institut für elektronische musik und akustik

IEM Report 32/06:

3LD – Library for Loudspeaker Layout Design

A Matlab Library for Rendering and Evaluating Periphonic Loudspeaker Layouts.

Verfasser:

Florian Hollerweger

Kooperationspartner:

Center for Research in Electronic Art Technology,
University of California at Santa Barbara

1.3.2006

IEM - INSTITUT FÜR ELEKTRONISCHE MUSIK UND AKUSTIK
Vorstand: O.Univ.-Prof. Mag. DI Dr. Robert HÖLDRICH
A-8010 Graz, Inffeldgasse 10/3, Tel.:+43/(0)316/389 – 3170, FAX:+43/(0)316/389 – 3171
office@iem.at http://iem.at/

3LD – Library for Loudspeaker Layout Design

Abstract

The generation of loudspeaker layouts for periphonic sound spatialization systems is a non-trivial task, just like the underlying mathematical problem: the homogeneous distribution of a number of points on the surface of a sphere. According approaches have a long tradition in mathematics (platonic solids, geodesic spheres) as well as in physics (minimal energy configurations). However, neither of these fullfills all the requirements of periphonic loudspeaker layouts, such as arbitrary choice of the number of loudspeakers, possibility of psychoacoustical optimization (loudspeaker density as a function of the ear's spatial resolution) and the consideration of practical limitations (forbidden and explicitely demanded loudspeaker positions).

Through a hybrid approach of the strategies mentioned above, it is possible to overcome the limitations of each single method and create a universal tool for the design of three-dimensional loudspeaker layouts. The theoretical background for this has been described in the author's diploma thesis [1]. This project was concerned with the practical implementation of the developed theory and has resulted in the 3LD Library for Loudspeaker Layout Design, which includes features for the generation, visualization and evaluation of periphonic loudspeaker layouts.

1 Periphonic Loudspeaker Layouts

In [1], a detailed description of different criteria regarding the design of 3D loudspeaker layouts has been given, which can be briefly summarized by stating that the design of a periphonic (i.e. 3D) loudspeaker layout has to take into account

- The applied soundfield reconstruction algorithms, e.g. VBAP, Ambisonics, etc.
- The homogeneity of soundfield reconstruction
- The properties of human spatial hearing
- The loudspeaker distribution in the horizontal plane
- The architectural circumstances of a periphonic sound system

In this chapter, we will evaluate different approaches of generating periphonic loudspeaker layouts according to these criteria, starting with the *Platonic solids*, which are optimal configurations regarding the homogeneity of soundfield reproduction.

1.1 Platonic Solids

The five Platonic solids are the only convex polyhedra which are mathematically regular. Figure 1 shows them in the following left-to-right order:

- Tetrahedron
- Hexahedron (Cube)
- Octahedron
- Icosahedron
- Dodecahderon

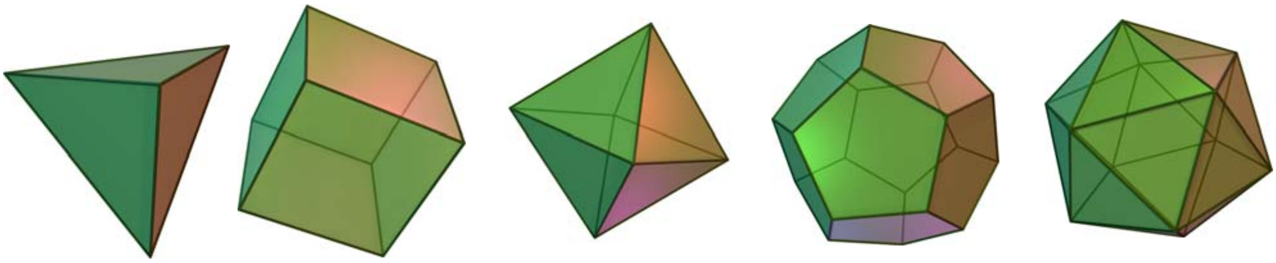

Figure 1: The five Platonic solids

In [2], a definition of regularity regarding Ambisonic soundfield reconstruction has been given, yielding that all Platonic solids are regular in the Ambisonic sense for first and the dodecahedron and icosahedron even for second order systems. Unfortunately, the Platonic solids are not suitable for large-scale periphonic audio reproduction systems, since they provide 20 vertices at the most. We will thus consider the possibility of their *geodesic extension* in the next chapter.

1.2 Geodesic Spheres

By tessellating the facets of a polyhedron and pushing the such created new vertices out to the radius of the original configuration, *geodesic spheres* can be built from the platonic solids or other polyhedra. An example of this iterative process is shown in figure 2. The method of geodesic spheres has been generalized in [1] towards maximum flexibility regarding the choice of the number of loudspeakers in a configuration, resulting in a set of tessellation rules. By applying these rules independently onto different facet shapes and different iterations of the process, we achieve significant freedom in the design of periphonic loudspeaker layouts.

