}}(\mathbf{v}) - \hat{\mathbf{p}}||^2$. While these numerical methods serve as a high-quality proxy, they remain approximations rather than exact GTs.

A.6. Additional Details on AGC Neural Network

We implemented our AGC Neural Network (AGCNN) in PyTorch [64] using the same architecture as the U-ResNet from MDGCNN [66], which is based on Residual Networks [34] and U-Net [71]. The architecture is depicted in Fig. A.7, where the ResNet stacks are made of two ResNet blocks, which are described on Fig. A.8. For the pooling layer, we subsampled the meshes using quadric decimation [28] and targeted around a quarter of the faces of the mesh. Like in [66], pooling and un-pooling are performed by matrix multiplying vertex features with sparse matrices obtained during quadric decimation, a common practice in many SotA methods [11, 25, 26, 31]. We also doubled the number of filters used on the subsampled meshes (in the stacks 2 and 3).

As mentioned in Sec. 5.2, our network takes as input a vector of real values for each vertex. Like in DiffusionNet [76], we considered two possible input types: the raw 3D coordinates of the vertices, and the Heat Kernel Signature (HKS) [80]. When using 3D coordinates, we paired them with rotation and scaling augmentations, using a random rotation and a uniform scaling between 0.85 and 1.15. We also normalised the sizes of the meshes beforehand. With HKS as input, the network is invariant to any orientation-preserving isometric deformation of the shape, so rotation augmentation is not necessary. To compute the HKS, we used the same setup as DiffusionNet [76], with 128 eigenvectors and heat kernel signatures sampled at 16 temporal values logarithmically spaced on $[0.01, 1]$.

We fit the model using Adam [41] optimiser, and a cosine annealing learning rate scheduler, with an initial learning rate of 10^{-3} and an η of 10^{-5}. We used a cross-entropy loss with 0.1 label smoothing. We trained our model on a Nvidia A100, taking 1 minute and 21 seconds per epoch and using around 6.8 GB of VRAM, while training on the full size meshes, which are made of around 10k vertices. This means, for each mesh, we compute roughly 30 million geodesic traces and still maintain a reasonable run time. We trained the model for 40 epochs on hks, and 60 epochs on xyz.

A.7. Additional Details on MeshFlow

Differences and Similarities with RFM. As mentioned in Secs. 1 and 5.3, MeshFlow is inspired by flow matching, but there are multiple fundamental differences that we highlight in this section. The pseudocodes of our MeshFlow and RFM [15] are reported in Alg. A.5 and A.6, respectively. Note that Alg. A.6 is adapted to adhere to the notation of this paper and reports only the case in which RFM is applied to meshes. Note also that $\mathbf{p}_i^{(1)}$ in Alg. A.6 corresponds to our $\mathbf{q}_i$ in Alg. A.5.

Algorithm A.5 MeshFlow

In: Mesh $\mathscr{M}$, distance d_{BH}, base $\mathcal{P}$, target $\mathcal{Q}$

Initialize parameters θ of $\mathbf{v}_\theta$
while not converged **do**
  $\mathbf{p}_1, \dots, \mathbf{p}_B \sim \mathcal{P}(\mathscr{M})$ (sample noise)
  $\mathbf{q}_1, \dots, \mathbf{q}_B \sim \mathcal{Q}(\mathscr{M})$ (sample training examples)
  $\sigma \leftarrow$ Eq. (A.7) (OT couplings)
  $\hat{\mathbf{q}}_i \leftarrow \mathrm{Exp}^{\circlearrowleft K}_{\mathbf{p}_i}(\mathbf{v}_\theta), \forall i = 1, \dots, B$
  $\mathcal{L} \leftarrow \frac{1}{B}\sum_{i=1}^{B} d^2_{BH}(\mathbf{q}_{\sigma(i)}, \hat{\mathbf{q}}_i)$
  $\theta \leftarrow$ `optimizer_step`$(\nabla_\theta \mathcal{L})$
end while

Both methods use a tangent vector field to transport points from a noisy source distribution to the target data distribution. The core differences between the two methods lie in: *(i)* how points are geodetically transported on the sur-

