Voronoi-based Variational Reconstruction of Unoriented Point Sets

Pechetti Sai Akhil B21AI025
Prashant Tandon B21AI053

Voronoi-PCA Estimation

Paper Concept

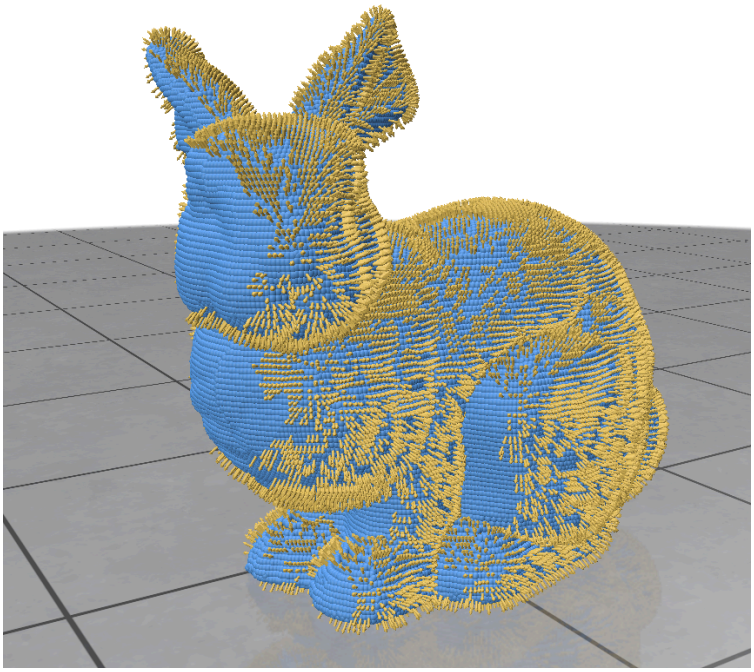
- **Voronoi Diagram & Normal Estimation:** The method in the paper uses Voronoi cells to provide a global analysis of the point set, estimating the normal based on the elongation (anisotropy) of these cells. This idea is incorporated in the code when computing the covariance matrix for the neighboring points, and performing PCA on these local regions.

- **Principal Component Analysis (PCA):** The method uses PCA (by computing the covariance matrix and finding eigenvalues/eigenvectors) to estimate the normal vector, as discussed in the paper. The normal is based on the eigenvector associated with the smallest eigenvalue.
- **Confidence via Anisotropy:** The confidence in the normal estimate is based on the ratio of the smallest eigenvalue to the sum of the eigenvalues, aligning with the idea in the paper that a higher anisotropy (elongation of the Voronoi cell) indicates a more reliable normal.

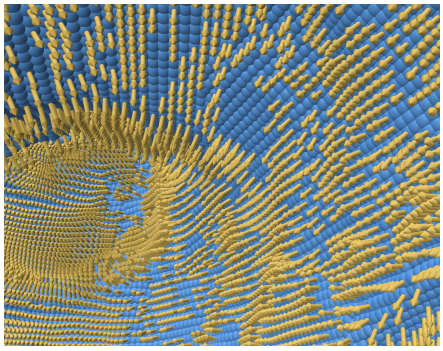
Code

```
def compute_voronoi_pca_normals(pcd, k=20):
    # Step 1: Convert to numpy array
    points = np.asarray(pcd.points)

    # Step 2: KDTree for neighbor search
    kdtree = o3d.geometry.KDTreeFlann(pcd)

    normals = []
    confidences = []

    for i in range(len(points)):
        # Get k nearest neighbors
        _, idx, _ = kdtree.search_knn_vector_3d(pcd.points[i], k)
        neighbors = points[idx]

        # Compute Voronoi diagram
        try:
            vor = Voronoi(neighbors)
        except:
            # Degenerate configuration
            normals.append([0, 0, 0])
            confidences.append(0)
            continue
```

```
        # PCA on neighbors
        cov = np.cov(neighbors.T)
        eigvals, eigvecs = np.linalg.eigh(cov)
        normal = eigvecs[:, 0]  # Smallest eigenvector

        # Confidence: ratio of smallest eigenvalue to sum
        confidence = 1 - eigvals[0] / (eigvals.sum() + 1e-8)

        normals.append(normal)
        confidences.append(confidence)

    pcd.normals = o3d.utility.Vector3dVector(np.array(normals))
    return np.array(normals), np.array(confidences)
```



