SpharaPy Documentation

Release 1.1.1

Uwe Graichen

Jan 31, 2022

CONTENTS

CHAPTER

ONE

QUICK START WITH SPHARAPY

Section contents

In this tutorial, we briefly introduce the vocabulary used in spatial harmonic analysis (SPHARA) and we give a simple learning example to SpharaPy.

1.1 SPHARA – The problem setting

Fourier analysis is one of the standard tools in digital signal and image processing. In ordinary digital image data, the pixels are arranged in a Cartesian or rectangular grid. Performing the Fourier transform, the image data $x[m, n]$ is compared (using a scalar product) with a two-dimensional Fourier basis $f[k, l] = \mathrm{e}^{-2\pi\mathrm{i}\cdot\left(\frac{mk}{M}+\frac{nl}{N}\right)}$. In Fourier transform on a Cartesian grid, the Fourier basis used is usually inherently given in the transformation rule

$$X[k, l] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m, n] \cdot \mathrm{e}^{-2\pi\mathrm{i}\cdot\left(\frac{mk}{M}+\frac{nl}{N}\right)} .$$

A Fourier basis is a solution to Laplace's eigenvalue problem (related to the Helmholtz equation)

$$L\vec{x} = \lambda\vec{x} , \qquad (1)$$

with the discrete *Laplace-Beltrami operator* in matrix notation $L \in \mathbb{R}^{M\times N}$, the eigenvectors $\vec{x}$ containing the harmonic functions and the eigenvalues λ the natural frequencies.

An arbitrary arrangement of sample points on a surface in three-dimensional space can be described by means of a *triangular mesh*. A spatial harmonic basis (**SPHARA basis**) is a solution of a Laplace eigenvalue problem for the given triangle mesh can be obtained by discretizing a Laplace-Beltrami operator for the mesh and solving the Laplace eigenvalue problem in equation (1). SpharaPy provides classes and functions to support these tasks:

- managing triangular meshes describing the spatial arrangement of the sample points,
- determining the Laplace-Beltrami operator of these meshes,
- computing a basis for spatial Fourier analysis of data defined on the triangular mesh, and
- performing the SPHARA transform and filtering.

1.2 The SpharaPy package

The SpharaPy package consists of five modules *spharapy.trimesh*, *spharapy.spharabasis*, *spharapy.spharatransform*, *spharapy.spharafilter* and *spharapy.datasets*. In the following we use three of the five SpharaPy modules to briefly show how a SPHARA basis can be calculated for given spatial sample points. The *spharapy.trimesh* module contains the TriMesh class, which can be used to specify the configuration of the spatial sample points. The SPHARA basis functions can be determined using the *spharapy.spharabasis* module, employing different discretizations. The *spharapy.datasets* module is an interface to the example data sets provided with the SpharaPy package.

```
# Code source: Uwe Graichen
# License: BSD 3 clause

# import modules from spharapy package
import spharapy.trimesh as tm
import spharapy.spharabasis as sb
import spharapy.datasets as sd

# import additional modules used in this tutorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

1.3 Specification of the spatial configuration of the sample points

To illustrate some basic functionality of the SpharaPy package, we load a simple triangle mesh from the example data sets.

```
# loading the simple mesh from spharapy sample datasets
mesh_in = sd.load_simple_triangular_mesh()
```

The imported mesh is defined by a **list of triangles** and a **list of vertices**. The data are stored in a dictionary with the two keys 'vertlist' and 'trilist'

```
print(mesh_in.keys())
```

Out:

```
dict_keys(['vertlist', 'trilist'])
```

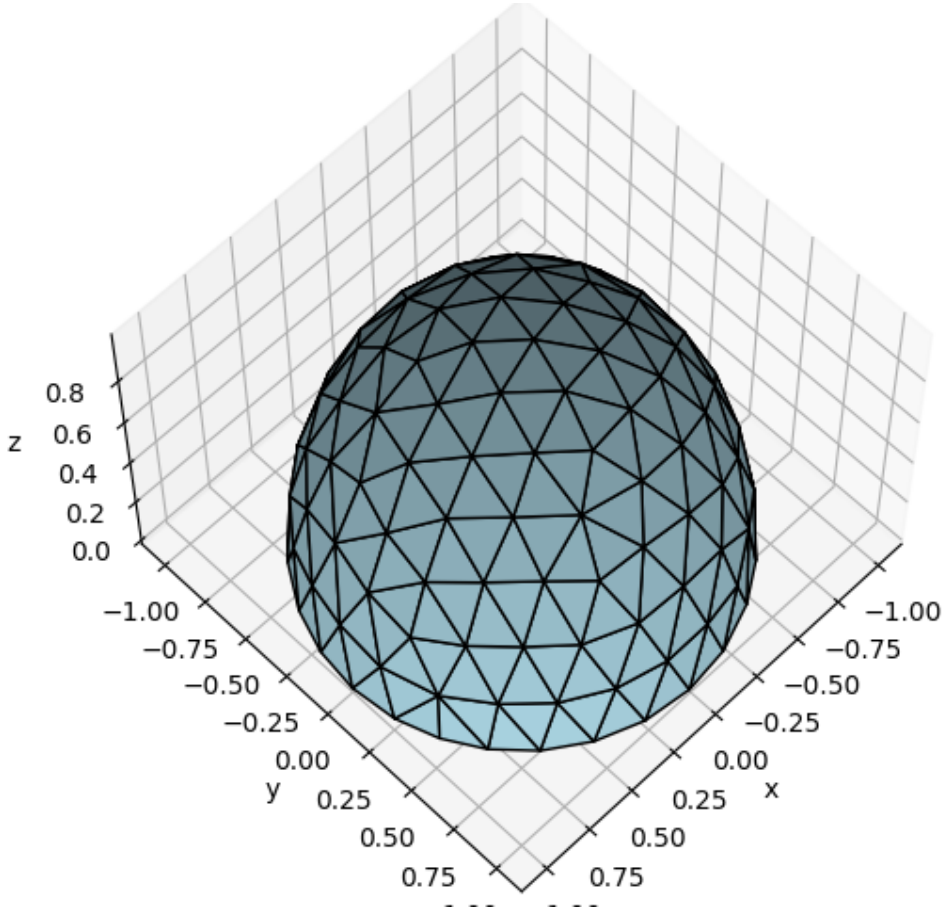
The simple, triangulated surface consists of 131 vertices and 232 triangles and is the triangulation of a hemisphere of an unit ball.

```
vertlist = np.array(mesh_in['vertlist'])
trilist = np.array(mesh_in['trilist'])
print('vertices = ', vertlist.shape)
print('triangles = ', trilist.shape)
```

Out:

```
vertices =  (131, 3)
triangles =  (232, 3)
```

```
fig = plt.figure()
fig.subplots_adjust(left=0.02, right=0.98, top=0.98, bottom=0.02)
ax = fig.gca(projection='3d')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.view_init(elev=60., azim=45.)
ax.set_aspect('auto')
ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1], vertlist[:, 2],
                triangles=trilist, color='lightblue', edgecolor='black',
                linewidth=1)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/spharapy/checkouts/latest/
↪examples/plot_01_quick_start.py:123: MatplotlibDeprecationWarning: Calling␣
↪gca() with keyword arguments was deprecated in Matplotlib 3.4. Starting two␣
↪minor releases later, gca() will take no keyword arguments. The gca()␣
↪function should only be used to get the current axes, or if no axes exist,␣
↪create new axes with default keyword arguments. To create a new axes with␣
↪non-default arguments, use plt.axes() or plt.subplot().
```

(continues on next page)

(continued from previous page)

```
  ax = fig.gca(projection='3d')

<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7f677070f950>
```

1.4 Determining the Laplace-Beltrami Operator

In a further step, an instance of the class *spharapy.trimesh.TriMesh* is created from the lists of vertices and triangles. The class *spharapy.trimesh.TriMesh* provides a number of methods to determine certain properties of the triangle mesh required to generate the SPHARA basis.

```
# print all implemented methods of the TriMesh class
print([func for func in dir(tm.TriMesh) if not func.startswith('__')])
```

Out:

```
['adjacent_tri', 'is_edge', 'laplacianmatrix', 'massmatrix', 'one_ring_
↪neighborhood', 'remove_vertices', 'stiffnessmatrix', 'trilist', 'vertlist',
↪'weightmatrix']
```

```
# create an instance of the TriMesh class
simple_mesh = tm.TriMesh(trilist, vertlist)
```

For the simple triangle mesh an instance of the class SpharaBasis is created and the finite element discretization ('fem') is used. The complete set of SPHARA basis functions and the natural frequencies associated with the basis functions are determined.

```
sphara_basis = sb.SpharaBasis(simple_mesh, 'fem')
basis_functions, natural_frequencies = sphara_basis.basis()
```

The set of SPHARA basis functions can be used for spatial Fourier analysis of the spatially irregularly sampled data.

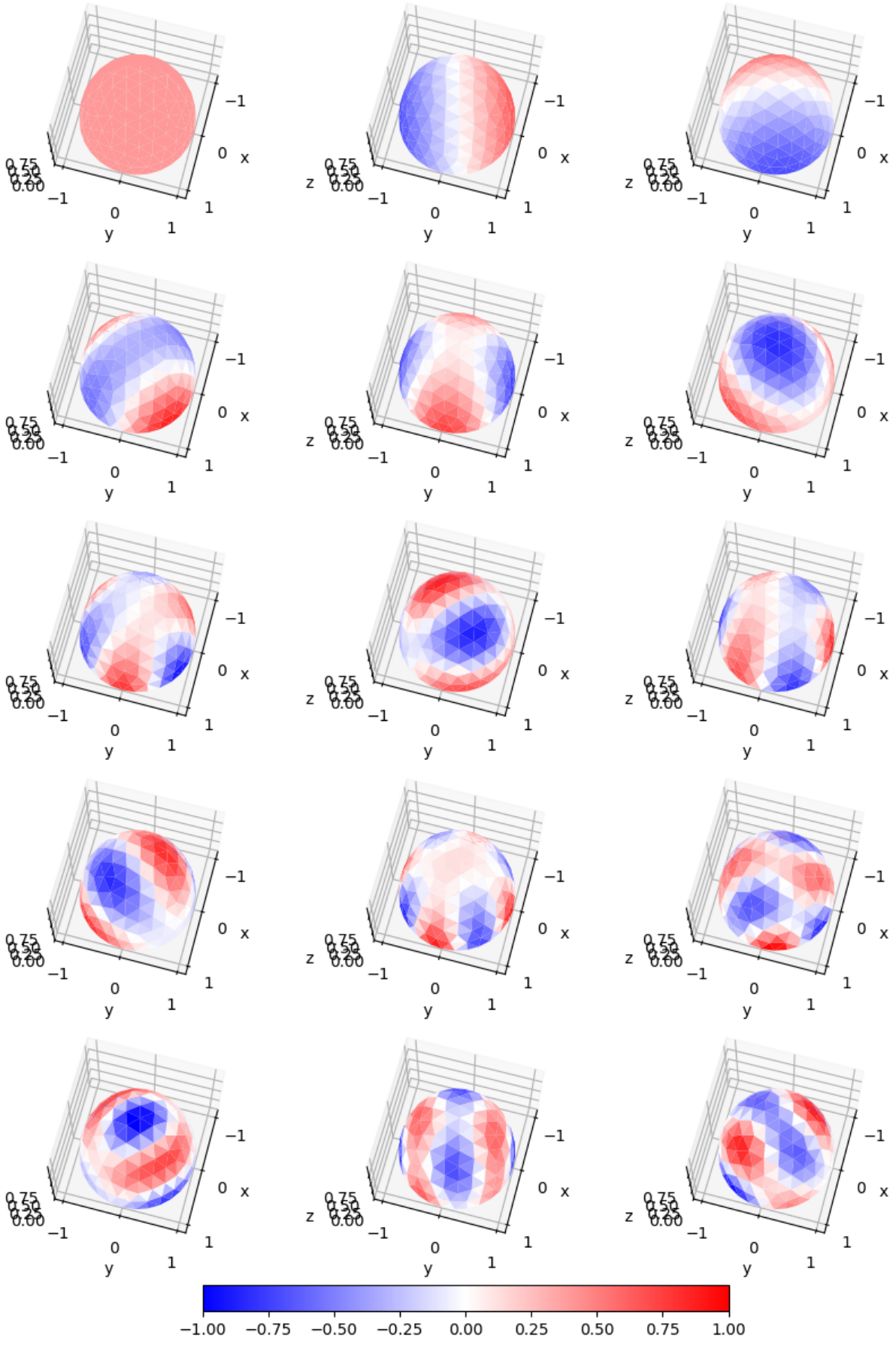
The first 15 spatially low-frequency SPHARA basis functions are shown below, starting with DC at the top left.

```
# sphinx_gallery_thumbnail_number = 2
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=70., azim=15.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
```

(continues on next page)

(continued from previous page)

```
                                    edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.set_clim(-1, 1)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.75,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.0))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```



Total running time of the script: (0 minutes 1.467 seconds)

CHAPTER

TWO

SPHARA – THE THEORETICAL BACKGROUND IN A NUTSHELL

2.1 Motivation

The discrete Fourier analysis of 2D data defined on a flat surface and represented by a Cartesian or a regular grid is very common in digital image processing and a fundamental tool in many applications. For such data, the basis functions (BF) for the Fourier transformation are usually implicitly specified in the transformation rule, compare [RKH10].

However, in many applications the sensors for data acquisition are not located on a flat surface and can not be represented by Cartesian or regular grids. An example from the field of biomedical engineering for non-regular sensor positions is the electroencephalography (*EEG*). In *EEG*, the sensors are placed at predetermined positions at the head surface, a surface in space $\mathbb{R}^3$. The positions of the sensors of these systems can be described by means of triangular meshes. Because of the particular sensor arrangement, the spatial analysis of multi-sensor data can not be performed using the standard 2D Fourier analysis. However, a spatial Fourier analysis can be also very useful for spatially irregularly sampled data.

In this Python package we implement a new method for SPatial HARmonic Analysis (SPHARA) of multisensor data using the eigenbasis of the *Laplace-Beltrami operator* of the meshed surface of sensor positions. Using this approach, basis functions of spatial harmonics for arbitrary arrangements of sensors can be generated. The recorded multisensor data are decomposed by projection into the space of the basis functions. For a much more detailed introduction of the theoretical principles of SPHARA see also [GEF+15].

2.2 Discrete representation of surfaces

For the discrete case we assume that the sensors are located on an arbitrary surface, which is represented by a triangular mesh in $\mathbb{R}^3$. The mesh $M = \{V, E, T\}$ consists of vertices $v \in V$, edges $e \in E$ and triangles $t \in T$. Each vertex $v_i \in \mathbb{R}^3$ represents a sensor position. The number of vertices, edges and triangles of M are defined by $|V|$, $|E|$ and $|T|$, respectively. The neighborhood $i^\star$ for a vertex $v_i \in V$ is defined by $i^\star = \{v_x \in V : e_{ix} \in E\}$, see Fig. 2.1 (b). The number of neighbors of v_i is $n_i = |i^\star|$. The angles α_{ij} and β_{ij} are located opposed to the edge e_{ij}. The triangles t_a and t_b, defined by the vertices (v_i, v_j, v_o) and (v_i, v_k, v_j), share the edge e_{ij}. The set of triangles sharing the vertex v_i is given by $i^\triangledown = \{t_x \in T : v_i \in t_x\}$. The area of a triangle t is given by $|t|$. An example for these mesh components is illustrated in Fig. 2.1 (b).

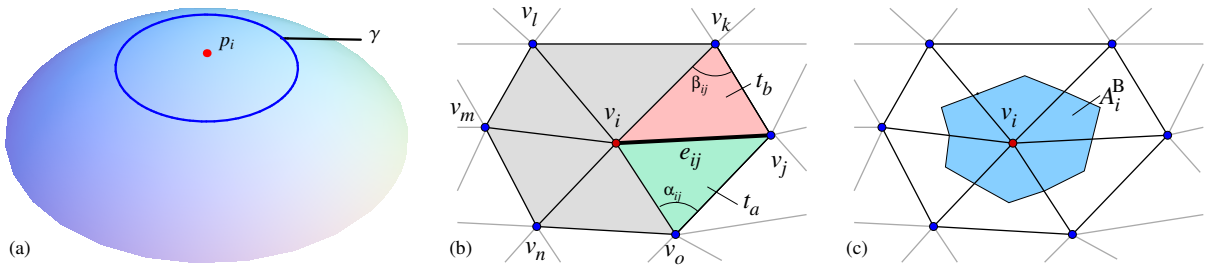


Fig. 2.1: **The approximation of the Laplace-Beltrami operator.** (a) continuous representation; (b) discrete representation. The neighborhood $i^\star$ of vertex v_i consists of the vertices $\{v_x \in V : e_{ix} \in E\}$. Either the length of e_{ij} or the size of the two angles α_{ij} and β_{ij} opposed to the edge e_{ij} are used to estimate the weight $w(i,j)$ for e_{ij}. The two triangles t_a and t_b both share the edge e_{ij}; (c) the area of the barycell A_i^{B} for the vertex v_i.