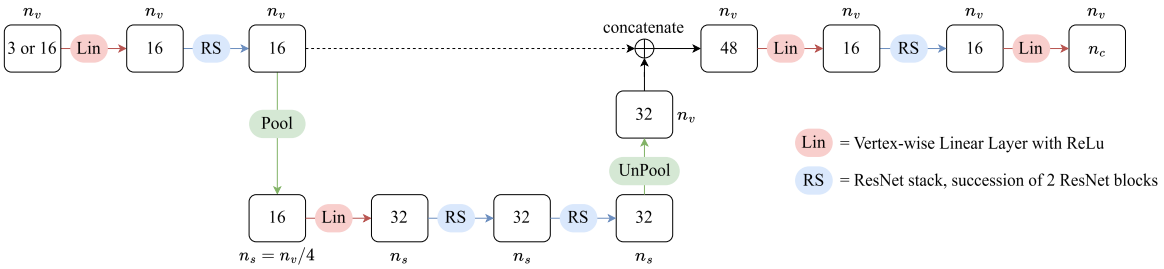


Figure A.7. U-ResNet architecture used for segmentation. With n_v the number of vertices, n_s the subsampled vertices and n_c the number of classes.

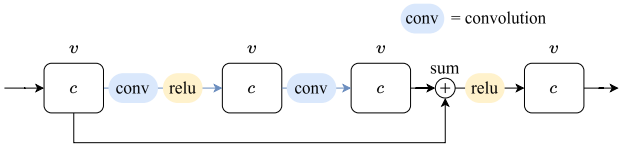


Figure A.8. ResNet block architecture.

Algorithm A.6 RFM [15] on $\mathscr{M}$

In: Mesh $\mathscr{M}$, base $\mathcal{P}$, target $\mathcal{Q}$, scheduler κ

Initialize parameters θ of $\mathbf{v}_\theta$
while not converged **do**
  $t_1, \dots, t_B \sim \mathcal{U}(0,1)$ (sample time)
  $\mathbf{p}_1^{(0)}, \dots, \mathbf{p}_B^{(0)} \sim \mathcal{P}(\mathscr{M})$ (sample noise)
  $\mathbf{p}_1^{(1)}, \dots, \mathbf{p}_B^{(1)} \sim \mathcal{Q}(\mathscr{M})$ (sample training examples)
  $\mathbf{p}_i^{(t_i)} \leftarrow \texttt{solve\_ODE}\big(t_i, \mathbf{p}_i^{(0)}, \mathbf{u}^{(t)}(\mathbf{p}|\mathbf{p}_i^{(1)})\big), \forall i$
  $\mathcal{L} \leftarrow \frac{1}{B}\sum_{i=1}^{B} \|\mathbf{v}_\theta^{(t_i)}(\mathbf{p}_i^{(t_i)}) - \dot{\mathbf{p}}_i^{(t_i)}\|_2^2$
  $\theta \leftarrow \texttt{optimizer\_step}(\nabla_\theta \mathcal{L})$
end while

face of a mesh, *(ii)* the choice of vector field, *(iii)* the loss function, and *(iv)* how points are paired.

MeshFlow leverages our differentiable exponential map

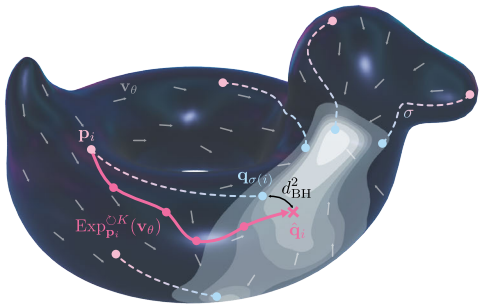


Figure A.9. Visual representation of MeshFlow. Noise samples, $\mathbf{p}_i$, are displaced via our K-steps exponential map according to the directions dictated by our learnable static vector field, $\mathbf{v}_\theta$. The end-points, $\hat{\mathbf{q}}_i$, are compared with the OT-coupled training samples, $\mathbf{q}_{\sigma(i)}$, using a squared biharmonic distance function d^2_{BH}.

to move points from source to target. Since our exponential map is based upon straightest geodesics, to enable curved paths, we compute K intermediate steps. On the other hand, RFM needs to solve the ODE corresponding to the gradient flow of the distance function, which is normalised to ensure constant speed along the trajectory. We leave the formal definition of the ODE to [15]. However, intuitively, the ODE dictates how to perform small steps along the trajectory. The movement direction is determined by the gradient of the biharmonic distance towards the target point. This Euler method performs Euclidean displacements, thus points need to be projected onto the mesh after every step.

