robust-laplacian 0.2.4

```
pip install robust-laplacian
```

Latest version

Released: Dec 14, 2021

Robust Laplace matrices for meshes and point clouds

Project description

Project details

Release history

Download files

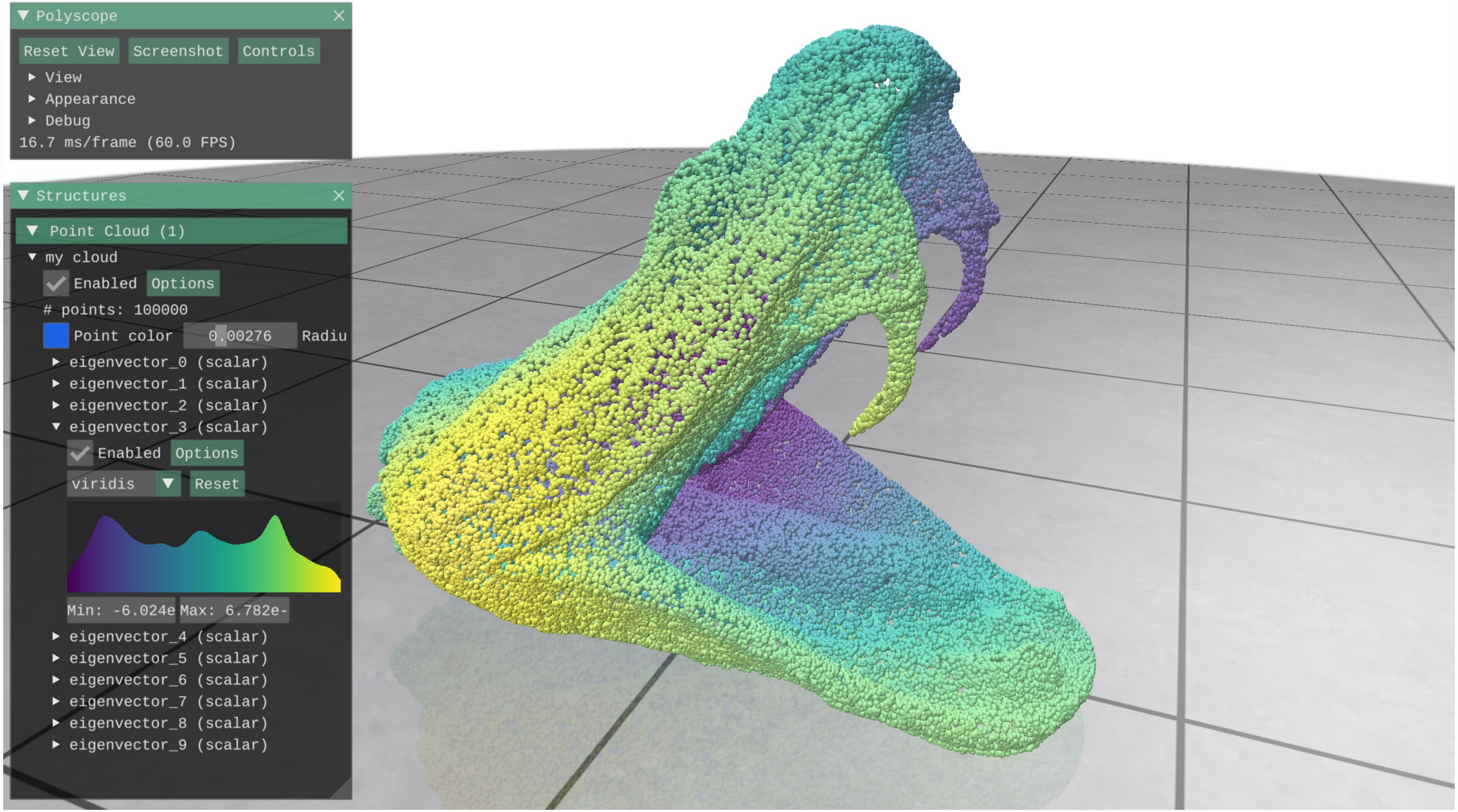
Project description

Test Linux passing Test macOS passing Test Windows passing pypi v0.2.4

A Python package for high-quality Laplace matrices on meshes and point clouds. `pip install robust_laplacian`

The Laplacian is at the heart of many algorithms across geometry processing, simulation, and machine learning. This library builds a high-quality, robust Laplace matrix which often improves the performance of these algorithms, and wraps it all up in a simple, single-function API!

Sample: computing eigenvectors of the point cloud Laplacian



Given as input a triangle mesh with arbitrary connectivity (could be nonmanifold, have boundary, etc), OR a point cloud, this library builds an `NxN` sparse Laplace matrix, where `N` is the number of vertices/points. This Laplace matrix is similar to the *cotan-Laplacian* used widely in geometric computing, but internally the algorithm constructs an *intrinsic Delaunay triangulation* of the surface, which gives the Laplace matrix great numerical properties. The resulting

Laplacian is always a symmetric positive-definite matrix, with all positive edge weights. Additionally, this library performs *intrinsic mollification* to alleviate floating-point issues with degenerate triangles.