2.3 Discrete Laplace-Beltrami Operators

A function $\vec{f}$ is defined for all vertices $v_i \in V$, it applies $\vec{f} : v_i \to \mathbb{R}$ with $i = 1, \ldots, |V|$. A discretization Δ_D of the *Laplace-Beltrami operator* for $\vec{f}$ is given by

$$\Delta_D \vec{f}_i = b_i^{-1} \sum_{x \in i^\star} w(i,x) \left(\vec{f}_i - \vec{f}_x \right) , \tag{2.1}$$

with the weighting function $w(i,x)$ for edges $e_{ix} \in E$ and the normalization coefficient b_i for the vertex v_i. For practical applications it is convenient to transform equation (2.1) into matrix notation. The elements of the matrices B^{-1} and S are determined using the coefficients b_i and $w(i,x)$ of equation (2.1). B^{-1} is a diagonal matrix, the elements are

$$B_{ij}^{-1} = \begin{cases} b_i^{-1} & \text{if} \quad i = j \\ 0 & \text{otherwise} , \end{cases} \tag{2.2}$$

and the entries of S are

$$S_{ij} = \begin{cases} \sum_{x \in i^\star} w(i,x) & \text{if} \quad i = j \\ -w(i,x) & \text{if} \quad e_{ix} \in E \\ 0 & \text{otherwise} . \end{cases} \tag{2.3}$$

A Laplacian matrix L can be expressed as product of a diagonal matrix B^{-1} and a matrix S

$$L = B^{-1} S , \tag{2.4}$$

compare also [ZvKD10]. The size of the Laplacian matrix L for a mesh M is $n \times n$, with $n = |V|$. Using the Laplacian matrix L, Δ_D applied to $\vec{f}$ can be written as

$$\Delta_D \vec{f} = -L \vec{f} . \tag{2.5}$$

In the following we present four different approaches to discretize the *Laplace-Beltrami operator*, a graph-theoretical approach and three geometric approaches.

First, we look at the graph-theoretical approach, where the coordinates of the positions of the vertices are not considered. The topological Laplacian results from equation (2.1) by using $w(i,x) = b_i^{-1} = 1$, see also [Tau95][Chu97][ZvKD07]. The graph-theoretical approach will be referred to as TL later in the text.

Second, for inhomogeneous triangular meshes, where the distances between vertices and the sizes of angles and triangles are different, the weighting function w has to be adapted according to the mesh geometry. In these approaches, the positions of the vertices are also considered. They are referred to as geometric approaches. There are different approaches to treat inhomogeneous meshes.

The first possibility is to use the Euclidean distance of adjacent vertices raised to the power of a value α. For equation (2.1) the coefficients $b_i^{-1} = 1$ and $w(i,x) = ||e_{ix}||^\alpha$ are chosen. A common choice is to use the inverse of the Euclidean distance with $\alpha = -1$ [Tau95][Fuj95]. This approach will be referred to later as IE.

The second approach for a geometric discretization of the *Laplace-Beltrami operator* is derived by minimizing the Dirichlet energy for a triangulated mesh [PP93][Pol02]. It uses cotangent weights with

$$w(i,x) = \frac{1}{2}\left(\cot(\alpha_{ix}) + \cot(\beta_{ix})\right) , \tag{2.6}$$

with the two angles α_{ix} and β_{ix} opposed to the edge e_{ix}, see Fig. 2.1 (b). For edges on the boundary of the mesh, the term $\cot(\beta_{ix})$ is omitted, which leads to Neumann *Boundary condition* (BC). A drawback of using the cotangent weights is that the value representing the integral of the Laplacian over a 1-ring neighborhood (area of the $i^\star$-neighborhood) is assigned to a point sample [ZvKD07]. To resolve this issue and to guarantee the correspondence between the continuous and the discrete approaches, the weights in equation (2.6) are divided by the area A_i^{B} of the barycell for the vertex v_i [MDSB03], resulting in

$$w(i,x) = \frac{1}{2A_i^{\mathrm{B}}}\left(\cot(\alpha_{ix}) + \cot(\beta_{ix})\right) . \tag{2.7}$$

The barycell for a vertex v_i is framed by a polygonal line that connects the geometric centroids of triangles in $i^\triangledown$ and the midpoints of the adjoined edges e_{ix}, see Fig. 2.1 (c). The area of the $i^\star$-neighborhood for a vertex v_i, which is the area of the triangles that are enclosed by the vertices $v_x \in i^\star$, is referred to as A_i^1. Then A_i^{B} can be determined by $A_i^{\mathrm{B}} = \frac{1}{3}A_i^1$. For the discretizations using the cotangent weighted formulation, the parameter b_i^{-1} in equation (2.1) is set to $b_i^{-1} = 1$. This approach, using cotangent weights will be referred to as COT later in the manuscript.

The third geometric approach to discretize the Laplace-Beltrami operator is the *Finite Element Method* (FEM), which is related to the approach using cotangent weights. Assuming that the function f is piecewise linear and defined by its values f_i on the vertices v_i of a triangular mesh, f can be interpolated using nodal basis functions ψ_i

$$f = \sum_{i=1}^{|V|} f_i\,\psi_i\,. \tag{2.8}$$

We use the hat function for ψ_i, with

$$\psi_i(j) = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{otherwise}\,. \end{cases} \tag{2.9}$$

For two functions f and g defined on M, a scalar product is given by

$$\int_M fg\,\mathrm{d}a = \sum_{i=0}^{|V|}\sum_{j=0}^{|V|} f_i g_i \int_M \psi_i\psi_j\,\mathrm{d}a = \left\langle \vec{f},\vec{g}\right\rangle_B , \tag{2.10}$$

with the area element $\mathrm{d}a$ on M and the mass matrix B. The sparse mass matrix B is given by

$$B_{ij} = \int_M \psi_i\psi_j\,\mathrm{d}a\,. \tag{2.11}$$

For the FEM approach using hat functions, the elements of B can be calculated by

$$B_{ij} = \begin{cases} \left(\sum_{t \in i^{\triangledown}} |t|\right)/6 & \text{if} \quad i = j \\ \left(|t_a| + |t_b|\right)/12 & \text{if} \quad e_{ij} \in E \\ 0 & \text{otherwise}\,, \end{cases} \tag{2.12}$$

where t_a and t_b are the two triangles adjacent to the edge e_{ij}, see Fig. 2.1 (b). For the FEM discretization of the *Laplace-Beltrami operator* also a stiffness matrix S has to be calculated. The elements of S_{ij} can be estimated using the equations (2.3) and (2.6), compare [DZMC07][ZvKD07][VL07].

2.4 Eigensystems of Discrete Laplace-Beltrami Operators

Desirable properties of the discrete Laplacian L are symmetry, positive weights, positive semi-definiteness, locality, linear precision and convergence [WMKG07]. The symmetry $L_{ij} = L_{ji}$ leads to real eigenvalues and orthogonal eigenvectors. Positive weights $w(i,j) \geq 0$ assure, together with the symmetry, the positive semi-definiteness of L. The locality of the discrete *Laplace-Beltrami operator* enables the determination of weights $w(i,j)$ using the $i^{\star}$-neighborhood of a vertex v_i, with $w(i,j) = 0$, if $e_{ij} \notin E$. The linear precision implies for a linear function f defined on vertices v_i that $\Delta_D \vec{f}_i = 0$ applies, which ensures the exact recovery of f from the samples. The convergence property provides the convergence from the discrete to the continuous *Laplace-Beltrami operator* $\Delta_D \rightarrow \Delta$ for a sufficient refinement of the mesh.

The Laplacian matrix L for the TL and the IE approach are positive semi-definite, symmetric and use positive weights. The COT and the FEM approach do not fulfill the positive weight property, if the mesh contains triangles with interior angles in the interval $(\pi/2, \pi)$, for which the cotangent is negative. The TL approach is no geometric discretization, because it violates the linear precision and the convergence property. In contrast, the COT and the FEM approach are geometric discretizations as they fulfill the linear precision and the convergence property, but they violate the symmetry property. None of the presented discretization methods fulfill all desirable properties, see also [WMKG07].

The discrete Laplacian eigenvalue problem for a real and symmetric Laplacian matrix is given by

$$L\,\vec{x}_i = \lambda_i\,\vec{x}_i\,, \tag{2.13}$$

with eigenvectors $\vec{x}_i$ and eigenvalues λ_i of L. The Laplacian matrix L is real and symmetric for the TL, the IE and the COT approach. Because L is real and symmetric, eigenvalues $\lambda_i \in \mathbb{R}$ with $\lambda_i \geq 0$ are obtained. The eigenvectors $\vec{x}_i$ are real-valued and form a harmonic orthonormal basis. The corresponding eigenvalues λ_i can be considered as spatial frequencies. The eigenvectors $\vec{x}_i$ can be used for a spectral analysis of functions defined on the mesh M. The projection of a discrete function $\vec{f}$ defined on M onto the basis of spatial harmonic functions is performed by the inner product for Euclidean n-spaces $\langle \vec{f}, \vec{x}_i \rangle$. For the matrix X, where the eigenvectors $\vec{x}_i$ represent the columns,

$$X = [\vec{x}_1\ \vec{x}_2 \cdots \vec{x}_n]\,,$$

it applies

$$X^{\top} X = I\,, \tag{2.14}$$

with the identity matrix I.

For the FEM formulation, the BF $\vec{y}_i$ are computed by solving the generalized symmetric definite eigenproblem

$$S\vec{y}_i = \lambda_i B \vec{y}_i\,. \tag{2.15}$$

Thus, the inversion of the mass matrix B is avoided. Because $B^{-1}S$ is not symmetric, the eigenvectors $\vec{y}_i$ are real-valued, but not orthonormal with respect to the inner product for Euclidean n-spaces $\langle . \rangle$. To use these eigenvectors as BF, the inner product, defined in equation~(ref{eq:scalarproductfem}), has to be used

$$\left\langle \vec{f}, \vec{y}_i \right\rangle_B = \vec{f}^{\top} B \, \vec{y}_i \,, \tag{2.16}$$

which assures the B-orthogonality, compare also cite{vallet07}. The eigenvectors computed by the FEM approach can be normalized by using the B-relative norm

$$\vec{\tilde{y}}_i = \frac{\vec{y}_i}{\|\vec{y}_i\|_B} \quad \text{with} \quad \|\vec{y}_i\|_B = \sqrt{\langle \vec{y}_i, \vec{y}_i \rangle_B} \,. \tag{2.17}$$

For a matrix $\tilde{Y}$, where the normalized eigenvectors $\vec{\tilde{y}}_i$ represent the columns

$$\tilde{Y} = [\vec{\tilde{y}}_1 \; \vec{\tilde{y}}_2 \cdots \vec{\tilde{y}}_n] \,, \tag{2.18}$$

it applies

$$\tilde{Y}^{\top} B \tilde{Y} = I \,. \tag{2.19}$$

2.5 SPHARA as signal processing framework

2.5.1 Requirements

To use the eigenvectors of the discrete Laplacian Beltrami operator in the context of a signal processing framework, it is necessary that they exhibit certain properties. The eigenvectors have to form a set of BF. An inner product for the decomposition and for the reconstruction of the data has to be defined, which is used for the transformation into the domain of the spatial frequencies and for the back-transformation into the spatial domain. To be utilized for practical applications, the transformation from the spatial domain into the spatial frequency domain has to have linear properties and should fulfill Parseval's theorem.

2.5.2 Basis functions

A complete set of linearly independent vectors can be used as basis. For real and symmetric Laplacian matrices L the orthonormality of the eigenvectors is given inherently, see equations (2.13) and (2.14). For the FEM approach, the orthonormality of the eigenvectors is assured explicitly, see equations (2.15) to (2.19). The property of orthogonality includes the linear independence. To use the eigenvectors as BF, they must further fulfill the property of completeness. The completeness can be shown by the dimension theorem for vector spaces. The dimensionality is equal for both the spatial representation and the representation in the spatial frequency domain. For a mesh with n vertices, n unit impulse functions are used as BF for the spatial representation. For the same mesh, we obtain n discrete spatial harmonic functions (eigenvectors) for the representation using spatial frequencies. The calculated eigenvectors are orthonormal and complete; therefore, they can be used as orthonormal BF.

2.5.3 Analysis and synthesis

For the analysis of discrete data defined on the vertices of the triangular mesh, the inner product is used (transformation from spatial domain to spatial frequency domain). For an analysis using the eigenvectors of a symmetric Laplacian matrix L (TL, IE and COT), the vector space inner product is applied. The coefficient c_i for a single spatial harmonic BF $\vec{x}_i$ can be determined by

$$c_i = \langle \vec{f}, \vec{x}_i \rangle \,. \tag{2.20}$$

The transformation from the spatial into the spatial frequency domain is computed by

$$\vec{c}^{\,\top} = \vec{f}^{\,\top} X \,. \tag{2.21}$$

For an analysis using eigenvectors computed by the FEM approach, the inner product that assures the B-orthogonality needs to be applied

$$c_i = \left\langle \vec{f}, \vec{\tilde{y}}_i \right\rangle_B = \vec{f}^{\,\top} B \, \vec{\tilde{y}}_i \,. \tag{2.22}$$

The transformation from the spatial into the spatial frequency domain is then be computed by

$$\vec{c}^{\,\top} = \vec{f}^{\,\top} B \, \tilde{Y} \,. \tag{2.23}$$

Discrete data are synthesized using the linear combination of the coefficients c_i and the corresponding BF $\vec{x}_i$ or $\vec{\tilde{y}}_i$

$$\vec{f} = \sum_{i=1}^{n} c_i \, \vec{x}_i \tag{2.24}$$

or

$$\vec{f}^{\,\top} = \vec{c}^{\,\top} \tilde{Y}^{\,\top} \,. \tag{2.25}$$

2.6 Spatial filtering using SPHARA

At the end of this short introduction we show the design of a spatial filter as a practical application of SPHARA. The prerequisite for the successful use of SPHARA-based filters is the separability of useful signal and interference in the spatial SPHARA spectrum. This applies, for example, to *EEG*. In *EEG*, the low-frequency SPHARA basis functions provide the main contribution to signal power. In contrast, single channel dropouts and spatially uncorrelated sensor noise exhibit an almost equally distributed spatial SPHARA spectrum, compare [GEF+15] and tutorial *Spatial SPHARA analysis of EEG data*.

A filter matrix F can be determined by

$$F = X \cdot R \cdot (X \cdot R)^{\intercal} \,.$$

The matrix X contains columnwise the SPHARA basis functions and the matrix R is a selection matrix, that contains an 1 on the main diagonal if the corresponding SPHARA basis function from X is chosen. All other elements of this matrix are 0.

If the Laplace-Beltrami Operator with FEM discretization is used to calculate the SPHARA basis functions, the mass matrix B must be added to the equation to compute the filter matrix

$$F_{\mathrm{FEM}} = B \cdot X \cdot R \cdot (X \cdot R)^{\intercal} \,.$$

The spatial SPHARA filter is applied to the data by multiplying the matrix containing the data D by the filter matrix F

$$\tilde{D} = D \cdot F \,.$$

The matrix D contains data, time samples in rows and spatial samples in columns and the matrix $\tilde{D}$ the filtered data, see also tutorial *Spatial SPHARA filtering of EEG data*.

CHAPTER

THREE

API REFERENCE

The Python toolbox SpharaPy contains following modules:

3.1 spharapy.trimesh: Triangular Mesh Data

Triangular mesh data

This module provides a class for storing triangular meshes. Attributes of the triangular mesh can be determined. In addition, methodes are available to derive further information from the triangular grid.

class spharapy.trimesh.**TriMesh**(*trilist*, *vertlist*)

Bases: object

Triangular mesh class

This class can be used to store data to define a triangular mesh and it provides atributes and methodes to derive further information about the triangular mesh.