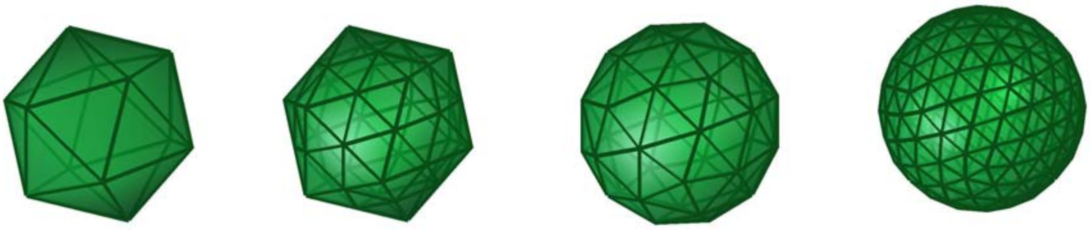

Figure 2: Construction process of a geodesic sphere

1.3 Minimal Energy Configurations

An approach from physics to the distribution of an arbitrary number of points on a sphere are so-called *minimal energy configurations*, which are generated by a random distribution of electrons on a spherical surface: Due to the repulsion forces among the electrons, they will arrange themselves in a natural equilibrum of minimal potential energy after some time. Note that the elctrons are only allowed to move on the surface of a sphere. Figure 3 shows some snapshots of this iterative process, which yields that the homogeneity of the configuration increases with the number of iterations. The obvious advantage of arbitrary numbers of electrons/loudspeakers has to be traded off for a lack of symmetry in the resulting layout.

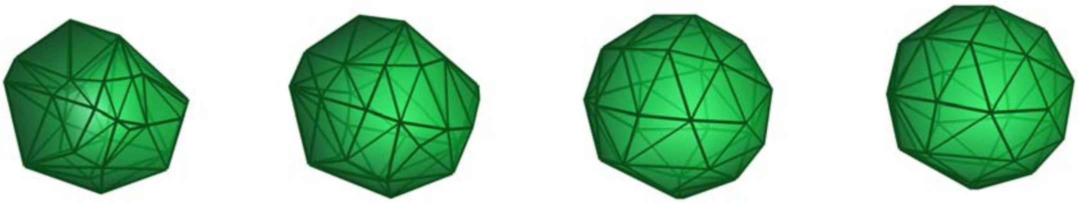

Figure 3: Electrons distributing themselves towards a minimal energy configuration

2 An Extended Loudspeaker Layout Design Strategy

In the author's master thesis [1], a new strategy for the design of periphonic loudspeaker layouts has been presented, which is a hybrid approach of the methods discussed so far. It bases on the separation of the design process into two stages, the first of which is dedicated to the construction of a homogeneous loudspeaker distribution, which is then psychoacoustically refined in the second stage: The spatial resolution of the human ear is best in the horizontal plane and for the front direction, whereas elevated and lateral sound sources can not be localized as well. By providing higher loudspeaker densities in areas of better auditory resolution, we can optimize a layout regarding the total number of loudspeakers. It has been suggested in [1] to use the charges of the electrons in a minimal energy algorithm for the implementation of spherical loudspeaker density functions: higher electron charges result in higher repulsion forces among the electrons, and thus in lower loudspeaker densities. A spherical loudspeaker density function can thus be derived as the inverse of a function representing direction-dependent electron charges (which do not exist in nature but are introduced here as a useful concept). However, a non-constant electron density also means that we cannot choose the initial electron distribution randomly any more, but rather have to use an initial configuration in which the electrons are already to some degree homogeneously distributed over the sphere. The Platonic solids or their geodesic extensions represent suitable intial layouts for this hybrid approach, which can be further extended in order to account for

- Non-spherical layout surfaces, i.e. a spherical radius function
- Gain and delay calibration due to differing loudspeaker distances
- Areas which do not allow for the mounting of loudspeakers
- Forced loudspeaker positions, i.e. 'locked' electrons

The combination of these considerations results in an *extended loudspeaker layout design strategy* shown in figure 4