Our method uses a time-invariant vector field $\mathbf{v}_\theta$, which is sampled at every step of the K-step exponential map. In contrast, RFM uses a time-conditioned vector field $\mathbf{v}_\theta^{(t)}$. During inference, this field is sampled for each small step of the Euler method. During training, it is optimised at multiple steps of the trajectory connecting source to target samples.

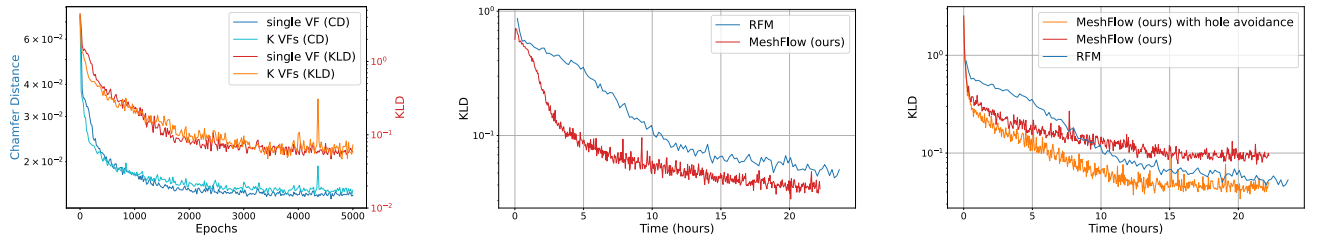
Since MeshFlow can move samples in a single pass and directly obtain the transported points, they can be directly compared against data samples using the biharmonic distance d_{BH}. RFM adopts a completely different strategy and computes the mean squared Euclidean distance between the tangent of the curve ($\dot{\mathbf{p}}_i^{(t_i)}$) at $\mathbf{p}_i^{(t_i)}$, and the vector field $\mathbf{v}_\theta^{(t_i)}$ at the arbitrarily selected time step t_i and corresponding position $\mathbf{p}_i^{(t_i)}$.

To find good matches, MeshFlow requires coupling noise ($\mathbf{p}_i \sim \mathcal{P}(\mathscr{M})$) with target ($\mathbf{q}_i \sim \mathcal{Q}(\mathscr{M})$) samples via optimal transport. RFM, on the other hand, can just select random couplings.

Optimal Transport (OT) Couplings. The OT coupling is computed using the Hungarian algorithm [45] and minimising the linear sum assignment of the pairwise squared Biharmonic distance:

$$\sigma = \underset{\hat{\sigma}\in\mathcal{S}}{\text{argmin}} \sum_{i=1}^{B_c} d^2_{BH}(\mathbf{p}_i, \mathbf{q}_{\hat{\sigma}(i)}), \tag{A.7}$$

where σ is the permutation of $\{1, ..., B_c\}$ representing the OT coupling between the noise $\mathbf{p}_1, \dots, \mathbf{p}_{B_c}$ and target



(a) Comparison between learning a single vector field and K vector fields.

(b) KLD over time when training on the Bunny with $k = 100$.

(c) KLD over time when training on Spot with holes and $k = 10$.

Figure A.10. Comparison of different validation metrics during training.

$\mathbf{q}_1, \dots, \mathbf{q}_{B_c}$ samples, and $\mathcal{S}$ the set of all possible permutations of the set $\{1, \dots, B_c\}$.

Although we can use the same number of paired samples during training and while establishing the couplings, a higher number of samples is particularly beneficial for OT coupling. Therefore, we enable the selection of two different batch sizes: B for training and B_c for OT coupling, with $B \leq B_c$. In this case, the K-step exponential map, loss, and optimiser step in Alg. A.5 are repeated B_c/B times, to use all the previously OT-coupled pairs.

Implementation Details. We optimise MeshFlow using Adam [41] with a learning rate of 3.10^{-5}, for $5,000$ epochs. We use 20k training samples and 5k test samples per mesh. We report results for $B_c = 1024$ and $B = 256$ (MeshFlow-1024), as well as $B_c = B = 256$ (MeshFlow-256) in Tab. 3. Also, as previously reported in Sec. 5.3, to learn the vector field, we use an MLP with 3 hidden layers of size 512, and $K = 5$. The MLP takes as input the 3D position of the samples and outputs the corresponding vector. We use the GFD differentiation scheme as we need to back-propagate also through positions.

We also experimented with learning K separate vector fields (one per step), and report the corresponding metrics in Fig. A.10a. However, this approach did not yield improved performance and required longer training time.