Parameters

trilist: array, shape (n_triangles, 3) List of triangles, each row of the array contains the edges of a triangle. The edges of the triangles are defined by the indices to the list of vertices. The index of the first vertex is 0. The number of triangles is n_triangles.

vertlist: array, shape (n_points, 3) List of coordinates x, y, z which describes the positions of the vertices.

Attributes

trilist: array, shape (n_triangles, 3) List of triangles of the mesh.

vertlist: array, shape (n_points, 3) List of coordinates of the vertices

adjacent_tri(*vertex_index=0*)

All triangles with the given vertex

The method determined all triangles of the triangular mesh that contain the given vertex.

Parameters

vertex_index [integer] Index of the vertex for which the adjacent vertices are to be determined. The index must be in the range 0 to number of vertices - 1.

Returns

tri_with_vertex [array, shape (3, n)] List of triangles containing the given vertex.

`is_edge`(*vertex1_index*, *vertex2_index*)

Are 2 vertices connected by an edge

The method determines whether two vertices are connected by an edge in the triangle mesh and if so, whether it is an internal edge or a boundary edge.

Parameters

vertex1_index, vertex2_index [integer] Indeces of the two vertices. The index must be in the range 0 to number of vertices - 1.

Returns

is_edge [integer] 0 if vertex1 and vertex2 are not connected by a single edge, 1 if vertex1 and vertex2 are connected by a boundary edge, 2 if vertex1 and vertex2 are connected by an internal edge.

`laplacianmatrix`(*mode='inv_euclidean'*)

Compute a laplacian matrix for a triangular mesh

The method creates a laplacian matrix for a triangular mesh using different weighting function.

Parameters

mode [{'unit', 'inv_euclidean', 'half_cotangent'}, optional] The methods for determining the edge weights. Using the option 'unit' all edges of the mesh are weighted by unit weighting function, the result is an adjacency matrix. The option 'inv_euclidean' results in edge weights corresponding to the inverse Euclidean distance of the edge lengths. The option 'half_cotangent' uses the half of the cotangent of the two angles opposed to an edge as weighting function. the default weighting function is 'inv_euclidean'.

Returns

laplacianmatrix [array, shape (n_points, n_points)] Matrix, which contains the discrete laplace operator for data defined at the vertices of a triangular mesh. The number of vertices of the triangular mesh is n_points.

Examples

```
>>> from spharapy import trimesh as tm
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                         [0., 0., 3.]])
>>> testtrimesh.laplacianmatrix(mode='inv_euclidean')
array([[ 0.76344136, -0.4472136 , -0.31622777],
       [-0.4472136 ,  0.72456369, -0.2773501 ],
       [-0.31622777, -0.2773501 ,  0.59357786]])
```

`massmatrix`(*mode='normal'*)

Mass matrix of a triangular mesh

The method determines a mass matrix of a triangular mesh.

Parameters

mode [{'normal', 'lumped'}, optional] The *mode* parameter can be used to select whether a normal mass matrix or a lumped mass matrix is to be determined.

Returns

massmatrix [array, shape (n_points, n_points)] Symmetric matrix, which contains the mass values for each edge and vertex for the FEM approch. The number of vertices of the triangular mesh is n_points.

References

[VL07][DZMC07][ZvKD07]

Examples

```
>>> from spharapy import trimesh as tm
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                       [0., 0., 3.]])
>>> testtrimesh.massmatrix()
array([[ 0.58333333,  0.29166667,  0.29166667],
       [ 0.29166667,  0.58333333,  0.29166667],
       [ 0.29166667,  0.29166667,  0.58333333]])
```

one_ring_neighborhood(*vertex_index=0*)

The 1 ring neighborhood of a vertex

The method determines all adjacent vertices of a vertex that is given by its index, the so called 1 ring neighborhood.

Parameters

vertex_index [integer] Index of the vertex for which the adjacent vertices are to be determined. The index must be in the range 0 to number of vertices - 1.

Returns

one_ring_neighborhood [array, shape (1, n)] Array of indexes on vertices adjacent to a given vertex.

remove_vertices(*vertex_index_list*)

Remove vertices from a triangular mesh

The method removes vertices from a triangle mesh. The Half-edge Collapse method is used. The positions of the remaining vertices are not affected and they are retriangulated.

Parameters

vertex_index_list [vector of ints] Indices of the vertices to remove from the mesh. The indices must be in the range 0 to number of vertices - 1.

Returns

triangsamples [trimesh object] A trimesh object from the package spharapy, where the given vertices are removed.

stiffnessmatrix()

Stiffness matrix of a triangular mesh

The method determines a stiffness matrix of a triangular mesh.

Returns

stiffmatrix [array, shape (n_points, n_points)] Symmetric matrix, which contains the stiffness values for each edge and vertex for the FEM approch. The number of vertices of the triangular mesh is n_points.

References

[VL07]

Examples

```
>>> from spharapy import trimesh as tm
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                         [0., 0., 3.]])
>>> testtrimesh.stiffnessmatrix()
array([[-0.92857143,  0.64285714,  0.28571429],
       [ 0.64285714, -0.71428571,  0.07142857],
       [ 0.28571429,  0.07142857, -0.35714286]])
```

property trilist

Get or set the list of triangles.

Setting the list of triangles will simultaneously check if the triangle list is in the correct format.

property vertlist

Get or set the list of vertices.

Setting the list of triangles will simultaneously check if the vertice list is in the correct format.

weightmatrix(*mode='inv_euclidean'*)

Compute a weight matrix for a triangular mesh

The method creates a weighting matrix for the edges of a triangular mesh using different weighting function.

Parameters

mode [{'unit', 'inv_euclidean', 'half_cotangent'}, optional] The parameter *mode* specifies the method for determining the edge weights. Using the option 'unit' all edges of the mesh are weighted by unit weighting function, the result is an adjacency matrix. The option 'inv_euclidean' results in edge weights corresponding to the inverse Euclidean distance of the edge lengths. The option 'half_cotangent' uses the half of the cotangent of the two angles opposed to an edge as weighting function. the default weighting function is 'inv_euclidean'.

Returns

weightmatrix [array, shape (n_points, n_points)] Symmetric matrix, which contains the weight of the edges between adjacent vertices. The number of vertices of the triangular mesh is n_points.

Examples

```
>>> from spharapy import trimesh as tm
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                         [0., 0., 3.]])
>>> testtrimesh.weightmatrix(mode='inv_euclidean')
array([[ 0.        ,  0.4472136 ,  0.31622777],
       [ 0.4472136 ,  0.        ,  0.2773501 ],
       [ 0.31622777,  0.2773501 ,  0.        ]])
```

spharapy.trimesh.**angles_triangle**(*vertex1*, *vertex2*, *vertex3*)

Estimate the three internal angles of a triangle given by three vertices

Parameters

vertex1 [array, shape (1, 3)]

vertex2 [array, shape (1, 3)]

vertex3 [array, shape (1, 3)]

Returns

angles [array, shape (1, 3)] Internal angles of the triangle given by the three vertices.

Examples

```
>>> from spharapy import trimesh as tm
>>> tm.angles_triangle([1, 0, 0], [0, 1, 0], [0, 0, 1])
array([1.04719755, 1.04719755, 1.04719755])
```

spharapy.trimesh.**area_triangle**(*vertex1*, *vertex2*, *vertex3*)

Estimate the area of a triangle given by three vertices

The area of the triangle given by three vertices is calculated by the half cross product formula.

Parameters

vertex1 [array, shape (1, 3)]

vertex2 [array, shape (1, 3)]

vertex3 [array, shape (1, 3)]

Returns

trianglearea [float] Area of the triangle given by the three vertices.

Examples

```
>>> from spharapy import trimesh as tm
>>> tm.area_triangle([1, 0, 0], [0, 1, 0], [0, 0, 1])
0.8660254037844386
```

spharapy.trimesh.**side_lens_triangle**(*vertex1*, *vertex2*, *vertex3*)

Estimate the three side length of a triangle given by three vertices

Parameters

vertex1 [array, shape (1, 3)]

vertex2 [array, shape (1, 3)]

vertex3 [array, shape (1, 3)]

Returns

side_lens [array, shape (1, 3)] Side lengths of the triangle given by the three vertices.

Examples

```
>>> from spharapy import trimesh as tm
>>> tm.side_lens_triangle([1, 0, 0], [0, 1, 0], [0, 0, 1])
array([1.41421356, 1.41421356, 1.41421356])
```

3.2 spharapy.spharabasis: SPHARA Basis

SPHARA basis functions

This module provides a class for determining SPHARA basis functions. Methods are provided to determine basis functions using different discretization schemes of the Laplace-Beltrami operator, as FEM, inverse euclidean and unit.

class spharapy.spharabasis.**SpharaBasis**(*triangsamples=None*, *mode='fem'*)

Bases: object

SPHARA basis functions class

This class can be used to determine SPHARA basis functions for spatially irregularly sampled functions whose topology is described by a triangular mesh.

Parameters

triangsamples [trimesh object] A trimesh object from the package spharapy in which the triangulation of the spatial arrangement of the sampling points is stored. The SPHARA basis functions are determined for this triangulation of the sample points.

mode [{'unit', 'inv_euclidean', 'fem'}, optional] The discretization method used to estimate the Laplace-Beltrami operator. Using the option 'unit' all edges of the mesh are weighted by unit weighting function. The option 'inv_euclidean' results in edge weights corresponding to the inverse Euclidean distance of the

edge lengths. The option 'fem' uses a FEM discretization. The default weighting function is 'fem'.

Attributes

triangsamples: trimesh object Triangulation of the spatial arrangement of the sampling points

mode: {'unit', 'inv_euclidean', 'fem'} Discretization used to estimate the Laplace-Beltrami operator

basis()

Return the SPHARA basis for the triangulated sample points

This method determines a SPHARA basis for spatially distributed sampling points described by a triangular mesh. A discrete Laplace-Beltrami operator in matrix form is determined for the given triangular grid. The discretization methods for determining the Laplace-Beltrami operator is specified in the attribute *mode*. The eigenvectors $\vec{x}$ and the eigenvalues λ of the matrix L containing the discrete Laplace-Beltrami operator are the SPHARA basis vectors and the natural frequencies, respectively, $L\vec{x} = \lambda\vec{x}$.

Parameters

Returns

basis [array, shape (n_points, n_points)] Matrix, which contains the SPHARA basis functions column by column. The number of vertices of the triangular mesh is n_points.

frequencies [array, shape (n_points, 1)] The natural frequencies associated to the SPHARA basis functions.

Examples

```
>>> from spharapy import trimesh as tm
>>> from spharapy import spharabasis as sb
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                        [0., 0., 3.]])
>>> sb_fem = sb.SpharaBasis(testtrimesh, mode='fem')
>>> sb_fem.basis()
(array([[ 0.53452248, -0.49487166,  1.42857143],
        [ 0.53452248, -0.98974332, -1.14285714],
        [ 0.53452248,  1.48461498, -0.28571429]]),
 array([  2.33627569e-16,   1.71428571e+00,   5.14285714e+00]))
```

massmatrix()

Return the massmatrix

The method returns the mass matrix of the triangular mesh.

property mode

Get or set the discretization method.

The discretization method used to estimate the Laplace-Beltrami operator, choosen from {'unit', 'inv_euclidean', 'fem'}. Setting the triangsamples object will simultaneously check the correct format.

property triangsamples
Get or set the triangsamples object.

The parameter *triangsamples* has to be an instance of the class *spharapy.trimesh.TriMesh*. Setting the triangsamples object will simultaneously check the correct format.

3.3 spharapy.spharatransform: SPHARA Transform

SPHARA transform

This module provides a class to perform the SPHARA transform. The class is derived from *spharapy.spharabasis.SpharaBasis*. It provides methodes the SPHARA anaylsis and synthesis of spatially irregularly sampled data.

class spharapy.spharatransform.**SpharaTransform**(*triangsamples=None*, *mode='fem'*)
Bases: *spharapy.spharabasis.SpharaBasis*

SPHARA transform class

This class is used to perform the SPHARA forward (analysis) and inverse (synthesis) transformation.

Parameters

triangsamples [trimesh object] A trimesh object from the package spharapy in which the triangulation of the spatial arrangement of the sampling points is stored. The SPHARA basic functions are determined for this triangulation of the sample points.

mode [{'unit', 'inv_euclidean', 'fem'}, optional] The discretisation method used to estimate the Laplace-Beltrami operator. Using the option 'unit' all edges of the mesh are weighted by unit weighting function. The option 'inv_euclidean' results in edge weights corresponding to the inverse Euclidean distance of the edge lengths. The option 'fem' uses a FEM discretisation. The default weighting function is 'fem'.

analysis(*data*)
Perform the SPHARA transform (analysis)

This method performs the SPHARA transform (analysis) of data defined at spatially distributed sampling points described by a triangular mesh. The forward transformation is performed by matrix multiplication of the data matrix and the matrix with SPHARA basis functions $\tilde{X} = X \cdot S$, with the SPHARA basis S, the data matrix X and the SPHARA coefficients matix $\tilde{X}$. In the forward transformation using SPHARA basic functions determined by discretization with FEM approach, the modified scalar product including the mass matrix is used $\tilde{X} = X \cdot B \cdot S$, with the mass matrix B.

Parameters

data [array, shape(m, n_points)] A matrix with data to be transformed (analyzed) by SPHARA. The number of vertices of the triangular mesh is

n_points. The order of the spatial sample points must correspond to that in the vertex list used to determine the SPHARA basis functions.

Returns

coefficients [array, shape (m, n_points)] A matrix containing the SPHARA coefficients. The coefficients are sorted column by column with increasing spatial frequency, starting with DC in the first column.

Examples

Import the necessary packages

```
>>> import numpy as np
>>> from spharapy import trimesh as tm
>>> from spharapy import spharatransform as st
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                         [0., 0., 3.]])
>>> st_fem_simple = st.SpharaTransform(testtrimesh, mode='fem')
>>> data = np.concatenate([[[0., 0., 0.], [1., 1., 1.]],
...                        np.transpose(st_fem_simple.basis()[0])])
>>> data
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.53452248,  0.53452248,  0.53452248],
       [-0.49487166, -0.98974332,  1.48461498],
       [ 1.42857143, -1.14285714, -0.28571429]])
>>> coef_fem_simple = st_fem_simple.analysis(data)
>>> coef_fem_simple
array([[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
       [  1.87082869e+00,   1.09883582e-16,  -4.18977022e-16],
       [  1.00000000e+00,  -2.75573800e-16,  -8.86630311e-18],
       [ -1.14766454e-16,   1.00000000e+00,   2.30648330e-16],
       [  6.52367763e-17,   1.68383874e-16,   1.00000000e+00]])
```

synthesis(*coefficients*)

Perform the inverse SPHARA transform (synthesis)

This method performs the inverse SPHARA transform (synthesis) for data defined at spatially distributed sampling points described by a triangular mesh. The forward transformation is performed by matrix multiplication of the data matrix and the matrix with SPHARA basis functions $\tilde{X} = X \cdot S$, with the SPHARA basis S, the data matrix X and the SPHARA coefficients matix $\tilde{X}$. In the forward transformation using SPHARA basic functions determined by discretization with FEM approach, the modified scalar product including the mass matrix is used $\tilde{X} = X \cdot B \cdot S$, with the mass matrix B.

Parameters

coefficients [array, shape (m, n_points)] A matrix containing the SPHARA coefficients. The coefficients are sorted column by column with increasing spatial frequency, starting with DC in the first column.

Returns

data [array, shape(m, n_points)] A matrix with data to be forward transformed (analyzed) by SPHARA. The number of vertices of the triangular mesh is n_points. The order of the spatial sample points must correspond to that in the vertex list used to determine the SPHARA basis functions.

Examples

```
>>> import numpy as np
>>> from spharapy import trimesh as tm
>>> from spharapy import spharatransform as st
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                        [0., 0., 3.]])
>>> st_fem_simple = st.SpharaTransform(testtrimesh, mode='fem')
>>> data = np.concatenate([[[0., 0., 0.], [1., 1., 1.]],
...                        np.transpose(st_fem_simple.basis()[0])])
>>> data
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.53452248,  0.53452248,  0.53452248],
       [-0.49487166, -0.98974332,  1.48461498],
       [ 1.42857143, -1.14285714, -0.28571429]])
>>> coef_fem_simple = st_fem_simple.analysis(data)
>>> coef_fem_simple
array([[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
       [  1.87082869e+00,   1.09883582e-16,  -4.18977022e-16],
       [  1.00000000e+00,  -2.75573800e-16,  -8.86630311e-18],
       [ -1.14766454e-16,   1.00000000e+00,   2.30648330e-16],
       [  6.52367763e-17,   1.68383874e-16,   1.00000000e+00]])
>>> recon_fem_simple = st_fem_simple.synthesis(coef_fem_simple)
>>> recon_fem_simple
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.53452248,  0.53452248,  0.53452248],
       [-0.49487166, -0.98974332,  1.48461498],
       [ 1.42857143, -1.14285714, -0.28571429]])
```

3.4 spharapy.spharafilter: SPHARA Filter

SPHARA filter

This module provides a class to perform a spatial filtering using a SPHARA basis. The class is derived from *spharapy.spharabasis.SpharaBasis*. It provides methodes to design different types of filters and to apply this filters to spatially irregularly sampled data.

class spharapy.spharafilter.**SpharaFilter**(*triangsamples=None*, *mode='fem'*, *specification=0*)

Bases: *spharapy.spharabasis.SpharaBasis*

SPHARA filter class

This class is used to design different types of filters and to apply this filters to spatially irregularly sampled data.