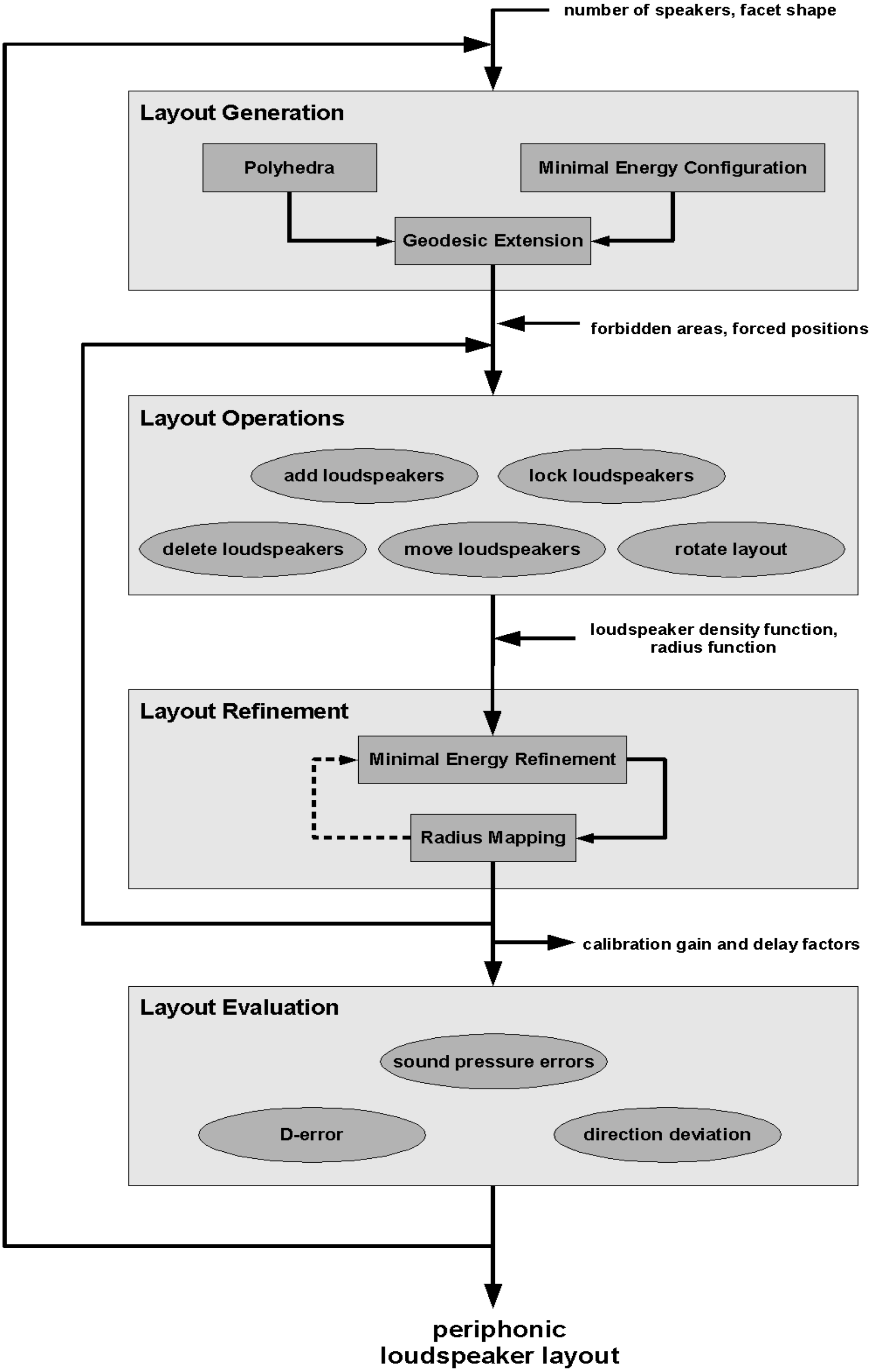

Figure 4: Extended Loudspeaker Layout Design Strategy

3 3LD – a Matlab Library for Periphonic Loudspeaker Layout Design

In this chapter, we will present the functions included in the 3LD Library for Loudspeaker Layout Design, including descriptions of their functionality, their input and output arguments and example applications. These can also be retrieved by typing `"help <functionname>"` in the Matlab command line. At the beginning of each section, the presented function is denoted including all of its input and output arguments. Optional input arguments are denoted in brackets. Note that the cartesian and spherical coordinate system conventions used in all of the functions follow the convention which is also used by native Matlab functions like `sph2cart`.

3.1 Core Functions

Two core functions have been implemented as useful extensions to native Matlab functions, with a range of possible applications clearly exceeding the field of periphonic loudspeaker layout generation:

spharmonic

`Y = spharmonic(n,az,elev[,norm])` computes the spherical harmonic functions of degree n and `m*sig = -n,-n+1,..,-1,0,1,..,n-1,n` , where `m` is the order and `sig = ±1` the superscript of a spherical harmonic. Note that the terms 'degree' and 'order' are sometimes used the other way around – for example, n denotes the *order* of an Ambisonic system. The `functions` are evaluated for each element of `az` and `elev`. n must be a scalar integer, and `az` and `elev` must be arrays of identical size containing azimuth and elevation in radians. `norm` is an optional argument, specifying different normalizations of the Legendre polynomials, which are used in the calculation of the spherical harmonics. Legal strings are `'unnorm'`, `'sch'` or `'norm'. The` default value is `'unnorm'`. `spharmonic` calls Matlab's `legendre` with this argument – more information on the normalization options can be found there.