The resulting Laplace matrix `L` is a "weak" Laplace matrix, so we also generate a diagonal lumped mass matrix `M`, where each diagonal entry holds an area associated with the mesh element. The "strong" Laplacian can then be formed as `M^-1 L`, or a Poisson problem could be solved as `L x = M y`.

A [C++ implementation and demo]() is available.

This library implements the algorithm described in [A Laplacian for Nonmanifold Triangle Meshes]() by [Nicholas Sharp]() and [Keenan Crane]() at SGP 2020 (where it won a best paper award!). See the paper for more details, and please use the citation given at the bottom if it contributes to academic work.

Example

Build a point cloud Laplacian, compute its first 10 eigenvectors, and visualize with [Polyscope]()

```
pip install numpy scipy plyfile polyscope robust_laplacian
```

```
import robust_laplacian
from plyfile import PlyData
import numpy as np
import polyscope as ps
import scipy.sparse.linalg as sla

# Read input
plydata = PlyData.read("/path/to/cloud.ply")
points = np.vstack((
    plydata['vertex']['x'],
    plydata['vertex']['y'],
    plydata['vertex']['z']
```

```
)).T

# Build point cloud Laplacian
L, M = robust_laplacian.point_cloud_laplacian(points)

# (or for a mesh)
# L, M = robust_laplacian.mesh_laplacian(verts, faces)

# Compute some eigenvectors
n_eig = 10
evals, evecs = sla.eigsh(L, n_eig, M, sigma=1e-8)

# Visualize
ps.init()
ps_cloud = ps.register_point_cloud("my cloud", points)
for i in range(n_eig):
    ps_cloud.add_scalar_quantity("eigenvector_"+str(i), evecs[:,i], enabled=True)
ps.show()
```

NOTE: No one can agree on the sign convention for the Laplacian. This library builds the *positive semi-definite* Laplace matrix, where the diagonal entries are positive and off-diagonal entries are negative. This is the *opposite* of the sign used by e.g. libIGL in `igl.cotmatrix`, so you may need to flip a sign when converting code.

API

This package exposes just two functions:

- `mesh_laplacian(verts, faces, mollify_factor=1e-5)`
    - `verts` is an `V x 3` numpy array of vertex positions
    - `faces` is an `F x 3` numpy array of face indices, where each is a 0-based index referring to a vertex

    - `mollify_factor` amount of intrinsic mollifcation to perform. `0` disables, larger values will increase numerical stability, while very large values will slightly implicitly smooth out the geometry. The range of reasonable settings is roughly `0` to `1e-3`. The default value should usually be sufficient.
    - `return L, M` a pair of scipy sparse matrices for the Laplacian `L` and mass matrix `M`
- `point_cloud_laplacian(points, mollify_factor=1e-5, n_neighbors=30)`
    - `points` is an `V x 3` numpy array of point positions
    - `mollify_factor` amount of intrinsic mollifcation to perform. `0` disables, larger values will increase numerical stability, while very large values will slightly implicitly smooth out the geometry. The range of reasonable settings is roughly `0` to `1e-3`. The default value should usually be sufficient.
    - `n_neighbors` is the number of nearest neighbors to use when constructing local triangulations. This parameter has little effect on the resulting matrices, and the default value is almost always sufficient.
    - `return L, M` a pair of scipy sparse matrices for the Laplacian `L` and mass matrix `M`

Installation

The package is availabe via `pip`

```
pip install robust_laplacian
```

The underlying algorithm is implemented in C++; the pypi entry includes precompiled binaries for many platforms.

Very old versions of `pip` might need to be upgraded like `pip install pip --upgrade` to use the precompiled binaries.

Alternately, if no precompiled binary matches your system `pip` will attempt to compile from source on your machine. This requires a working C++ toolchain, including cmake.

Known limitations

- For point clouds, this repo uses a simple method to generate planar Delaunay triangulations, which may not be totally robust to collinear or degenerate point clouds.

Dependencies

This python library is mainly a wrapper around the implementation in the geometry-central library; see there for further dependencies. Additionally, this library uses pybind11 to generate bindings, and jc_voronoi for 2D Delaunay triangulation on point clouds. All are permissively licensed.

Citation

```
@article{Sharp:2020:LNT,
  author={Nicholas Sharp and Keenan Crane},
  title={{A Laplacian for Nonmanifold Triangle Meshes}},
  journal={Computer Graphics Forum (SGP)},
  volume={39},
  number={5},
  year={2020}
}
```

For developers

This repo is configured with CI on github actions to build wheels across platform.

Deploy a new version

- Commit the desired version to the `master` branch, be sure the version string in `setup.py` corresponds to the new version number. Include the string `[ci build]` in the commit message to ensure a build happens.

- Watch th github actions builds to ensure the test & build stages succeed and all wheels are compiled.
- While you're waiting, update the docs.
- Tag the commit with a tag like `v1.2.3`, matching the version in `setup.py`. This will kick off a new github actions build which deploys the wheels to PyPI after compilation.