Parameters

triangsamples [trimesh object] A trimesh object from the package spharapy in which the triangulation of the spatial arrangement of the sampling points is stored. The SPHARA basic functions are determined for this triangulation of the sample points.

mode [{'unit', 'inv_euclidean', 'fem'}, optional] The discretisation method used to estimate the Laplace-Beltrami operator. Using the option 'unit' all edges of the mesh are weighted by unit weighting function. The option 'inv_euclidean' results in edge weights corresponding to the inverse Euclidean distance of the edge lengths. The option 'fem' uses a FEM discretisation. The default weighting function is 'fem'.

specification [integer or array, shape (1, n_points)] If an integer value for specification is passed to the constructor, it must be within the interval (-n_points, n_points), where n_points is the number of spatial sample points. If a positive integer value is passed, a spatial low-pass filter with the corresponding number of SPHARA basis functions is created, if a negative integer value is passed, a spatial low-pass filter is created. If a vector is passed, then all SPHARA basis functions corresponding to nonzero elements of the vector are used to create the filter. The default value of specification is 0, it means a neutral all-pass filter is designed and applied.

`filter`(*data*)

Perform the SPHARA filtering

This method performs the spatial SPHARA filtering for data defined at spatially distributed sampling points described by a triangular mesh. The filtering is performed by matrix multiplication of the data matrix and a precalculated filter matrix.

Parameters

data [array, shape(m, n_points)] A matrix with data to be filtered by spatial SPHARA filter. The number of vertices of the triangular mesh is n_points. The order of the spatial sample points must correspond to that in the vertex list used to determine the SPHARA basis functions.

Returns

data_filtered [array, shape (m, n_points)] A matrix containing the filtered data.

Examples

```
>>> import numpy as np
>>> from spharapy import trimesh as tm
>>> from spharapy import spharafilter as sf
>>> # define the simple test mesh
>>> testtrimesh = tm.TriMesh([[0, 1, 2]], [[1., 0., 0.], [0., 2., 0.
↪],
...                                         [0., 0., 3.]])
```

(continues on next page)

(continued from previous page)

```
>>> # create a spatial lowpass filter, FEM discretisation
>>> sf_fem = sf.SpharaFilter(testtrimesh, mode='fem',
...                          specification=[1., 1., 0.])
>>> # create some test data
>>> data = np.concatenate([[[0., 0., 0.], [1., 1., 1.]],
...                          np.transpose(sf_fem.basis()[0])])
>>> data
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.53452248,  0.53452248,  0.53452248],
       [-0.49487166, -0.98974332,  1.48461498],
       [ 1.42857143, -1.14285714, -0.28571429]])
>>> # filter the test data
>>> data_filtered = sf_fem.filter(data)
>>> data_filtered
array([[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00],
       [  1.00000000e+00,   1.00000000e+00,   1.00000000e+00],
       [  5.34522484e-01,   5.34522484e-01,   5.34522484e-01],
       [ -4.94871659e-01,  -9.89743319e-01,   1.48461498e+00],
       [ -1.69271249e-16,  -2.75762028e-16,   3.10220481e-16]])
```

property specification

Get or set the specification of the filter.

The parameter *specification* has to be an integer or a vector. Setting the *specification* will simultaneously apply a plausibility check.

3.5 spharapy.datasets: Sample data sets

The *spharapy.datasets*: module includes utilities to provide sample datasets.

spharapy.datasets.**load_eeg_256_channel_study**()

Load sensor setup and measured EEG data

The data set consists of a triangulation of a 256 channel equidistant EEG cap and EEG data from previously performed experiment addressing the cortical activation related to somatosensory-evoked potentials (SEP). During the experiment the median nerve of the right forearm was stimulated by bipolar electrodes (stimulation rate: 3.7 Hz, interstimulus interval: 270 ms, stimulation strength: motor plus sensor threshold [MAB+99][CAC+08], constant current rectangular pulse wave impulses with a length of 50 mu s, number of stimulations: 6000). Data were sampled at 2048 Hz and software high-pass (24 dB/oct, cutoff-frequency 2 Hz) and notch (50 Hz and two harmonics) filtered. All trials were manually checked for artifacts, the remaining trials were averaged, see also S1 data set in [GEF+15].

Number of vertices	256
Number of triangles	480
SEP Data (EEG)	256 channels, 369 time samples
Time range	50 ms before to 130 ms after stimulation
Sampling frequency	2048 Hz

Parameters

None

Returns

triangulation and EEG data: dictionary Dictionary-like object containing the triangulation of a simple triangular mesh. The attributes are: 'vertlist', the list of vertices, 'trilist', the list of triangles, 'labellist' the list of labels of the EEG channels, 'eegdata', an array containing the EEG data.

spharapy.datasets.**load_minimal_triangular_mesh**()

Returns the triangulation of a single triangle

The data set consists of a list of three vertices at the unit vectors of vector space :math:`mathbb{R}^3`and a list of a single triangle.

Number of vertices	3
Number of triangles	1

Parameters

None

Returns

triangulation [dictionary] Dictionary-like object containing the triangulation of a single triangle. The attributes are: 'vertlist', the list of vertices, 'trilist', the list of triangles.

spharapy.datasets.**load_simple_triangular_mesh**()

Returns the triangulation of a simple triangular mesh

The data set consists of a triangulation of an unit hemisphere.

Number of vertices	131
Number of triangles	232

Parameters

None

Returns

triangulation [dictionary] Dictionary-like object containing the triangulation of a simple triangular mesh. The attributes are: 'vertlist', the list of vertices, 'trilist', the list of triangles.

CHAPTER

FOUR

TUTORIALS AND INTRODUCTORY EXAMPLES

4.1 Determination of the SPHARA basis functions for an EEG sensor setup

Section contents

This tutorial introduces the steps necessary to determine a generalized spatial Fourier basis for an *EEG* sensor setup using SpharaPy. The special properties of the different discretization approaches of the Laplace-Beltrami operator will be discussed.

4.1.1 Introduction

A Fourier basis is a solution to Laplace's eigenvalue problem

$$L\vec{x} = \lambda\vec{x}\,, \qquad (1)$$

with the discrete *Laplace-Beltrami operator* in matrix notation $L \in \mathbb{R}^{M\times N}$, the eigenvectors $\vec{x}$ containing the harmonic functions and the eigenvalues λ the natural frequencies.

By solving a Laplace eigenvalue problem, it is also possible to determine a basis for a spatial Fourier analysis. Often in practical applications, a measured quantity to be subjected to a Fourier analysis is only known at spatially discrete sampling points (the sensor positions). An arbitrary arrangement of sample points on a surface in three-dimensional space can be described by means of a *triangular mesh*. In the case of an *EEG* system, the sample positions (the vertices of the triangular mesh) are the locations of the sensors arranged on the head surface. A SPHARA basis is a solution of a Laplace eigenvalue problem for the given triangle mesh, that can be obtained by discretizing a Laplace-Beltrami operator for the mesh and solving the Laplace eigenvalue problem in equation (1). The SpharaPy package provides three methods for the discretization of the Laplace-Beltrami operator; unit weighting of the edges and weighting with the inverse of the Euclidean distance of the edges, and a FEM approach. For more detailed information please refer to *SPHARA – The theoretical background in a nutshell* and *Eigensystems of Discrete Laplace-Beltrami Operators*.

At the beginning we import three modules of the SpharaPy package as well as several other packages and single functions of packages.

```
# Code source: Uwe Graichen
# License: BSD 3 clause
```

(continues on next page)

(continued from previous page)

```
# import modules from spharapy package
import spharapy.trimesh as tm
import spharapy.spharabasis as sb
import spharapy.datasets as sd

# import additional modules used in this tutorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

4.1.2 Specification of the spatial configuration of the EEG sensors

Import information about EEG sensor setup of the sample data set

In this tutorial we will determine a SPHARA basis for a 256 channel *EEG* system with equidistant layout. The data set is one of the example data sets contained in the SpharaPy toolbox, see *spharapy.datasets* and *spharapy.datasets.load_eeg_256_channel_study()*.

```
# loading the 256 channel EEG dataset from spharapy sample datasets
mesh_in = sd.load_eeg_256_channel_study()
```

The dataset includes lists of vertices, triangles, and sensor labels, as well as *EEG* data from previously performed experiment addressing the cortical activation related to somatosensory-evoked potentials (SEP).

```
print(mesh_in.keys())
```

Out:

```
dict_keys(['vertlist', 'trilist', 'labellist', 'eegdata'])
```

The triangulation of the *EEG* sensor setup consists of 256 vertices and 480 triangles.

```
vertlist = np.array(mesh_in['vertlist'])
trilist = np.array(mesh_in['trilist'])
print('vertices = ', vertlist.shape)
print('triangles = ', trilist.shape)
```

Out:

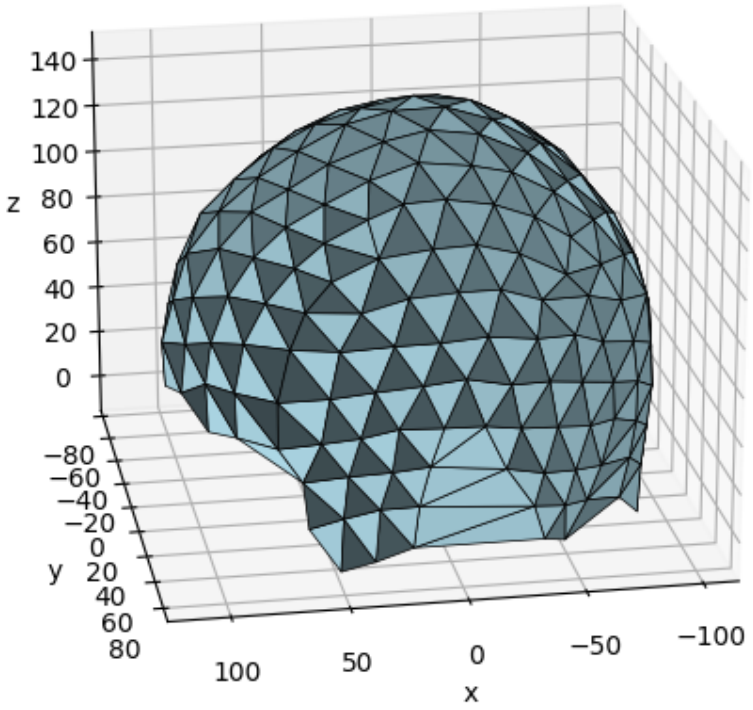
```
vertices =  (256, 3)
triangles =  (482, 3)
```

```
fig = plt.figure()
fig.subplots_adjust(left=0.02, right=0.98, top=0.98, bottom=0.02)
ax = fig.gca(projection='3d')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('The triangulated EEG sensor setup')
```

(continues on next page)

(continued from previous page)

```
ax.view_init(elev=20., azim=80.)
ax.set_aspect('auto')
ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1], vertlist[:, 2],
                triangles=trilist, color='lightblue', edgecolor='black',
                linewidth=0.5, shade=True)
plt.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/spharapy/checkouts/latest/
↪examples/plot_02_sphara_basis_eeg.py:106: MatplotlibDeprecationWarning:␣
↪Calling gca() with keyword arguments was deprecated in Matplotlib 3.4.␣
↪Starting two minor releases later, gca() will take no keyword arguments.␣
↪The gca() function should only be used to get the current axes, or if no␣
↪axes exist, create new axes with default keyword arguments. To create a new␣
↪axes with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

Create a SpharaPy TriMesh instance

In the next step we create an instance of the class *spharapy.trimesh.TriMesh* from the list of vertices and triangles.

```
# create an instance of the TriMesh class
mesh_eeg = tm.TriMesh(trilist, vertlist)
```

The class *spharapy.trimesh.TriMesh* provides a number of methods to determine certain properties of the triangle mesh required to generate the SPHARA basis, listed below:

```
# print all implemented methods of the TriMesh class
print([func for func in dir(tm.TriMesh) if not func.startswith('__')])
```

Out:

```
['adjacent_tri', 'is_edge', 'laplacianmatrix', 'massmatrix', 'one_ring_
↪neighborhood', 'remove_vertices', 'stiffnessmatrix', 'trilist', 'vertlist',
↪'weightmatrix']
```

4.1.3 Determining SPHARA bases using different discretisation approaches

Computing the basis functions

In the final step of the tutorial we will calculate SPHARA bases for the given *EEG* sensor setup. For this we create three instances of the class *spharapy.spharabasis.SpharaBasis*. We use the three discretization approaches implemented in this class for the Laplace-Beltrami operator: unit weighting ('unit') and inverse Euclidean weigthing ('inv_euclidean') of the edges of the triangular mesh as well as the FEM discretization ('fem')

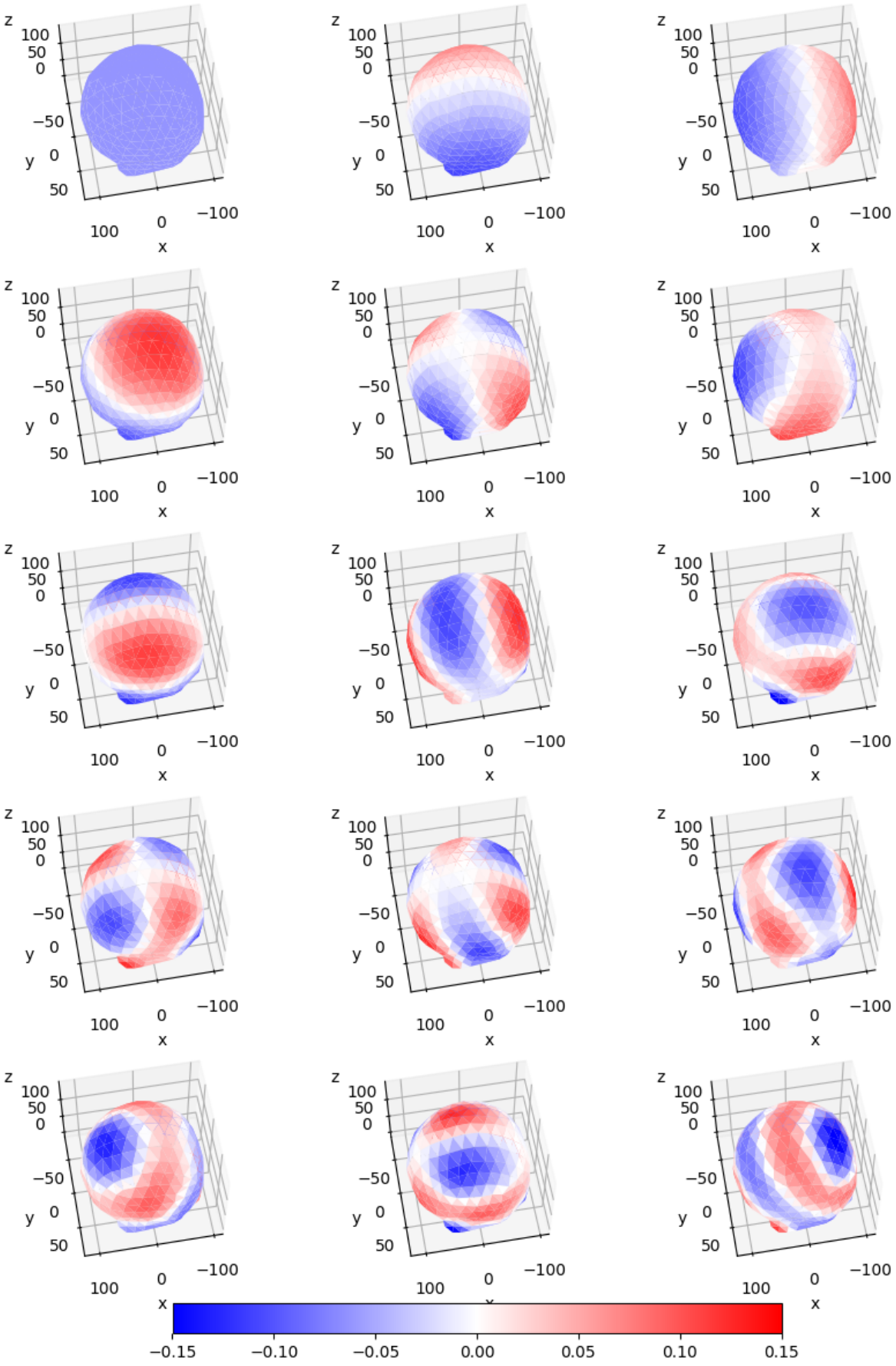
```
# 'unit' discretization
sphara_basis_unit = sb.SpharaBasis(mesh_eeg, 'unit')
basis_functions_unit, natural_frequencies_unit = sphara_basis_unit.basis()

# 'inv_euclidean' discretization
sphara_basis_ie = sb.SpharaBasis(mesh_eeg, 'inv_euclidean')
basis_functions_ie, natural_frequencies_ie = sphara_basis_ie.basis()

# 'fem' discretization
sphara_basis_fem = sb.SpharaBasis(mesh_eeg, 'fem')
basis_functions_fem, natural_frequencies_fem = sphara_basis_fem.basis()
```

Visualization the basis functions

The first 15 spatially low-frequency SPHARA basis functions are shown below, starting with DC at the top left.