The returned array Y has one more dimension than `az` and `elev`. Each element `Y(m*sig+n+1,i,j,k,...)` contains the spherical harmonic function of degree `n`, order `m`, and superscript `sig`, evaluated at `az(i,j,k,...)`, `elev(i,j,k,...)`.

For our spherical coordinate system, the spherical harmonic functions are given as:

```
                    Y(n,m,sig;az,elev) =


                               _
                              |  sin(m*az)   for sig=-1
=    P(n,m;sin(elev)) * <
                              |_ cos(m*az)   for sig=+1
```

For m=`0`, `spharmonic` only calculates the term for sig=+`1`, since the solution for `SIG=-1` is always zero. `P(n,m;sin(elev))` is the Legendre polynomial of degree `n` and order `m`, calculated at `sin(elev)` by the means of the Matlab function `legendre`.

Examples:

- `spharmonic(2 , 0.0:0.1:0.2, 0.3:0.1:0.5)` returns the matrix:

```
           .| az=0, el=0.3     az=0.1, el=0.4     az=0.2,el=0.5
-----------|--------------------------------------------------
m*sig = -2 |         0              0.5056              0.8997
m*sig = -1 |         0             -0.1074             -0.2508
m*sig =  0 |   -0.3690             -0.2725             -0.1552
m*sig =  1 |   -0.8470             -1.0707             -1.2370
m*sig =  2 |    2.7380              2.4943              2.1281
```

- `az = rand(2,4,5); elev = rand(2,4,5); n = 2;`
  `Y = spharmonic(n,az,elev);`
  so that `size(Y)` is 5x2x4x5 and
  `Y(:,1,2,3)` is the same as
  `spharmonic(n,az(1,2,3),elev(1,2,3))`.

See also: `legendre`.

ezspherical

`h = ezspherical(fct[,ngrid])` is an easy-to-use plotter for spherical functions. `fct` must be a function handle with two arguments, the first of which is interpreted as the azimuth, and the second one of which is interpreted as the elevation. `ngrid` specifies the resolution at which the plot is calculated: for both angles, a function value is calculated at every interval `2*pi/ngrid`. `ezspherical` actually plots the absolute value of the radius in each direction. Positive and negative function values can be distinguished by color (red for +1, blue for -1 in the default case). Note that `ezspherical` uses Matlab's `surf`, so you can set the properties of the returned graphic handle h as in `surf`.

Example:

- `ezspherical` can be used to plot spherical harmonic functions, created by `spharmonic` (3LD). Use `handlespharm` (3LD) to create an according function handle. For example, to plot the spherical harmonic function of degree 1, order 1, and superscript -1, try

```
fct = handlespharm('Y(1,-1)');
h = ezspherical(fct);
```

See also: `handlespharm` (3LD), `solospharm` (3LD), `spharmonic` (3LD), `surf`, `ezpolar`.

3.2 Loudspeaker Layout Generation and Modification

Note that all 3LD loudspeaker layout generation functions except `minenergyconf` return a faces/vertices structure which can be directly plotted using Matlab's `patch` or 3LD's `plot3LD`. Faces are always oriented counterclockwise as seen from the origin of a configuration.

platonicsolid

`p = platonicsolid(shape[,radius])` generates one of the five convex regular polyhedra, which are also refered to as Platonic solids. Those are the tetrahedron, the hexahedron (cube), the octahedron, the dodecahedron, and the icosahedron. `shape` is a string specifying which polyhedron to generate: Choices are `'tetrahedron'` (or `'tetra'`), `'hexahedron'` (or `'hex'` or `'cube'`), `'octahedron'` (or `'oct'`), `'dodecahedron'` (or `'dodec'`), `'icosahedron'` (or `'ico'`).

`radius` refers to the radius of the Platonic solid, i.e. the distance of its vertices to the center of the polyhedron, which is always located at the origin of the coordinate system. If not specified, `radius` defaults to 1.

`p` has fields `vertices` and `faces`. `vertices` is a V-by-3 matrix with rows representing the x,y,z coordinates of the V vertices, and `faces` is a F-by-S matrix with each row listing the row indices of the vertices forming one of the F faces of the polyhedron. S refers to the number of vertices in a face of the polyhedron: The tetrahedron, octahedron, and icosahedron have triangular faces (S=3), the cube has rectangular faces (S=4), and the dodecahedron has pentagonal faces (S=5). `p` can be plotted directly using Matlab's `patch` or 3LD's `plot3LD`.