Detailed Performance Comparison with RFM. We benchmark MeshFlow against RFM on an Nvidia A100 GPU, and use the same batch size, training samples, and test samples. For our method, using an OT batch size of 1024, each epoch took around 8 seconds and used 1.6 GB of VRAM, and was trained for $5,000$ epochs, or around 11 hours. For RFM, each epoch took around 162 seconds, and used 38 GB of VRAM, and was trained for around 400 epochs, or around 18 hours. In Fig. A.10b we compare the convergence speed of MeshFlow and RFM when trained for approximately the same time. Our method not only converges significantly faster (Fig. A.10b), but also has significantly faster inference (98s for RFM vs $5.8 \cdot 10^{-3}$s for ours), as it requires less steps and does not rely on projections to stay on the manifold.

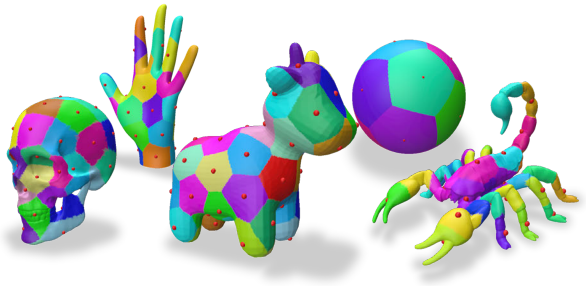


Figure A.11. Examples of GCVTs using our method on Skull, Hand, Spot, Ball and Scorpion.

While already significantly faster than RFM, the main bottleneck of MeshFlow is computing the OT coupling. The Hungarian algorithm we currently use runs on the CPU and, with a complexity of $\mathcal{O}(B_c^3)$, it does not scale well with larger batch sizes. Nevertheless, this problem can be efficiently addressed using faster coupling techniques, such as the Sinkhorn algorithm [24, 77].

MeshFlow with Hole Avoidance. We tested our method with and without the hole avoidance mechanism introduced in Sec. 4 and further detailed in Sec. A.1, where the algorithm attempts to parallel transport the direction of the geodesic along the hole, and prevents the trace from stopping abruptly. As observed on Fig. A.10c, while RFM proved to be robust to the presence of holes, our method still outperforms RFM when using the hole avoidance mechanism. The gap in performance between our method with and without hole avoidance justifies the introduction of this key modification to the straightest geodesics algorithm.

A.8. Additional Details on Mesh-LBFGS and GCVT

The Geodesic Centroidal Voronoi Tessellation (GCVT) on a mesh $\mathscr{M}$ aims to move the seeds $\mathbf{S} = \{\mathbf{s}_i\}_{i=1}^S \in \mathscr{M}^S$ to the centre of mass of their respective Voronoi cells Ω_i. Centroids are defined as the Karcher mean of the Voronoi cells [75].

Since we are operating on triangular meshes, we dis-

cretise the computation of the Karcher mean to make it tractable. We compute the vertex area $A(\mathbf{x})\ \forall \mathbf{x} \in \mathbf{X} \in \mathscr{M}$, which is defined as a third of the sum of the areas of its connected triangles. We then use the algorithm proposed by [75] to compute the Karcher mean while also leveraging their heat-based logarithm map formulation. In practice, we only take a single step of the Karcher mean algorithm. We define the update vector $\mathbf{v}_i$ pointing towards the centroid of region Ω_i as:

$$\mathbf{v}_i = \frac{\sum_{\Omega_i} A(\mathbf{x})\rho(\mathbf{x}) \log_{\mathbf{s}_i}(\mathbf{x})}{\sum_{\Omega_i} A(\mathbf{x})\rho(\mathbf{x})},$$

where $\rho(\mathbf{x}) = \frac{k_t(\mathbf{s}_i,\cdot)}{\sum_j k_t(\mathbf{s}_j,\cdot)}$ is the density function defined at the vertices of the mesh via the scalar heat kernel k_t. Using the standard Lloyd's algorithm, seeds are then updated via $\mathbf{s}_i' = \mathrm{Exp}_{\mathbf{s}_i}(\mathbf{v}_i)$. This is equivalent to minimising the following energy function:

$$E(\mathbf{S}) = \frac{1}{2S} \sum_{i=1}^{S} \sum_{\Omega_i} A(\mathbf{x})\rho(\mathbf{x}) || \log_{\mathbf{s}_i}(\mathbf{x})||^2.$$

To extend this method to a Riemannian second-order optimiser (our Mesh-LBFGS), we leverage the fact that $\mathbf{v}_i$ represents the steepest descent direction. Thus, we approximate the gradient of the energy with the opposite of the directions computed via the Karcher mean, i.e., $\nabla E(\mathbf{S}) \approx (-\mathbf{v}_0, \dots, -\mathbf{v}_n)$. The vector transport Π needed for the optimiser is computed via our Exp map when computing $\mathbf{s}_{i+1}$. Its adjoint is simply the inverse of this parallel transport, where a vector will be moved backwards on the curve.