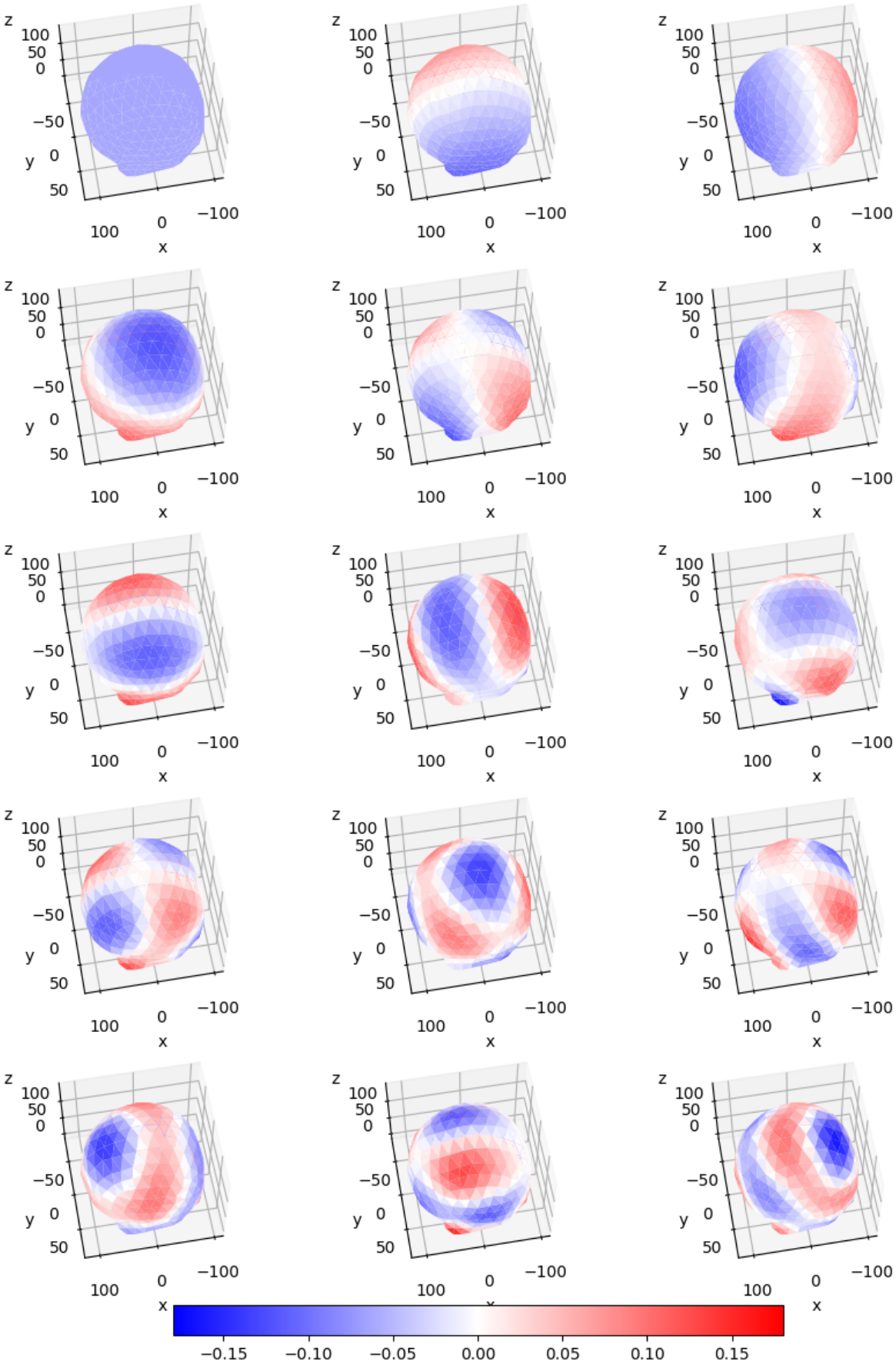
SPHARA basis using the discretization approache 'unit'

```
# sphinx_gallery_thumbnail_number = 2
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions_unit[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=60., azim=80.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
                                  edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.set_clim(-0.15, 0.15)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.5))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```


z
100
50
0
−50
y
0
50
100
0
−100
x
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

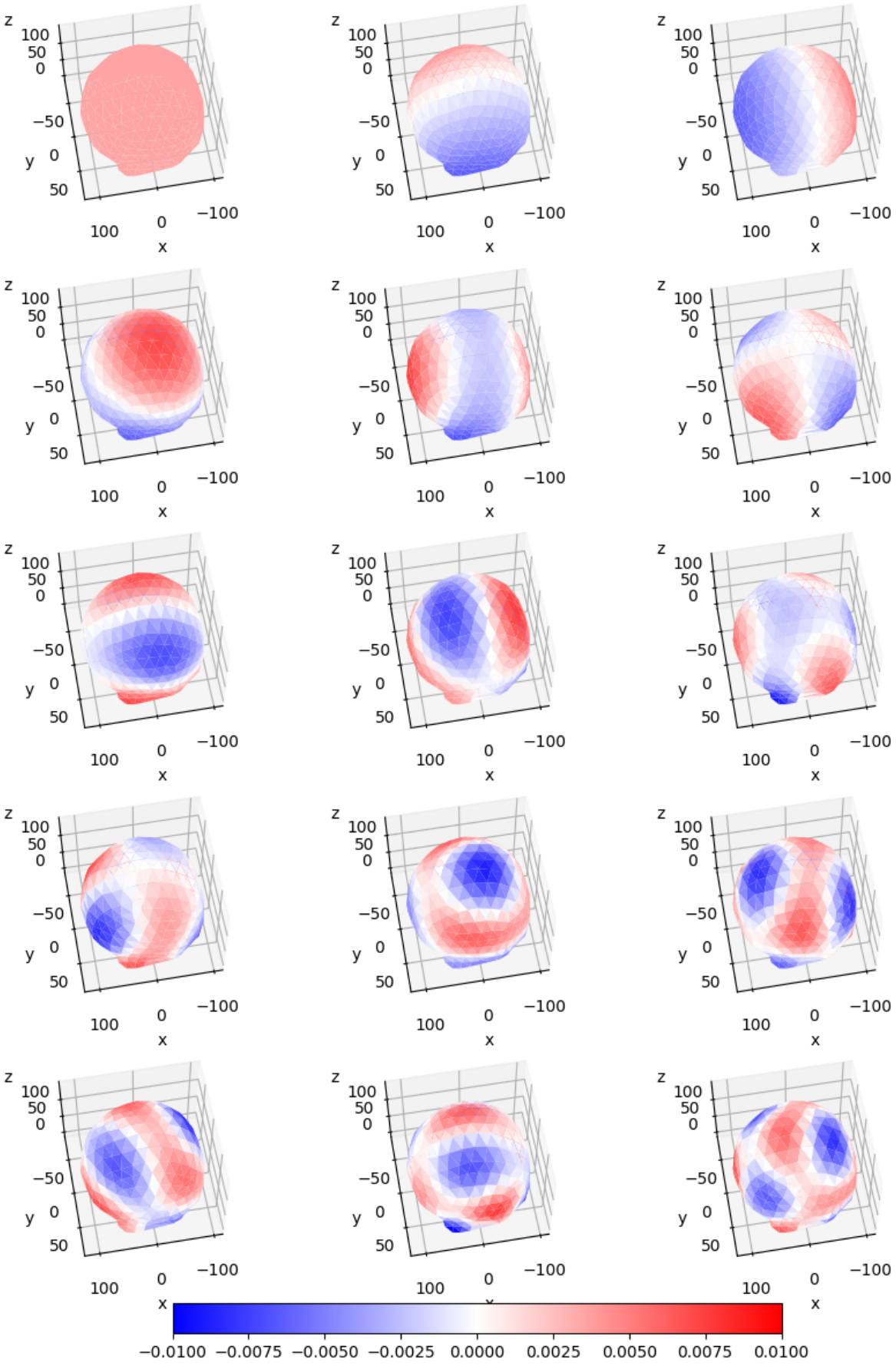
SPHARA basis using the discretization approache 'inv_euclidean'

```
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions_ie[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=60., azim=80.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
                                  edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.set_clim(-0.18, 0.18)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.5))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```


z
100
50
0
−50
y
0
50
100
0
x
−100
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

SPHARA basis using the discretization approache 'fem'

```
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions_fem[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=60., azim=80.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
                                  edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.set_clim(-0.01, 0.01)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.5))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```



Total running time of the script: (0 minutes 4.290 seconds)

4.2 Spatial SPHARA analysis of EEG data

Section contents

This tutorial shows exemplarily the spatial SPHARA analysis of 256-channel EEG data. The FEM discretization of the Laplace-Beltrami operator is employed to calculate the SPHARA basic functions that are used for the SPHARA decomposition.

4.2.1 Introduction

As explained in *SPHARA – The theoretical background in a nutshell* and in the tutorial *Determination of the SPHARA basis functions for an EEG sensor setup* a spatial Fourier basis for a arbitrary sensor setup can be determined as solution of the Laplace's eigenvalue problem dicretized for the considered sensor setup

$$L\vec{x} = \lambda\vec{x}\,,$$

with the discrete *Laplace-Beltrami operator* in matrix notation $L \in \mathbb{R}^{M\times N}$, the eigenvectors $\vec{x}$ containing the harmonic functions and the eigenvalues λ the natural frequencies.

The spatial Fourier basis determined in this way can be used for the spatial Fourier analysis of data recorded with the considered sensor setup.

For the anaylsis of discrete data defined on the vertices of the triangular mesh - the SPHARA transform - the inner product is used (transformation from spatial domain to spatial frequency domain). For an analysis using eigenvectors computed by the FEM approach, the inner product that assures the B-orthogonality needs to be applied.

For the reverse transformation, the discrete data are synthesized using the linear combination of the SPHARA coefficients and the corresponding SPHARA basis functions. More detailed information can be found in the section *Analysis and synthesis* and in [GEF+15].

At the beginning we import three modules of the SpharaPy package as well as several other packages and single functions of packages.

```
# Code source: Uwe Graichen
# License: BSD 3 clause

# import modules from spharapy package
import spharapy.trimesh as tm
import spharapy.spharatransform as st
import spharapy.datasets as sd

# import additional modules used in this tutorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

4.2.2 Import the spatial configuration of the EEG sensors and the SEP data

In this tutorial we will apply the SPHARA analysis to SEP data of a single subject recorded with a 256 channel EEG system with equidistant layout. The data set is one of the example data sets contained in the SpharaPy toolbox.

```
# loading the 256 channel EEG dataset from spharapy sample datasets
mesh_in = sd.load_eeg_256_channel_study()
```

The dataset includes lists of vertices, triangles, and sensor labels, as well as EEG data from previously performed experiment addressing the cortical activation related to somatosensory-evoked potentials (SEP).

```
print(mesh_in.keys())
```

Out:

```
dict_keys(['vertlist', 'trilist', 'labellist', 'eegdata'])
```

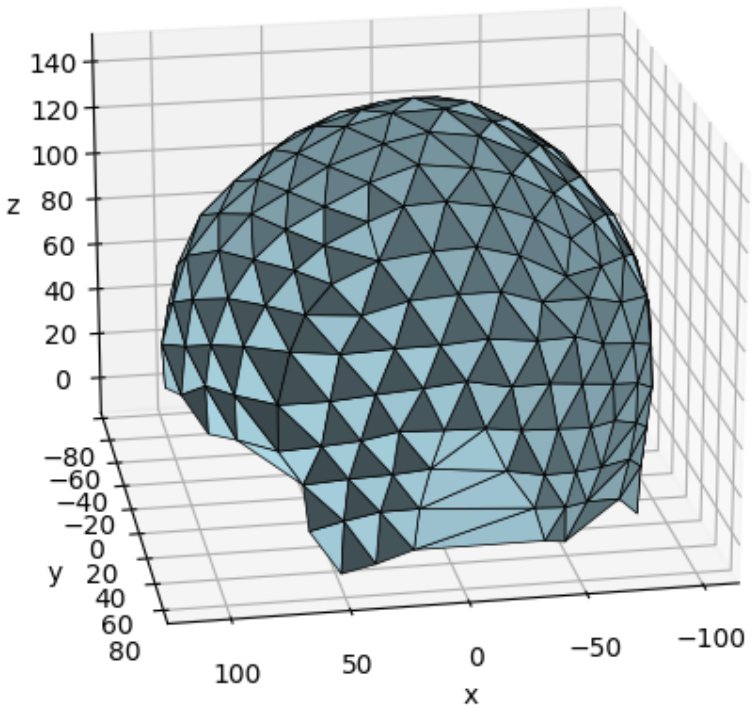
The triangulation of the EEG sensor setup consists of 256 vertices and 480 triangles. The EEG data consists of 256 channels and 369 time samples, 50 ms before to 130 ms after stimulation. The sampling frequency is 2048 Hz.

```
vertlist = np.array(mesh_in['vertlist'])
trilist = np.array(mesh_in['trilist'])
eegdata = np.array(mesh_in['eegdata'])
print('vertices = ', vertlist.shape)
print('triangles = ', trilist.shape)
print('eegdata = ', eegdata.shape)
```