<u>Example:</u>

- Plot an icosahedron:

```
p = platonicsolid('ico');
plot3LD(p);
```

<u>See also:</u> `geosphere` (3LD), `minenergyconf` (3LD), `sphere`, `ellipsoid`, cylinder, `patch`.

bucky2

`b = bucky2([radius])` generates the vertices/faces structure of a truncated icosahedron, also refered to as 'bucky ball'. This function uses Matlab's `bucky`, but replaces its adjacency matrix with a faces matrix. The output structure `b` has fields `vertices`, which is a V-by-3 array specifying the `x,y,z` coordinates as returned by `bucky`, and a F-by-6 `faces` array with rows containing the row indices of the vertices forming a facet. Note that the Bucky ball consists of hexagons and pentagons. For the rows in the faces matrix representing a pentagon, the last entry is `NaN`. The output structure `B` can be plotted directly using Matlab's `patch` or 3LD's `plot3LD`.

See also: `bucky`.

geosphere

`p = geosphere(shape[,freq,radius])` generates geodesic spheres from one of the five platonic solids using 3LD's `platonicsolid`, or from any other vertices/faces structure at its input. Geodesic spheres are constructed either by adding a vertex in the middle of each facet in a polyhedron and connecting it to every other vertex in the facet (figure 5, left and right pictures), or by subdividing the edges of each facet and connecting the new vertices in a way that depends on the facet's shape (figure 5, mid pictures). The first approach can be applied to arbitrarily shaped facets, while the latter can only be applied to triangles or rectangles. Here, the *frequency* `f` of the geodesic sphere determines into how many parts each edge is subdivided.

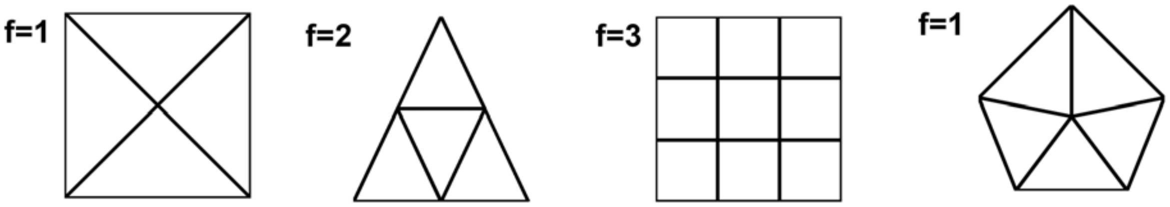

Figure 5: Different tessellations of various facet shapes [1]

`shape` can be a string specifying a platonic solid from which to build a geodesic sphere, or an existing polyhedron. In the first case, choices are `'tetrahedron'` (or `'tetra'`), `'hexahedron'` or `'hex'` or `'cube'`, `'octahedron'` or `'oct'`, `'dodecahedron'` or `'dodec'`, `'icosahedron' or 'ico'`. In the latter case, shape has to be a structure with fields `vertices` and `faces`. `vertices` is a V-by-3 matrix with rows representing the x,y,z coordinates of the V vertices, and `faces` is a F-by-S matrix with each row listing the row indices of the vertices forming one of the F faces of the polyhedron. S refers to the number of vertices in a face of the polyhedron.

`freq` is a matrix which determines the frequencies applied in the tessellation and the number of iterations. The j-th column refers to the j-th iteration. The i-th row refers to the polygons with j minus 2 vertices, i.e. triangles for the first row, rectangles for the second, pentagons for the third, and so on. For example, the element (4,3) of `freq` specifies the frequency applied to any hexagon in our polyhedron in the third iteration. Possible values for `freq` are:

- `freq` = 0 -> faces are not tessellated
- `freq` = 1 -> faces are midpoint-triangulated
- `freq > 1` -> triangular faces are triangulated, and rectangular faces are rectangulated at the frequency `freq`.

Note that since any faces with more than 4 vertices can only be midpoint-triangulated, elements >1 in the rows i>3 will be clipped to 1. If `freq` has less rows than there are different facet shapes in the polyhedron, the missing rows will be filled up with the entries of the last available row. For example, `freq`=[0; 3] means that all triangles will not be modified, all rectangles will be rectangulated at frequency 3, and all faces with more than 4 vertices will be midpoint-triangulated, since that's our only option. If not specified, `freq` defaults to 2, i.e. triangles are triangulated at frequency 2, rectangles rectangulated at frequency 2, and pentagons, etc. are midpoint-triangulated.

`radius` refers to the radius of the geodesic sphere, i.e. the distance of its vertices to the center of the sphere. It can be either a scalar or a handle to a function with two arguments, the first of which is interpreted as azimuth, and the second one as elevation. In the first case, all vertices are set to the same radius specified by the scalar. In the second case, each vertex is set to the radius specified by the radius function for its respective direction. Note that in both cases, the radii of existing vertices will be overwritten if `shape` is an existing polyhedron. If `radius` is empty or not specified, the polyhedron will be tessellated, but the new vertices will not be pushed out to a sphere or elsewhere.

p is a structure with fields `vertices` and `faces` which provide the same properties as required for the `shape` input argument. It can be plotted directly using 3LD's `plot3LD` or Matlab's `patch`.