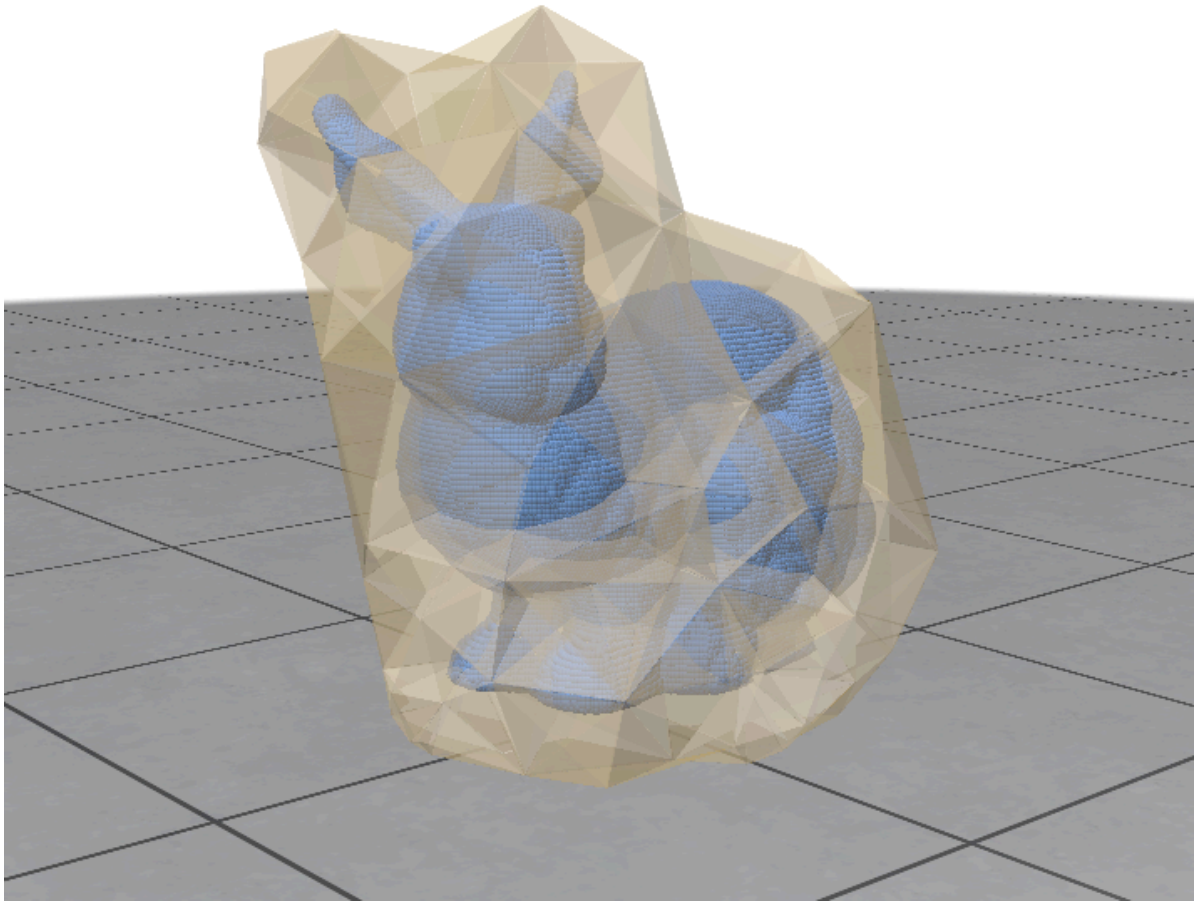
Unoriented Vertex Normals



Internal facing Normals

Delaunay Refinement

- The paper explains **Delaunay refinement**, which is a method to improve 3D models created from point clouds that might be messy or contain errors.
- This method works by taking a Delaunay tetrahedral mesh and making it better through several steps of improvements.
- These improvements help prevent mathematical errors that can happen when the 3D shapes in the model are poorly formed or misshapen.
- To make the model better, the method adds new points (called **Steiner points**) in carefully chosen spots. These new points help ensure that all the 3D shapes in the model meet quality standards.
- The method is smart about where it makes improvements. It only works within a slightly enlarged box around the original points, so it doesn't waste time on areas that don't matter.
- This focused approach helps maintain the original data's accuracy while avoiding unnecessary changes to the model.