Out:

```
vertices =  (256, 3)
triangles =  (482, 3)
eegdata =  (256, 369)
```

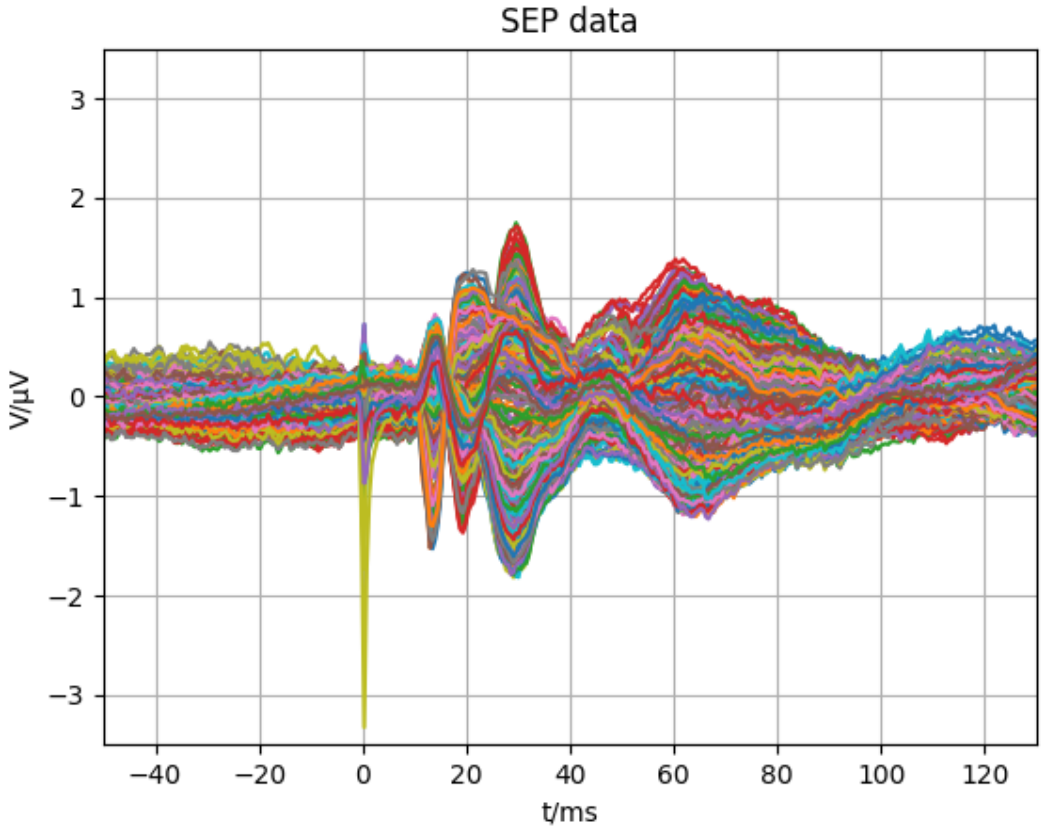
```
fig = plt.figure()
fig.subplots_adjust(left=0.02, right=0.98, top=0.98, bottom=0.02)
ax = fig.gca(projection='3d')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('The triangulated EEG sensor setup')
ax.view_init(elev=20., azim=80.)
ax.set_aspect('auto')
ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1], vertlist[:, 2],
                triangles=trilist, color='lightblue', edgecolor='black',
                linewidth=0.5, shade=True)
plt.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/spharapy/checkouts/latest/
↪examples/plot_03_sphara_analysis_eeg.py:105: MatplotlibDeprecationWarning:␣
↪Calling gca() with keyword arguments was deprecated in Matplotlib 3.4.␣
↪Starting two minor releases later, gca() will take no keyword arguments.␣
↪The gca() function should only be used to get the current axes, or if no␣
↪axes exist, create new axes with default keyword arguments. To create a new␣
↪axes with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

```
x = np.arange(-50, 130, 1/2.048)
figeeg = plt.figure()
axeeg = figeeg.gca()
axeeg.plot(x, eegdata[:, :].transpose())
axeeg.set_xlabel('t/ms')
axeeg.set_ylabel('V/µV')
axeeg.set_title('SEP data')
axeeg.set_ylim(-3.5, 3.5)
axeeg.set_xlim(-50, 130)
axeeg.grid(True)
plt.show()
```



4.2.3 Create a SpharaPy TriMesh instance

In the next step we create an instance of the class *spharapy.trimesh.TriMesh* from the list of vertices and triangles.

```
# create an instance of the TriMesh class
mesh_eeg = tm.TriMesh(trilist, vertlist)
```

4.2.4 SPHARA transform using FEM discretisation

Create a SpharaPy SpharaTransform instance

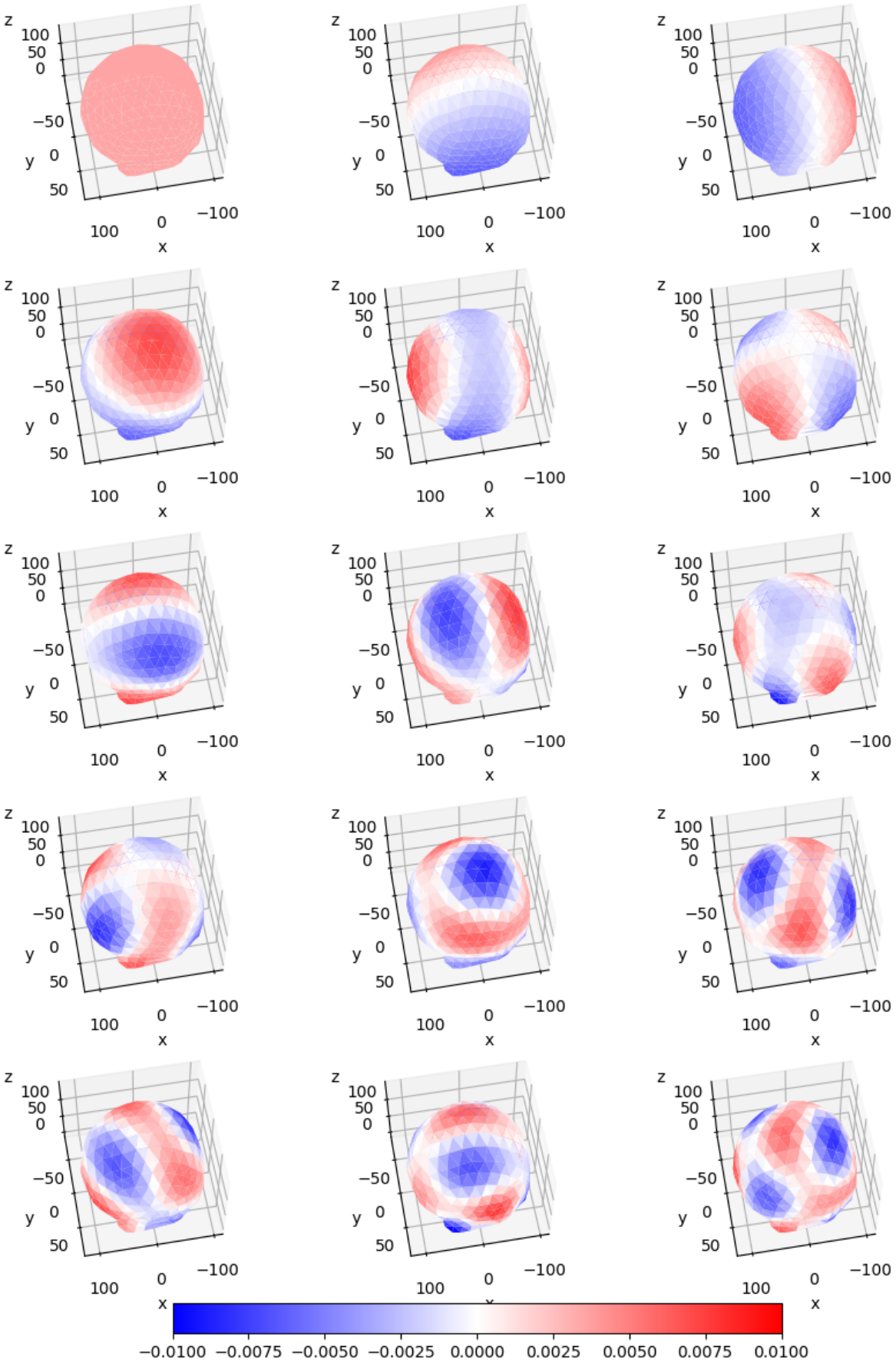
In the next step of the tutorial we determine an instance of the class SpharaTransform, which is used to execute the transformation. For the determination of the SPHARA basis we use a Laplace-Beltrami operator, which is discretized by the FEM approach.

```
sphara_transform_fem = st.SpharaTransform(mesh_eeg, 'fem')
basis_functions_fem, natural_frequencies_fem = sphara_transform_fem.basis()
```

Visualization the basis functions

The first 15 spatially low-frequency SPHARA basis functions of the basis used for the transform are shown below, starting with DC at the top left.

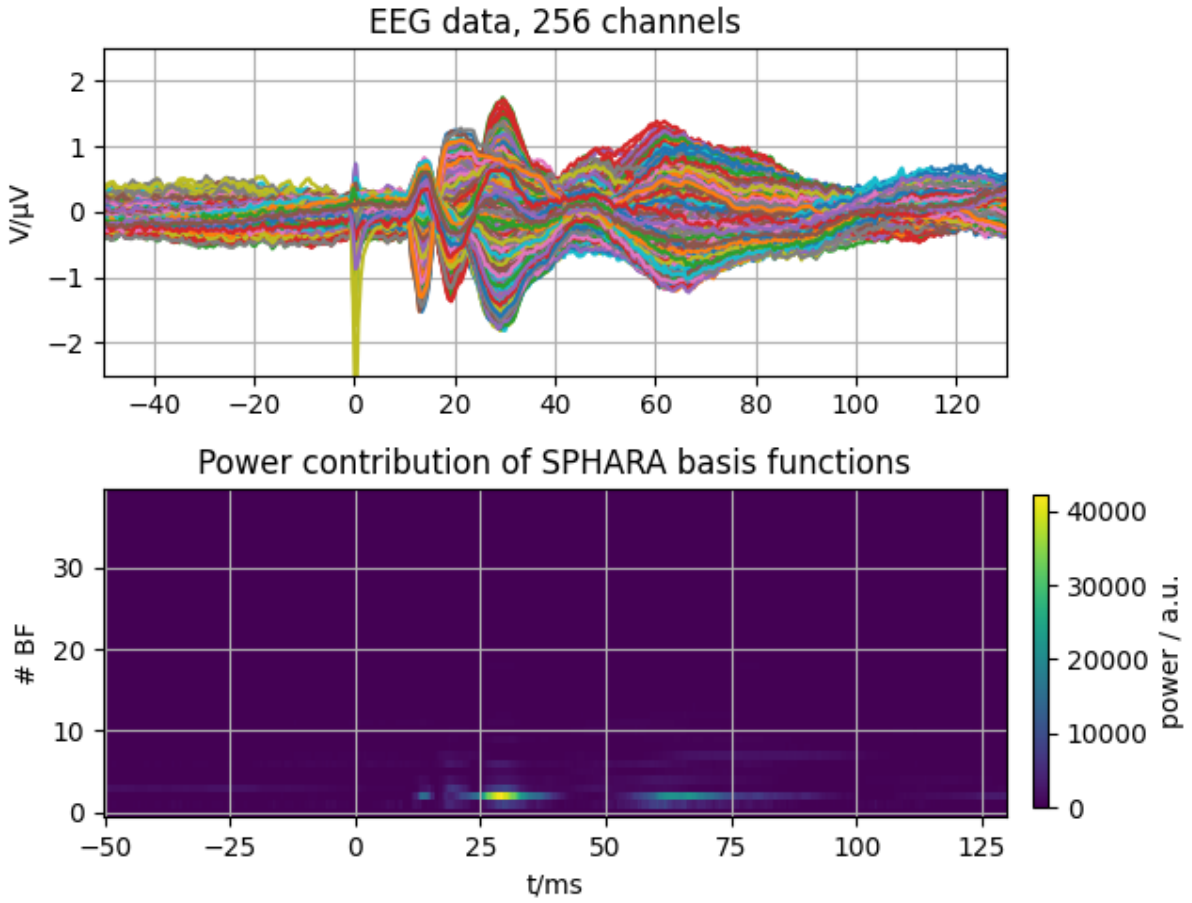
```
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions_fem[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=60., azim=80.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
                                  edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.autoscale()
    trisurfplot.set_clim(-0.01, 0.01)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.5))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```


z
100
50
0
−50
y
0
50
100
0
x
−100
−0.0100 −0.0075 −0.0050 −0.0025 0.0000 0.0025 0.0050 0.0075 0.0100

SPHARA transform of the EEG data

In the final step we perform the SPHARA transformation of the EEG data. As a result, a butterfly plot of all channels of the EEG is compared to the visualization of the power contributions of the first 40 SPHARA basis functions. Only the first 40 out of 256 basis functions are used for the visualization, since the power contribution of the higher basis functions is very low.

```
# perform the SPHARA transform
sphara_trans_eegdata = sphara_transform_fem.analysis(eegdata.transpose())

# 40 low-frequency basis functions are displayed
ysel = 40
figsteeg, (axsteeg1, axsteeg2) = plt.subplots(nrows=2)

y = np.arange(0, ysel)
x = np.arange(-50, 130, 1/2.048)

axsteeg1.plot(x, eegdata[:, :].transpose())
axsteeg1.set_ylabel('V/µV')
axsteeg1.set_title('EEG data, 256 channels')
axsteeg1.set_ylim(-2.5, 2.5)
axsteeg1.set_xlim(-50, 130)
axsteeg1.grid(True)

pcm = axsteeg2.pcolormesh(x, y,
                          np.square(np.abs(sphara_trans_eegdata.transpose()
                                           [0:ysel, :])))
axsteeg2.set_xlabel('t/ms')
axsteeg2.set_ylabel('# BF')
axsteeg2.set_title('Power contribution of SPHARA basis functions')
axsteeg2.grid(True)
figsteeg.colorbar(pcm, ax=[axsteeg1, axsteeg2], shrink=0.45,
                  anchor=(0.85, 0.0), label='power / a.u.')

plt.subplots_adjust(left=0.1, right=0.85, bottom=0.1, top=0.95, hspace=0.35)
plt.show()
# sphinx_gallery_thumbnail_number = 4
```



Total running time of the script: (0 minutes 2.408 seconds)

4.3 Spatial SPHARA filtering of EEG data

Section contents

In this tutorial we show how to use the SPHARA basis functions to design a spatial low pass filter for application to EEG data. The FEM discretization of the Laplace-Beltrami operator is used to calculate the SPHARA basic functions that are used for the SPHARA low pass filter. The applicability of the filter is shown using an EEG data set that is disturbed by white noise in different noise levels.

4.3.1 Introduction

The human head as a volume conductor exhibits spatial low-pass filter properties. For this reason, the potential distribution of the EEG on the scalp surface can be represented by a few low-frequency SPHARA basis functions, compare *Spatial SPHARA analysis of EEG data*. In contrast, single channel dropouts and spatially uncorrelated sensor noise exhibit an almost equally distributed spatial SPHARA spectrum. This fact can be exploited for the design of a spatial filter for the suppression of uncorrelated sensor noise.

At the beginning we import three modules of the SpharaPy package as well as several other packages and single functions from packages.

```
# Code source: Uwe Graichen
# License: BSD 3 clause

# import modules from spharapy package
import spharapy.trimesh as tm
import spharapy.spharafilter as sf
import spharapy.datasets as sd

# import additional modules used in this tutorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

4.3.2 Import the spatial configuration of the EEG sensors and the SEP data

In this tutorial we will apply a spatial SPHARA filter to SEP data of a single subject recorded with a 256 channel EEG system with equidistant layout. The data set is one of the example data sets contained in the SpharaPy toolbox.

```
# loading the 256 channel EEG dataset from spharapy sample datasets
mesh_in = sd.load_eeg_256_channel_study()
```

The dataset includes lists of vertices, triangles, and sensor labels, as well as EEG data from previously performed experiment addressing the cortical activation related to somatosensory-evoked potentials (SEP).

```
print(mesh_in.keys())
```

Out:

```
dict_keys(['vertlist', 'trilist', 'labellist', 'eegdata'])
```

The triangulation of the EEG sensor setup consists of 256 vertices and 480 triangles. The EEG data consists of 256 channels and 369 time samples, 50 ms before to 130 ms after stimulation. The sampling frequency is 2048 Hz.