Examples:

- `p = geosphere('oct',2,1); plot3LD(p);`
- `b = bucky2; p = geosphere(b,1,1); plot3LD(p);`
- `p = geosphere('oct',[2 3]); plot3LD(p);`
- `radius = @(az,elev) abs(cos(az) .* cos(elev)) + 2;`
- `p = geosphere('oct',[2 2 2],radius); plot3LD(p);`

See also: `platonicsolid` (3LD), `minenergyconf` (3LD), `sphere`, `ellipsoid`, `cylinder, patch`

minenergyconf

`v = minenergyconf(e[,n,radius,repulsion,density,lock])` simulates the process of electrons distributing themselves over the surface of a sphere until they reach what is refered to as a 'minimal energy configuration', i.e. a natural equilibrum of minimal potential energy. In `minenergyconf`, this algorithm has been extended to allow for arbitrary surface shapes, a spherical electron density function, scalable repulsion forces, and 'locked' electrons.

e can be a scalar, defining the number of electrons in a new configuration. In this case, the initial positions of the e electrons will be randomly distributed. e can also be an F-by-3 array, representing the x,y,z coordinates of an existing configuration with F electrons, which is then used as the initial layout for further modification.

n represents the number of iterations applied. In each iteration, the repulsion forces among all possible electron pairs are calculated, and the electrons are moved to their according new positions. If not specified, n defaults to 1.

`radius` determines the radius of the configuration. If `radius` is a scalar, the electrons will be distributed on a sphere with that radius. However, `radius` can also represent a handle to a spherical function, in which case it represents a surface with a radius depending on azimuth (first argument) and elevation (second argument). Both angels should be specified in radians. If not specified, `radius` defaults to 1.

`repulsion` specifies the power of the repulsion forces among the electrons. If not specified, `repulsion` defaults to 2, which represents the natural case of repulsion forces between two electrons which are proportional to their inverse square distance. If e.g. `repulsion=1`, the forces will be proportional to their plain inverse distance.

`density` is a handle to a spherical function representing the electron density as a function of direction (azimuth = first argument, elevation = second argument), in exactly the same way as `RADIUS` defines the radius as a spherical function. Higher repulsion forces among the electrons will occur in areas of lower density and vice versa. The repulsion forces among two electrons are then proportional to the product of the inverse densities at their positions. If not specified, `density` defaults to a constant density of 1 over the entire surface.

`lock` is a vector with a length matching the number of electrons in the configuration. A non-zero entry means that the electron with the according row index in the output array V will be 'locked', i.e. it will exercise repulsion forces on the other electrons, but is immune to the forces exercised on itself and will thus remain in its initial position. If not specified, all electrons remain unlocked, i.e. `lock` is a null vector. If `lock` is undersized, the remaining electrons will be considered unlocked.

V is an e-by-3 matrix specifying the resulting x,y,z coordinates of the e electrons.

Example:

- A minimal energy configuration with 50 electrons and 20 ierations:
  `p.vertices = minenergyconf(50,20);`

See also: `platonicsolid` (3LD), `geosphere` (3LD).

rotate_xyz

`v = rotate_xyz(coord,axis,angle)` rotates the P-by-3 array `coord, which` represents the x,y,z coordinates of P points, around either `AXIS (`specified by strings `'x'`, `'y'`, or `'z'`) by an `angle` given in radians. Arbitrary rotation axes can be achieved by subsequent application of `rotate_xyz`.

Example:

- Rotate an octahedron 45 degrees around the x axis

```
p = platonicsolid('oct');
p.vertices = rotate_xyz(p.vertices,'x',pi/4);
```

See also: `platonicsolid` (3LD), `geosphere` (3LD), `bucky2` (3LD), `minenergyconf` (3LD).

map_to_surface

`s = map_to_surface(V[,radius])` maps the N vertices of a polyhedron - represented by the N-by-3 array `V` which contains their x,y,z coordinates - to a surface defined by a function handle `radius`, which represents the radius as a spherical function. The first argument of the function handle is interpreted as the azimuth, and the second as the elevation. `radius` can also be a scalar, in which case the vertices are mapped to a sphere of that radius. If `radius` is not specified, the vertices are mapped to the unit sphere of radius `1`. The N-by-3 output array represents the new x,y,z coordinates of the vertices. Only the radius of the vertices will be affected, whereas their direction is maintained.

Example:

- Create a geodesic sphere and map its vertices to a surface derived from a spherical harmonic function, using 3LD functions. Plot the original polyhedron, the radius function and the new polyhedron.

```
p = geosphere('ico',[2,2],1);
patch(p,'facecolor',[.1 .7 .3],'facealpha',0.8);
radius = handlespharm('abs(3*Y(7,-5))+1');
figure; ezspherical(radius,200);
p.vertices = map_to_surface(p.vertices,radius);
figure; patch(p,'facecolor',[.1 .7 .3],'facealpha',0.8);
```

See also: `handlespharm` (3LD), `ezspherical` (3LD), calibrate_`layout` (3LD).