Output of Delaunay Refinement to construct a volumetric mesh

Bounding Box (AABB) Creation and Enlargement

```
# Compute AABB
self.points = points
min_bound = np.min(points, axis=0)
max_bound = np.max(points, axis=0)
delta = max_bound - min_bound
# Enlarge bounding box
self.lower = min_bound - padding * delta
self.upper = max_bound + padding * delta
```

The `BBoxDomain` class calculates the **Axis-Aligned Bounding Box (AABB)** of the input points and then enlarges it by a padding factor (`padding`). This ensures that the mesh refinement procedure occurs within a well-defined space.

Steiner Point Insertion and Mesh Refinement

```
mesh = pygalmesh.generate_mesh(
    domain,
    exude=True,                          # Exude Steiner points to improve mesh quali
    max_edge_size_at_feature_edges=0.02,       # Maximum edge size at feature
    min_facet_angle=25.0,                # Minimum facet angle in degrees
    max_radius_surface_delaunay_ball=0.1,      # Maximum radius of the surface D
    max_facet_distance=0.01,             # Maximum distance between facet circ
    max_circumradius_edge_ratio=2.0,           # Maximum circumradius to edge ra
    max_cell_circumradius=0.07,          # Maximum circumradius of tetrahedra
    exude_time_limit=100.0,              # Optional: time limit for exuding Steiner p
    exude_sliver_bound=0.1,              # Optional: sliver-bound limit for exuding
)
```

- `pygalmesh.generate_mesh` function is used to generate the mesh, with the `exude` parameter set to `True`, which triggers the insertion of **Steiner points.**
- Parameters like `max_edge_size_at_feature_edges` and `max_facet_angle` help refine the mesh and ensure that the generated tetrahedra is proper.

Constrained optimization - Generalized Eigen Problem

Solving a **constrained optimization problem** to compute a **piecewise linear function** over a 3D tetrahedral mesh obtained via Delaunay refinement in the previous step.

To solve this optimization problem, the paper introduces the concept of a **generalized eigenvalue problem (GEP)**, specifically:

$$A\mathbf{F} = \lambda B\mathbf{F}$$

where:

- A is the matrix for **anisotropic Dirichlet energy**.
- B is the matrix for **biharmonic energy**.
- $\mathbf{F}$ is the vector representing the piecewise linear function.
- λ is the eigenvalue corresponding to the eigenvector $\mathbf{F}$

Cholesky Factorization of Matrix B

```
# Ensure B is positive definite (a requirement for Cholesky factorization)
B = B + n * np.eye(n)  # Adding a diagonal dominance to make B positive definite

# Method 1: Using Cholesky factorization and classical operator
L = cholesky(B, lower=True)  # Cholesky decomposition
```

- Adds **diagonal dominance** to matrix B to ensure that it is positive definite.
- Matrix B is **decomposed using Cholesky factorization** to obtain L, the lower triangular matrix.

Matrix Operator M

After Cholesky factorization, the paper transforms the generalized eigenvalue problem into a

classical eigenvalue problem by applying the inverse of L

$$L^{-1}AL^{-T}G = \lambda G$$

where $G = L^T F$.

```
def apply_M(x):
    y = solve_triangular(L.T, x, lower=False)
    z = A @ y
    return solve_triangular(L, z, lower=True)

M_op = LinearOperator((n, n), matvec=apply_M)
```

Function `apply_M(x)`, which effectively implements the **matrix operator** that transforms the generalized eigenvalue problem into a classical one.

Solving the Eigenvalue Problem

The paper uses the **Arnoldi iteration** method to solve the eigenvalue problem for the largest eigenvalue.

```
from scipy.sparse.linalg import eigsh
eigvals_classical, eigvecs_classical = eigsh(M_op, k=k, which='LA')  # Solve eige
G_classical = eigvecs_classical[:, 0]  # The eigenvector corresponding to the larg

# Recover original eigenvector F
F_classical = solve_triangular(L.T, G_classical, lower=False)
```

- The **eigenvalue problem** is solved using the `eigsh` function (which implements **Arnoldi iteration**) to find the eigenvector corresponding to the **largest algebraic eigenvalue.**
- The final solution is the eigenvector F corresponding to the largest eigenvalue. This eigenvector defines a **piecewise linear function** over the tetrahedral mesh.