```
vertlist = np.array(mesh_in['vertlist'])
trilist = np.array(mesh_in['trilist'])
eegdata = np.array(mesh_in['eegdata'])
print('vertices = ', vertlist.shape)
print('triangles = ', trilist.shape)
print('eegdata = ', eegdata.shape)
```

Out:

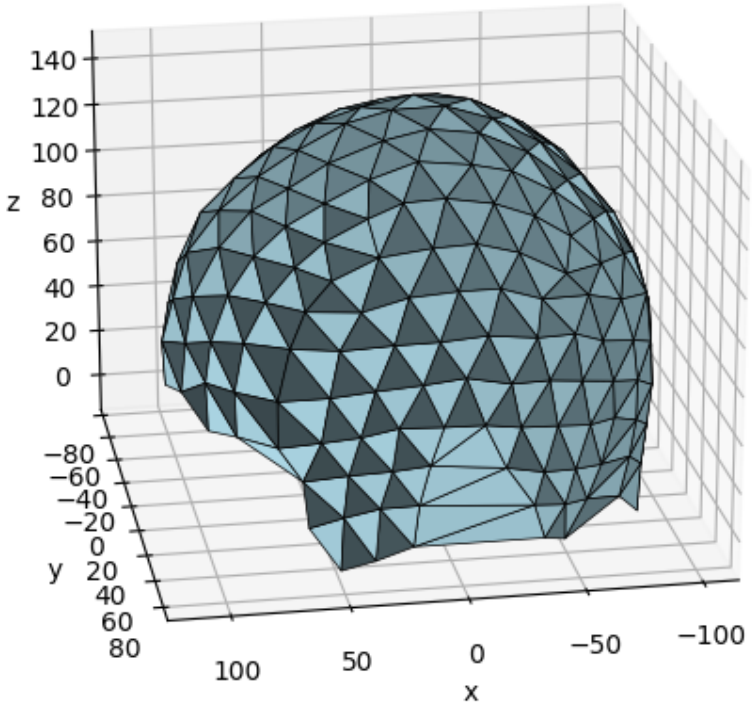
```
vertices =  (256, 3)
triangles =  (482, 3)
eegdata =  (256, 369)
```

```
fig = plt.figure()
fig.subplots_adjust(left=0.02, right=0.98, top=0.98, bottom=0.02)
```

(continues on next page)

(continued from previous page)

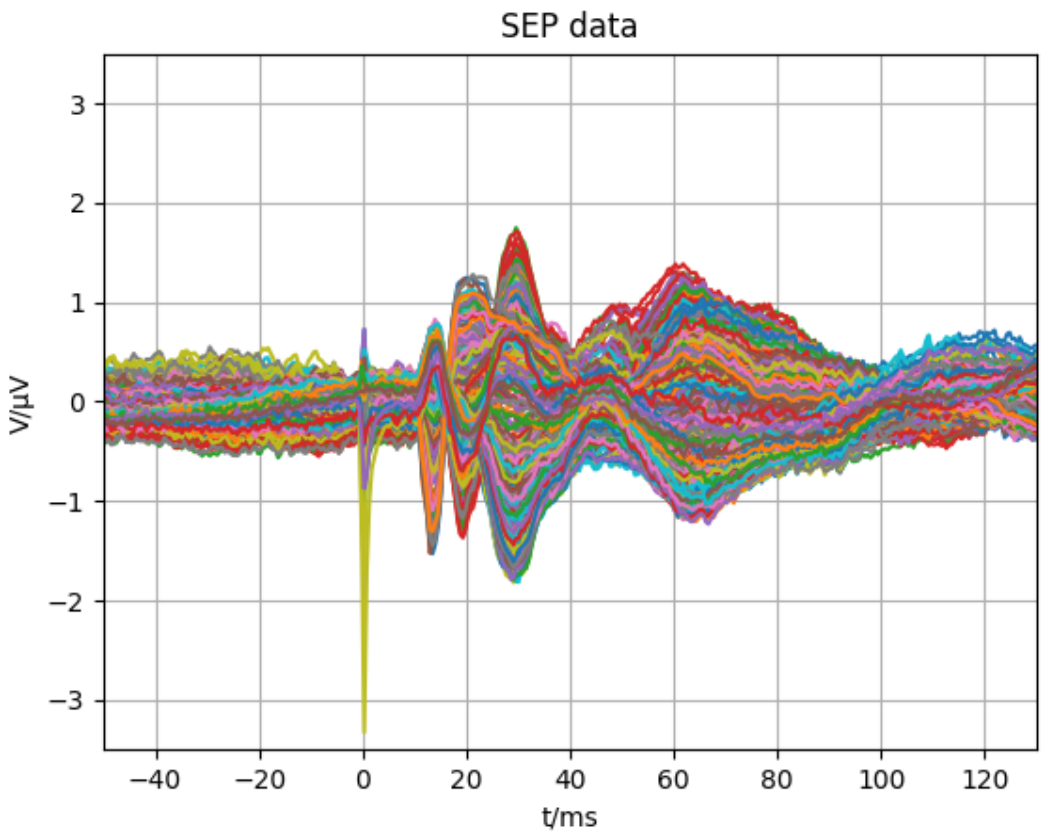
```
ax = fig.gca(projection='3d')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('The triangulated EEG sensor setup')
ax.view_init(elev=20., azim=80.)
ax.set_aspect('auto')
ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1], vertlist[:, 2],
                triangles=trilist, color='lightblue', edgecolor='black',
                linewidth=0.5, shade=True)
plt.show()
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/spharapy/checkouts/latest/
↪examples/plot_04_sphara_filter_eeg.py:88: MatplotlibDeprecationWarning:␣
↪Calling gca() with keyword arguments was deprecated in Matplotlib 3.4.␣
↪Starting two minor releases later, gca() will take no keyword arguments.␣
↪The gca() function should only be used to get the current axes, or if no␣
↪axes exist, create new axes with default keyword arguments. To create a new␣
↪axes with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

```
x = np.arange(-50, 130, 1/2.048)
figeeg = plt.figure()
axeeg = figeeg.gca()
axeeg.plot(x, eegdata[:, :].transpose())
axeeg.set_xlabel('t/ms')
axeeg.set_ylabel('V/µV')
axeeg.set_title('SEP data')
axeeg.set_ylim(-3.5, 3.5)
axeeg.set_xlim(-50, 130)
axeeg.grid(True)
plt.show()
```



Create a SpharaPy TriMesh instance

In the next step we create an instance of the class *spharapy.trimesh.TriMesh* from the list of vertices and triangles.

```
# create an instance of the TriMesh class
mesh_eeg = tm.TriMesh(trilist, vertlist)
```

4.3.3 SPHARA filter using FEM discretisation

Create a SpharaPy SpharaFilter instance

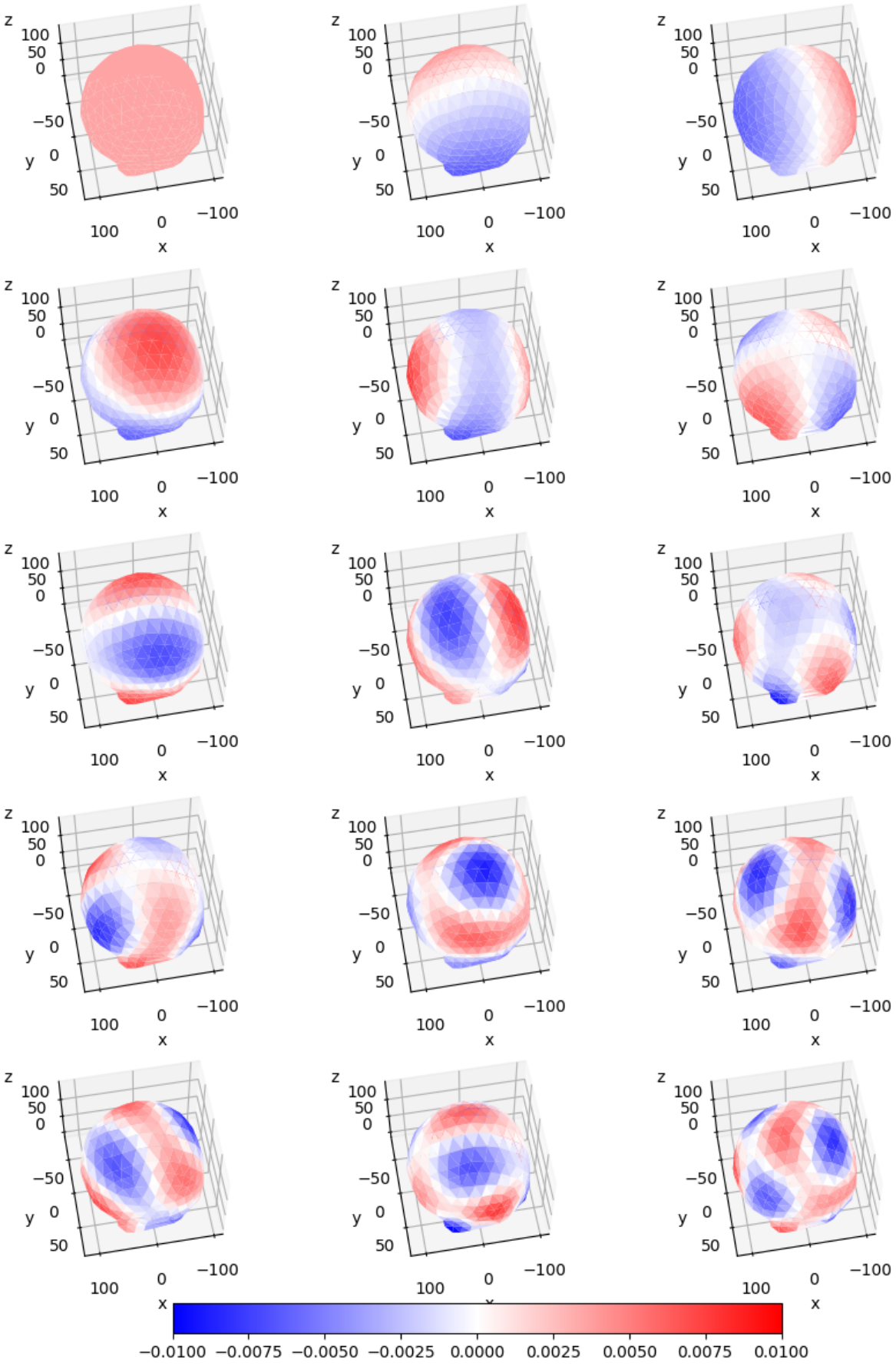
In the following step of the tutorial we determine an instance of the class SpharaFilter, which is used to execute the spatial filtering. For the determination of the SPHARA basis we use a Laplace-Beltrami operator, which is discretized by the FEM approach.

```
sphara_filter_fem = sf.SpharaFilter(mesh_eeg, mode='fem',
                                    specification=20)
basis_functions_fem, natural_frequencies_fem = sphara_filter_fem.basis()
```

Visualization the basis functions

The first 15 spatially low-frequency SPHARA basis functions are shown below, starting with DC at the top left.

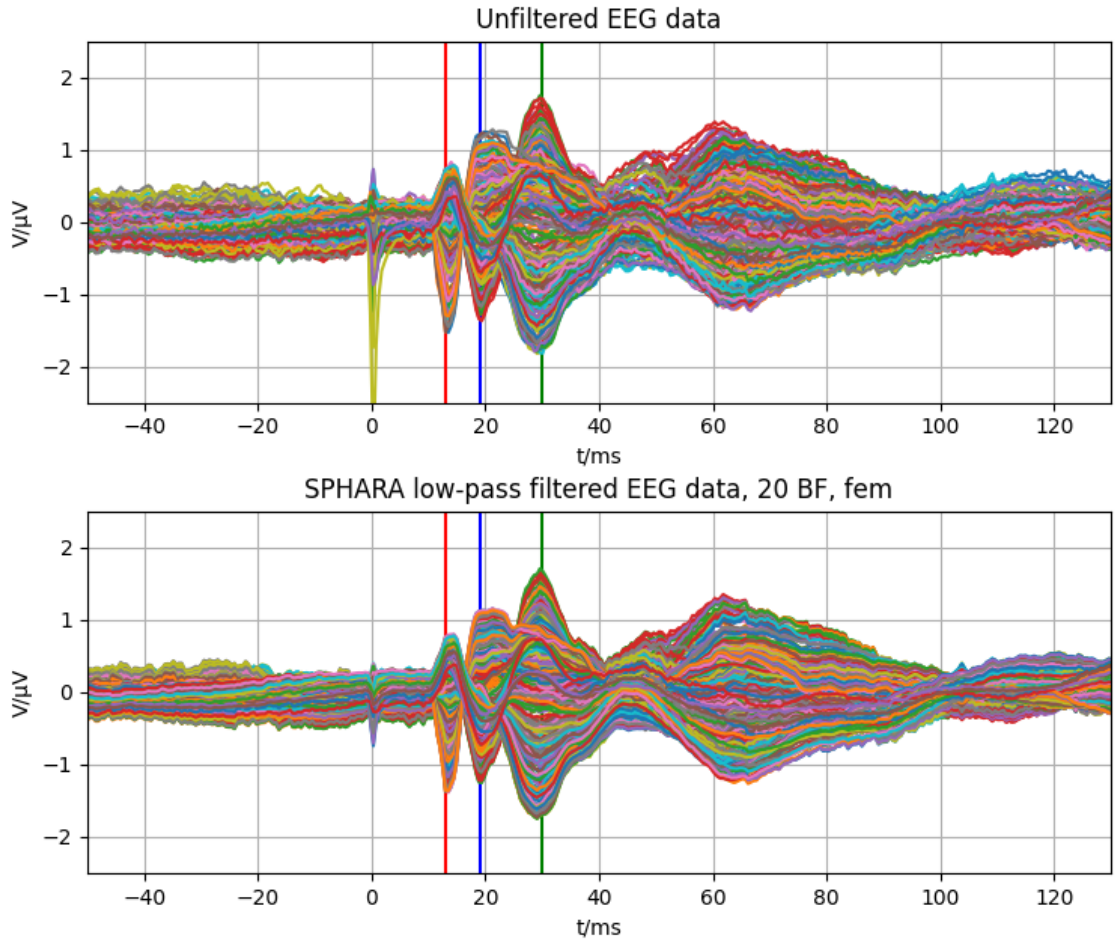
```
figsb1, axes1 = plt.subplots(nrows=5, ncols=3, figsize=(8, 12),
                             subplot_kw={'projection': '3d'})
for i in range(np.size(axes1)):
    colors = np.mean(basis_functions_fem[trilist, i + 0], axis=1)
    ax = axes1.flat[i]
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=60., azim=80.)
    ax.set_aspect('auto')
    trisurfplot = ax.plot_trisurf(vertlist[:, 0], vertlist[:, 1],
                                  vertlist[:, 2], triangles=trilist,
                                  cmap=plt.cm.bwr,
                                  edgecolor='white', linewidth=0.)
    trisurfplot.set_array(colors)
    trisurfplot.set_clim(-0.01, 0.01)

cbar = figsb1.colorbar(trisurfplot, ax=axes1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, -4.5))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.08, top=1.0)
plt.show()
```


z
100
50
0
−50
y
0
50
100
0
x
−100
−0.0100 −0.0075 −0.0050 −0.0025 0.0000 0.0025 0.0050 0.0075 0.0100

SPHARA filtering of the EEG data

In the next step we perform the SPHARA filtering of the EEG data. As a result, the butterfly plots of all channels of the EEG with and without filtering is compared. For the marked time samples also topo plots are provided.

```
# perform the SPHARA filtering
sphara_filt_eegdata = sphara_filter_fem.filter(eegdata.transpose()).
↪transpose()

figsteeg, (axsteeg1, axsteeg2) = plt.subplots(nrows=2, figsize=(8, 6.5))

axsteeg1.axvline(13, color='red')
axsteeg1.axvline(19, color='blue')
axsteeg1.axvline(30, color='green')
axsteeg1.plot(x, eegdata[:, :].transpose())
axsteeg1.set_title('Unfiltered EEG data')
axsteeg1.set_ylabel('V/µV')
axsteeg1.set_xlabel('t/ms')
axsteeg1.set_ylim(-2.5, 2.5)
axsteeg1.set_xlim(-50, 130)
axsteeg1.grid(True)

axsteeg2.axvline(13, color='red')
axsteeg2.axvline(19, color='blue')
axsteeg2.axvline(30, color='green')
axsteeg2.plot(x, sphara_filt_eegdata[:, :].transpose())
axsteeg2.set_title('SPHARA low-pass filtered EEG data, 20 BF, fem')
axsteeg2.set_ylabel('V/µV')
axsteeg2.set_xlabel('t/ms')
axsteeg2.set_ylim(-2.5, 2.5)
axsteeg2.set_xlim(-50, 130)
axsteeg2.grid(True)

plt.subplots_adjust(left=0.1, right=0.95, bottom=0.1, top=0.95, hspace=0.3)
plt.show()
```



Out:

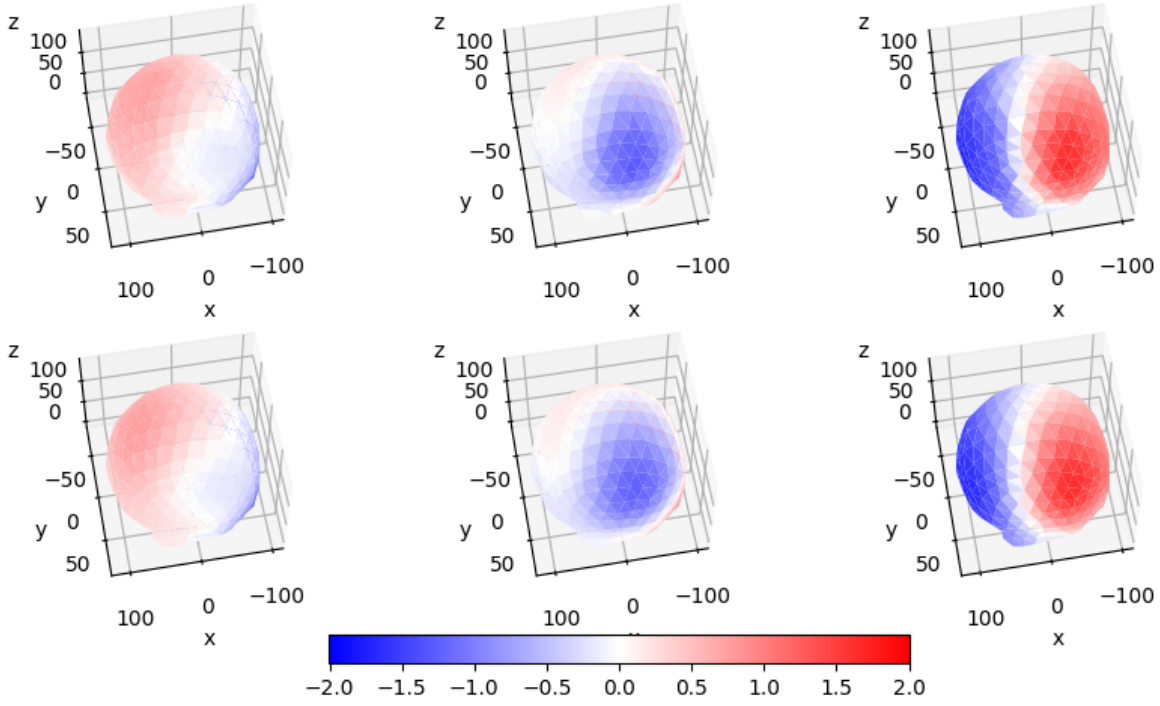
```
[[ 3.55257745e-03 -4.26785159e-03  3.71790460e-03 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 3.55257745e-03 -2.44107146e-03 -1.30848353e-05 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 3.55257745e-03 -1.68384132e-03 -3.73106354e-04 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 ...
 [ 3.55257745e-03 -7.00089372e-04 -1.07176673e-03 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 3.55257745e-03  8.98504311e-04 -8.07965233e-04 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 3.55257745e-03  3.00232831e-04 -1.57396181e-03 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]]
```

```
time_pts = [129, 141, 164]
figsf1, axessf1 = plt.subplots(nrows=2, ncols=3, figsize=(8, 5),
                               subplot_kw={'projection': '3d'})

for i in range(2):
    for j in range(3):
        if i == 0:
```

(continues on next page)

(continued from previous page)

```
            colorssf1 = np.mean(eegdata[trilist, time_pts[j]], axis=1)
        else:
            colorssf1 = np.mean(sphara_filt_eegdata[trilist, time_pts[j]],
                                axis=1)
        ax = axes1.flat[i]
        axessf1[i, j].set_xlabel('x')
        axessf1[i, j].set_ylabel('y')
        axessf1[i, j].set_zlabel('z')
        axessf1[i, j].view_init(elev=60., azim=80.)
        axessf1[i, j].set_aspect('auto')

        trisurfplot = axessf1[i, j].plot_trisurf(vertlist[:, 0],
                                                 vertlist[:, 1],
                                                 vertlist[:, 2],
                                                 triangles=trilist,
                                                 cmap=plt.cm.bwr,
                                                 edgecolor='white',
                                                 linewidth=0.)
        trisurfplot.set_array(colorssf1)
        trisurfplot.set_clim(-2., 2)

cbar = figsb1.colorbar(trisurfplot, ax=axessf1.ravel().tolist(), shrink=0.85,
                       orientation='horizontal', fraction=0.05, pad=0.05,
                       anchor=(0.5, 0))

plt.subplots_adjust(left=0.0, right=1.0, bottom=0.2, top=1.0)
plt.show()
```



Application of the the spatial SPHARA filter to data with artificial noise

In a final step the EEG data are disturbed by white noise with different noise levels (3dB, 0dB and -3dB). A spatial low-pass SPHARA filter with 20 basis functions is applied to these data. The results of the filtering are shown below.

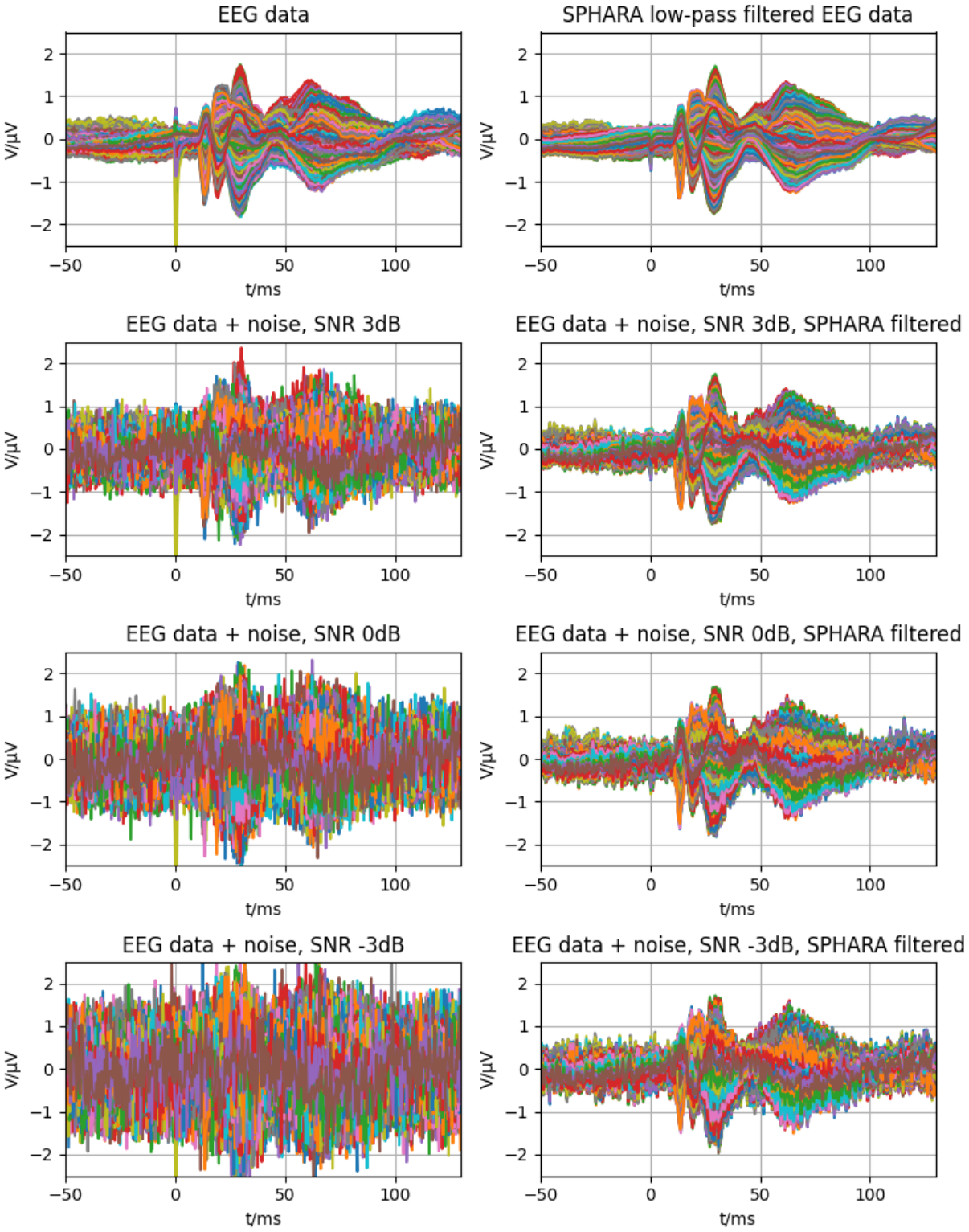
```
# vector with noise levels in dB
db_val_vec = [3, 0, -3]

# compute the power of the SEP data
power_sep = np.sum(np.square(np.absolute(eegdata))) / eegdata.size

# compute a vector with standard deviations of the noise in relation
# to signal power for the given noise levels
noise_sd_vec = list(map(lambda db_val:
                    np.sqrt(power_sep / (10 ** (db_val / 10))),
                        db_val_vec))

# add the noise to the EEG data
eegdata_noise = list(map(lambda noise_sd:
                         eegdata + np.random.normal(0, noise_sd, [256, 369]),
                         noise_sd_vec))

# filter the EEG data containing the artificial noise
eegdata_noise_filt = list(map(lambda eeg_noise:
                              (sphara_filter_fem.filter(eeg_noise.
↪transpose()).
                               transpose()),
                              eegdata_noise))
```

```
figfilt, axesfilt = plt.subplots(nrows=4, ncols=2, figsize=(8, 10.5))

axesfilt[0, 0].plot(x, eegdata[:, :].transpose())
axesfilt[0, 0].set_title('EEG data')
axesfilt[0, 0].set_ylabel('V/µV')
axesfilt[0, 0].set_xlabel('t/ms')
axesfilt[0, 0].set_ylim(-2.5, 2.5)
axesfilt[0, 0].set_xlim(-50, 130)
axesfilt[0, 0].grid(True)

axesfilt[0, 1].plot(x, sphara_filt_eegdata[:, :].transpose())
axesfilt[0, 1].set_title('SPHARA low-pass filtered EEG data')
axesfilt[0, 1].set_ylabel('V/µV')
axesfilt[0, 1].set_xlabel('t/ms')
axesfilt[0, 1].set_ylim(-2.5, 2.5)
axesfilt[0, 1].set_xlim(-50, 130)
axesfilt[0, 1].grid(True)

for i in range(3):
    axesfilt[i + 1, 0].plot(x, eegdata_noise[i].transpose())
```

(continues on next page)

(continued from previous page)

```
    axesfilt[i + 1, 0].set_title('EEG data + noise, SNR ' +
                                 str(db_val_vec[i]) + 'dB')
    axesfilt[i + 1, 0].set_ylabel('V/µV')
    axesfilt[i + 1, 0].set_xlabel('t/ms')
    axesfilt[i + 1, 0].set_ylim(-2.5, 2.5)
    axesfilt[i + 1, 0].set_xlim(-50, 130)
    axesfilt[i + 1, 0].grid(True)

    axesfilt[i + 1, 1].plot(x, eegdata_noise_filt[i].transpose())
    axesfilt[i + 1, 1].set_title('EEG data + noise, SNR ' +
                                 str(db_val_vec[i]) + 'dB, SPHARA filtered')
    axesfilt[i + 1, 1].set_ylabel('V/µV')
    axesfilt[i + 1, 1].set_xlabel('t/ms')
    axesfilt[i + 1, 1].set_ylim(-2.5, 2.5)
    axesfilt[i + 1, 1].set_xlim(-50, 130)
    axesfilt[i + 1, 1].grid(True)

plt.subplots_adjust(left=0.07, right=0.97, bottom=0.05, top=0.95, hspace=0.45)
plt.show()
# sphinx_gallery_thumbnail_number = 6
```



Total running time of the script: (0 minutes 6.177 seconds)

CHAPTER

FIVE

GLOSSARY OF COMMON TERMS

Boundary condition In partial differential equations, boundary conditions (BC) are constraints of the solution function u for a given domain D. Thus, the values of the function are specified on the boundary (in the topological sense) of the considered domain D. Neumann and Dirichlet boundary conditions are frequently used. The Python implementation of SPHARA uses the Neumann boundary condition in the solution of the Laplacian eigenvalue problem.

EEG EEG is an electrophysiological method for measuring the electrical activity of the brain by recording potentials on the surface of the head.

Finite Element Method The Finite Element Method (FEM) is a approach to solve (partial differential) equations, where continuous values are approximated as a set of values at discrete points. For the approximation nodal basis functions are used.

Laplace-Beltrami operator The generalized Laplace operator, that can applied on functions defined on surfaces in Euclidean space and, more generally, on Riemannian and pseudo-Riemannian manifolds. For triangulated manifolds, there are several methods to discretize the Laplace-Beltrami operator.

Triangular mesh A triangular mesh is a piecewise planar approximation of a smooth surface in $\mathbb{R}^3$ using triangles. The triangles of the mesh are connected by their common edges or corners. The sample points used for the approximation are the verices $\vec{c} \in V$ with $\vec{v}_i \in \mathbb{R}^3$. A triangle t is defined by three indices to the list of vertices. Thus, a triangular grid is represented by a list of vertices and a list of triangles.

BIBLIOGRAPHY

[Chu97] F. R. K. Chung. *Spectral Graph Theory*. Volume 92. American Mathematical Society, 1997. CBMS Regional Conference Series in Mathematics. doi:10.1090/cbms/092.

[CAC+08] G. Cruccu, M.J. Aminoff, G. Curio, J.M. Guerit, R. Kakigi, F. Mauguiere, P.M. Rossini, R.-D. Treede, and L. Garcia-Larrea. Recommendations for the clinical use of somatosensory-evoked potentials. *Clinical Neurophysiology*, 119(8):1705 – 1719, 2008. doi:10.1016/j.clinph.2008.03.016.

[DZMC07] R. Dyer, R. H. Zhang, T. Möller, and A. Clements. An investigation of the spectral robustness of mesh laplacians. Technical Report, Simon Fraser University, GrUVi Lab, Burnaby, Canada, 2007. URL: https://eprints.cs.univie.ac.at/4961.

[Fuj95] K. Fujiwara. Eigenvalues of Laplacians on a closed riemannian manifold and its nets. *Proceedings of the American Mathematical Society*, 123(8):2585–2594, 1995. doi:10.1090/S0002-9939-1995-1257106-5.

[GEF+15] U. Graichen, R. Eichardt, P. Fiedler, D. Strohmeier, F. Zanow, and J. Haueisen. SPHARA - a generalized spatial fourier analysis for multi-sensor systems with non-uniformly arranged sensors: application to EEG. *PLoS ONE*, 04 2015. doi:10.1371/journal.pone.0121741.

[MAB+99] F. Mauguiere, T. Allison, C. Babiloni, H. Buchner, A. A. Eisen, D.S. Goodin, S.J. Jones, R. Kakigi, S. Matsuoka, M.R. Nuwer, P.M. Rossini, and H. Shibasaki. Somatosensory evoked potentials. In G. Deuschl and A. Eisen, editors, *Recommendations for the Practice of Clinical Neurophysiology: Guidelines of the International Federation of Clinical Neurophysiology*, chapter 2.4, pages 79–90. Elsevier Science B. V., 1999. URL: http://www.scopus.com/inward/record.url?scp=0032621105&partnerID=8YFLogxK.

[MDSB03] M. Meyer, M. Desbrun, P. Schröder, and A. Barr. Discrete differential geometry operators for triangulated 2-manifolds. In H. C. Hege and K. Polthier, editors, *Visualization and Mathematics III*, pages 35–57. Springer, 2003. doi:10.1007/978-3-662-05105-4_2.

[PP93] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2:15–36, 1993. doi:10.1080/10586458.1993.10504266.

[Pol02] K. Polthier. Computational aspects of discrete minimal surfaces. In J. Hass, D. Hoffman, A. Jaffe, H. Rosenberg, R. Schoen, and M. Wolf, editors, *Proceedings of the Clay Summer School on Global Theory of Minimal Surfaces*. 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.6834&rank=1.

[RKH10] K.R. Rao, D.N. Kim, and J.J. Hwang. *Fast Fourier Transform: Algorithms and Applications*. Signals and communication technology. Springer, 2010. doi:10.1007/978-1-4020-6629-0.

[Tau95] G. Taubin. Signal processing approach to fair surface design. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 351–358. 1995. doi:10.1145/218380.218473.

[VL07] B. Vallet and B. Levy. Spectral geometry processing with manifold harmonics. Technical Report inria-00186931, Université Nancy, Institut National Polytechnique de Lorraine, 2007. doi:10.1111/j.1467-8659.2008.01122.x.

[WMKG07] M. Wardetzky, S. Mathur, F. Kälberer, and E. Grinspun. Discrete laplace operators: no free lunch. In A. Belyaev and M Garland, editors, *SGP07: Eurographics Symposium on Geometry Processing*, 33–37. Eurographics Association, 2007. doi:10.2312/SGP/SGP07/033-037.

[ZvKD07] H. Zhang, O. van Kaick, and R. Dyer. Spectral methods for mesh processing and analysis. In D. Schmalstieg and J. Bittner, editors, *STAR Proceedings of Eurographics*, volume 92, 1–22. 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.8135&rep=rep1&type=pdf.

[ZvKD10] H. Zhang, O. van Kaick, and R. Dyer. Spectral mesh processing. *Computer Graphics Forum*, 29(6):1865–1894, 2010. doi:10.1111/j.1467-8659.2010.01655.x.

PYTHON MODULE INDEX

S

INDEX

T

V

W