3.3 Loudspeaker Driving Signal Generation

amb3d_encoder

`B = amb3d_encoder(M,src[,gain,identical,sort])` calculates the weightings of the spherical harmonic components of a 3D Ambisonic soundfield. The N3D encoding convention as specified in [1] is applied.

`M` is the Ambisonic order and determines the number of Ambisonic channels `N = (M+1)^2`, which is the number of rows in the field `gain` of the output structure `B`.

`src` specifies the directions of the virtual sound sources. It is an S-by-2 array representing azimuth and elevation in radians.

`gain` specifies the gains of the virtual sound sources. It can be a scalar, in which case it is applied to all sources, or an S-by-1 array, with different gain factors for each source. Additional values will be ignored, and missing values will be set to 1. If not specified, gain defaults to 1 for all sources.

`identical` specifies whether the spatialized sound sources are fed by the same audio signal. If `identical` is not zero, this is assumed to be the case, while independent sources are assumed if `identical` is `0`. This affects the dimensions of the output array `B.gain`. If not specified, `identical` defaults to `0`.

`sort` is a string which affects the order of the rows in the output array. For `'spharmonic'` (or `'sph'` or `'s'`), the order of rows matches the one of 3LD's `spharmonic`, while for `'ambisonic'` (or `'amb'` or `'a'`), the Ambisonic channels are sorted according to a convention used in [1]. If not specified, `sort` defaults to `'spharmonic'`.

The output structure `B` has two fields. `B.gain` is an N-by-S matrix if `identical=0` and an N-by-1 matrix if `identical=1` and represents the weightings of the N spherical harmonic components. In the latter case, all sound sources are fed by the same signal and their weightings are superponed. B.sort is equivalent to the `sort` input argument. It is added to the output structure for later decoding with 3LD's `amb3d_decoder`.

Example:

- Encode a front and a top source at third order
  ```
  source_position = [0 0;0 pi/2];
  source_gain = [1 0.5];
  B = amb3d_encoder(3,source_position,source_gain)
  ```

See also: `amb3d_decoder` (3LD), `amb3d_regularity` (3LD), `vbp` (3LD), `spharmonic` (3LD).

amb3d_decoder

`g = amb3d_decoder(B,spk[,M,method,flavor])` derives the loudspeaker gains for an array of `L` speakers from the Ambisonic channel gains of a 3D encoded soundfield.

`B`, which typically represents the output of 3LD's `amb3d_encoder`, has to be a structure with fields `gain`, and `sort`. `gain` is an `N`-by-`S` array, with rows representing the `N` Ambisonic channels of `S` sound sources represented in the columns. `sort` determines the order in which the Ambisonic channels appear. Check the documentation of `amb3d_encoder` for a description.

`spk` specifies the directions of the loudspeakers. It must be an `L`-by-2 array, representing azimuth and elevation in radians. Note that `amb3d_decoder` calls `amb3d_encoder` for re-encoding the loudspeaker layout. Thus, the N3D encoding convention [2] is applied in this process

`M` is the Ambisonic order at which the decoder operates. If not specified, it is derived from the encoded input material `B`. Otherwise, the order of the input material can only be overridden with smaller values.

`method` specifies the decoding method. Legal strings are `'projection'` (or `'proj'`) and `'pseudoinverse'` (or `'pinv'`).
`flavor` specifies the decoder flavor. Legal strings are `'basic'` and `'inphase'`.

`g` is an L-by-S matrix. Each column represents the loudspeaker gains required to reproduce one of the S independent virtual sound sources, which are represented in the rows. Note that the function does by no means normalize the output gains. It is your responsibility to avoid clipping.

<u>Example:</u>

- Decode the output of the example in the `AMB3D_ENCODER` documentation to an icosahedron loudspeaker layout at second Ambisonic order.

  ```
  p = platonicsolid('ico');
  [az elev] = cart3sph(p.vertices);
  g = amb3d_decoder(B,[az elev],2,'pinv','inphase')
  ```

<u>See also:</u> `amb3d_encoder` (3LD), `amb3d_regularity` (3LD), `vbp` (3LD), `spharmonic` (3LD).

amb3d_regularity

`[regularity, condnum, C] = amb3d_regularity(M,spk)` returns a string `regularity`, which specifies whether a 3D layout of `L` loudspeakers is `'regular'`, `'semiregular'` or `'irregular'` in the Ambisonic sense for a specific Ambisonic order `M`, as defined on page 176 in [1]. It also returns the condition number `condnum` of the re-encoding matrix `C`, which can be built from the loudspeaker position information and is also a regularity criterion according to [3]. The third output argument `C` is the matrix (C*C')/L, which is used to evaluate the regularity. The layout is regular if this matrix is the unity matrix, and semi-regular if it is a diagonal matrix. Note that slight variations have been allowed in order to account for numerical inaccuracies. `spk` is an L-by-2 matrix containing the azimuths (first column) and elevations (second column) of the `L` loudspeakers. Note that as usual, the N3D encoding convention [1] is applied in evaluating the regularity and condition number of the layout.