We then apply Alg. A.7 to optimise the seeds. In the general Riemannian L-BFGS framework, a Riemannian metric is required; in our case, however, the mesh is embedded in $\mathbb{R}^3$, so the metric is the one induced by the ambient Euclidean space, $g_{\mathbf{p}}(\mathbf{u}, \mathbf{v}) = \langle \mathbf{u}, \mathbf{v} \rangle$.

Since we are moving a batch of seeds simultaneously, we operate on the product manifold $\mathscr{M}^S$, and treat the collection $\mathbf{S}$ as a single state. The metric, Exp and vector transport are easily derived on this product manifold, for $\mathbf{S}, \mathbf{S}' \in \mathscr{M}^S$ and $\mathbf{V}, \mathbf{V}' \in \mathcal{T}_{\mathbf{S}}\mathscr{M}^S$:

$$\begin{aligned} g_{\mathbf{S}}(\mathbf{V}, \mathbf{V}') &= \langle \mathbf{V}, \mathbf{V}' \rangle = \langle \mathbf{v}_1, \mathbf{v}_1' \rangle + \dots + \langle \mathbf{v}_S, \mathbf{v}_S' \rangle \\ \mathrm{Exp}_{\mathbf{S}}(\mathbf{V}) &= (\mathrm{Exp}_{\mathbf{s}_1}(\mathbf{v}_1), \dots, \mathrm{Exp}_{\mathbf{s}_S}(\mathbf{v}_S)) \\ \Pi_{\mathbf{S}}^{\mathbf{S}'} \mathbf{V} &= (\Pi_{\mathbf{s}_1}^{\mathbf{s}_1'} \mathbf{v}_1, \dots, \Pi_{\mathbf{s}_S}^{\mathbf{s}_S'} \mathbf{v}_S) \end{aligned}$$

We set the base learning rate α_0 to 1 for Lloyd's algorithm and 0.5 for Mesh-LBFGS. While the smaller rate slows the initial steps, it yields greater stability and faster overall convergence, outperforming Lloyd's algorithm after a few iterations.

We compare our optimiser with Lloyd's algorithm on different meshes with different initial sampling methods, and compare both the loss function and total function calls. In Fig. A.12 we use random clustered seeds, and in Fig. A.13 we use random uniform seeds. We resample the seeds at each run and plot the medians with the 25% and 75% quartiles. Overall, our method converges in fewer iterations than Lloyd's algorithm, but at the same time, it also incurs more function calls. Nevertheless, because it reaches convergence much faster overall, the total number of function calls is ultimately lower, especially when using clustered seeds.

Algorithm A.7 Mesh-LBFGS

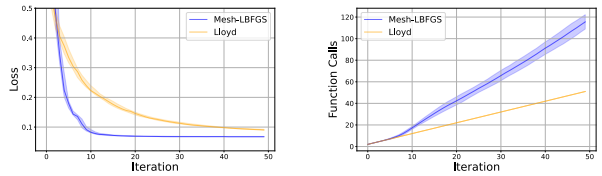
In: Mesh $\mathscr{M}$; vector transport $\prod$; exponential map Exp; initial value $\mathbf{S}_0 \in \mathscr{M}^S$; smooth function $f : \mathscr{M}^S \to \mathbb{R}$

Set $H_{\mathrm{diag}} \leftarrow 1$
for $t = 0, 1, \dots$ **do**
  $\mathbf{V}_t \leftarrow \texttt{desc}(-\nabla f(\mathbf{S}_t), t)$ //descent direction
  Use line-search to find α satisfying Wolfe conditions
  $\mathbf{S}_{t+1} \leftarrow \mathrm{Exp}_{\mathbf{S}_t}(\alpha \mathbf{V}_t)$
  $\mathbf{A}_{t+1} \leftarrow \prod_{\mathbf{S}_t}^{\mathbf{S}_{t+1}}(\alpha \mathbf{V}_t)$
  $\mathbf{B}_{t+1} \leftarrow \nabla f(\mathbf{S}_{t+1}) - \prod_{\mathbf{S}_t}^{\mathbf{S}_{t+1}}(\nabla f(\mathbf{S}_t))$
  $H_{\mathrm{diag}} \leftarrow \dfrac{\langle \mathbf{A}_{t+1}, \mathbf{B}_{t+1} \rangle}{\langle \mathbf{B}_{t+1}, \mathbf{B}_{t+1} \rangle}$
end for
Return $\mathbf{S}_{t+1}$