Iso-contouring

Evaluate only input (original) points, not Steiner points

```
from scipy.spatial import cKDTree

# mesh.points is the array of vertices of the tetrahedral volume mesh
mesh_points = mesh.points

# Build KD-tree on mesh vertices
mesh_kdtree = cKDTree(mesh_points)

# Query nearest mesh vertex for each original point
distances, indices = mesh_kdtree.query(points_np, k=1)

# 'indices' gives us the closest mesh vertex for each input point
original_point_indices = np.unique(indices)
```

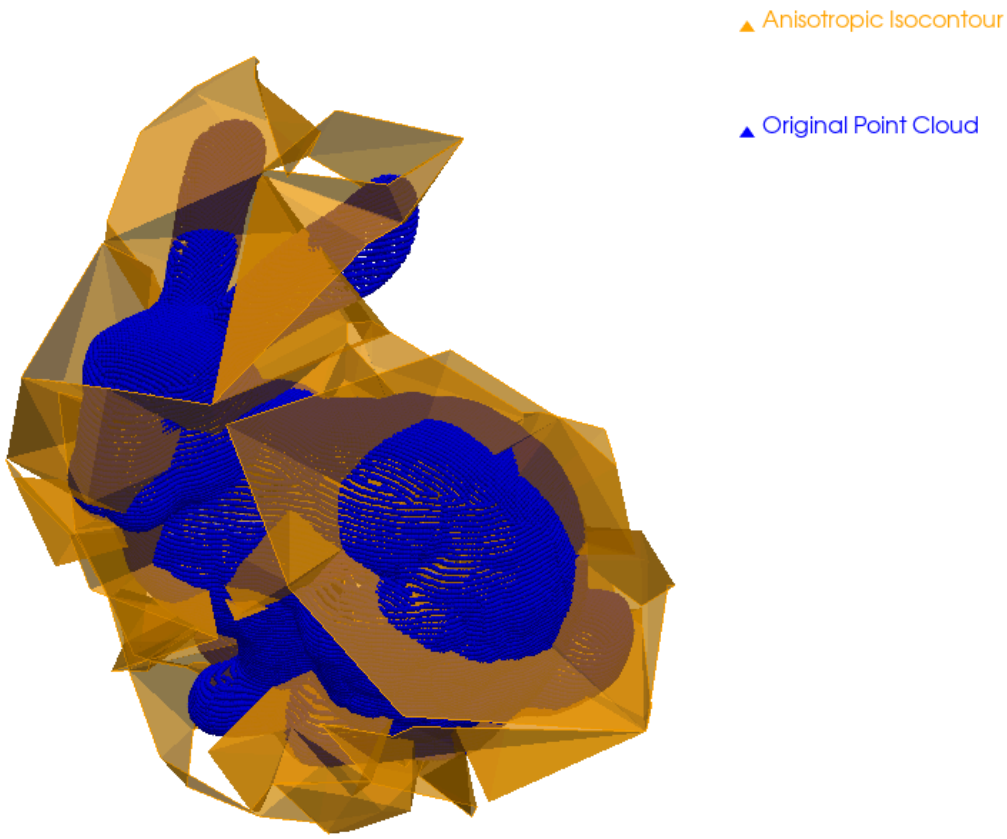
Using a **k-d tree** to match the **original point cloud** to the **volume mesh vertices** is a great way to identify `original_point_indices`

Pick the median of those values as the isovalue for contouring

```
# Now compute isovalue over only original mesh points
  isovalue = np.median(F_classical[original_point_indices])
  print("Isovalue (median over original points):", isovalue)
```

Only use function values at original input locations for extracting the isosurface.

Surface Extraction



Final output of the algorithm

```
mesh.point_data["f"] = F_classical
  contour = mesh.contour(isosurfaces=[isovalue], scalars="f")
  points_np = np.asarray(pcd.points)
  point_cloud = pv.PolyData(points_np)

  # Plotting both the contour and original point cloud
  plotter = pv.Plotter()
  plotter.add_mesh(contour, color="orange", show_edges=False, opacity=0.7, la
  plotter.add_points(point_cloud, color="blue", point_size=4.0, render_points_as
  plotter.add_legend()
  plotter.show()
```

Which is VTK's **marching tetrahedra** under the hood.