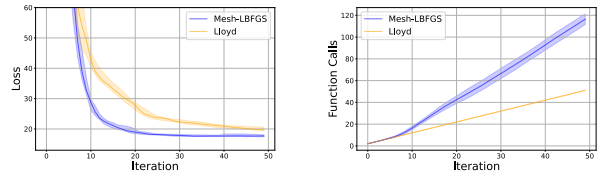
Algorithm A.8 `desc`

In: Vector $\mathbf{V} \in \mathcal{T}_{\mathbf{S_t}}\mathscr{M}^S$, iteration t

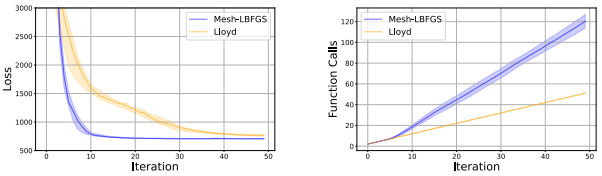
if $t > 0$ **then**
  $\tilde{\mathbf{V}} \leftarrow \mathbf{V} - \dfrac{\langle \mathbf{A}_t, \mathbf{V} \rangle}{\langle \mathbf{B}_t, \mathbf{A}_t \rangle} \mathbf{B}_t$
  $\hat{\mathbf{V}} \leftarrow \prod_{\mathbf{S}_{t-1}}^{\mathbf{S}_t} \Big(\texttt{desc}\big({}^*\prod_{\mathbf{S}_{t-1}}^{\mathbf{S}_t} \tilde{\mathbf{V}}, t-1 \big) \Big)$
  // with ${}^*\prod$ the adjoint of $\prod$
  return $\hat{\mathbf{V}} - \dfrac{\langle \mathbf{B}_t, \hat{\mathbf{V}} \rangle}{\langle \mathbf{B}_t, \mathbf{A}_t \rangle} \mathbf{A}_t + \dfrac{\langle \mathbf{A}_t, \mathbf{A}_t \rangle}{\langle \mathbf{B}_t, \mathbf{A}_t \rangle} \mathbf{V}$
else
  return $H_{\mathrm{diag}}\, \mathbf{V}$
end if



(a) On Spot mesh

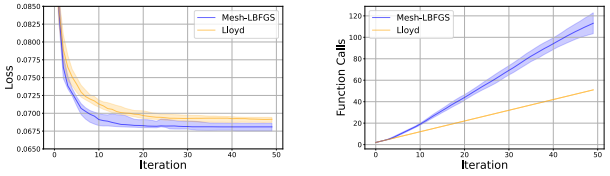


(b) On Scorpion mesh

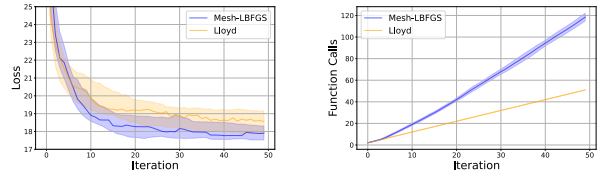


(c) On Skull mesh

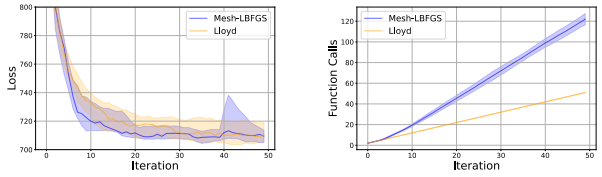
Figure A.12. Comparison of different GCVT optimisers for 50 **clustered** seeds over 20 runs on different meshes. *Left*: Median loss functions. *Right*: Median total function calls.



(a) On Spot mesh



(b) On Scorpion mesh



(c) On Skull mesh

Figure A.13. Comparison of different GCVT optimisers for 50 **uniform** seeds over 20 runs on different meshes. *Left*: Median loss functions. *Right*: Median total function calls.