Example:

- A dodecahedron is regular for second order Ambisonic

```
p = platonicsolid('dodec');
[az elev] = cart3sph(p.vertices);
amb3d_regularity(2,[az elev])
```

See also: `amb3d_encoder` (3LD), `amb3d_decoder` (3LD).

vbp

`g = vbp(src,spk,group[,type,gain,identical])` calculates the gains of an array of L loudspeakers due to S vector-base panned virtual sound sources. 2D or 3D VBP can be applied.

`src` specifies the directions of the virtual sound sources. For 2D VBP, it is an S-by-1 array representing the azimuths, whereas for 3D VBAP VBAP, it is an S-by-2 array representing azimuths and elevations. All angles have to be specified in radians.

`spk` specifies the directions of the loudspeakers. It can be an L-by-1 (2D VBP) or an L-by-2 (3D VBP) array, and is interpreted in the same way as `SRC`.

`group` specifies pairs (2D VBP) or triples (3D VBP) of loudspeakers. and thus is an R-by-2 or R-by-3 matrix, where R is the number of pairs/ triples. The entries of `group` represent the row indices of the respective loudspeakers in `spk`.

`type` is a string that specifies whether vector base amplitude panning [4] or vector base intensity panning [5] is applied. For the first case, the string should be `'vbap'` or `'a'`. For the second case, `'vbip'` or `'i'` are legal. If not specified, this argument defaults to `'vbap'`.

`gain` specifies the gains of the virtual sound sources. It can be a scalar, in which case it is applied to all sources, or an S-by-1 array, if different gain factors for each source are to be applied. Additional values will be ignored, and missing values will be set to 1. If not specified, `gain` defaults to 1.

`identical` specifies whether the spatialized sound sources are fed by the same audio signal. If `identical` is not zero, this is assumed to be the case, while independent sources are assumed if `identical` is 0. This affects the dimensions of the output array `G`. If not specified, `identical` defaults to 0.

The output array g represents the gains of the loudspeakers. It is an L-by-1 matrix if the sound sources are fed by the same audio signal (i.e. `identical=1`) or an L-by-S matrix if not (i.e. `identical=0`). In the latter case, the columns of g represent the gain factors for the S independent sound sources. Note that the function does by no means normalize the output gain factors. It is your responsibility to avoid clipping.

Example:

- Reproduce a front source on an octahedron loudspeaker layout

```
source_position = [0 0];
p = platonicsolid('oct');
[az elev] = cart3sph(p.vertices);
g = vbp(source_position,[az elev],p.faces,'vbip')
```

See also: `amb3d_encoder` (3LD), `amb3d_decoder` (3LD), `amb3d_regularity` (3LD).

calibrate_layout

`[g,d] = calibrate_layout(sppk[,c])` calculates the calibration gains g and delays d for an array of loudspeaker with varying radii. `spk` is a L-by-3 array representing the x,y,z coordinates of the L loudspeakers in the array. The same cartesian coordinate system as in Matlab's `SPH2CART` is applied. c is the speed of sound, which defaults to 343.3 meters per second (air at 20°C and at sea level) if not specified.

g is a vector of length L, representing the gain factors of all loudspeakers in the array due to the 1/r law of sound pressure amplitude decay. The speaker with the greatest radius will be assigned a gain factor of 1, whereas the gains of the closer speakers will be attenuated to factors < 1.

`d` is a vector of length `L`, representing the delay factors of all loudspeakers in the array due to the finite speed of sound `C` in seconds. The speaker with the greatest radius will be assigned a gain factor of `0`.

Note that loudspeaker array calibration results in the virtual sound sources moving at a distance which is equal to the maximum radius of a loudspeaker in the array, i.e. the distance of the loudspeaker which the layout is normalized too.

Example:

- Create a platonic solid using a 3LD function and move one of its vertices to a greater radius. Calculate the calibration gains and delays for loudspeakers positioned at the vertices of the solid.

```
p = platonicsolid('tetra');
p.vertices(1,:) = [0.8, 0.8, 0.8];
p.vertices
[g,d] = calibrate_layout(p.vertices)
```

See also: `map_to_surface` (3LD).

3.4 Soundfield Rendering and Evaluation

soundfielder

```
[p,v,VV,uV] = soundfielder(src,freq,type,X,Y,Z,time
...[,gain,sum,dir,T])
```

calculates the complex sound pressure, velocity, velocity vector, and u velocity of soundfields in air, created by S monochromatic sound sources with different radiation characteristics. The positions of the sources and sinks of the field can be arbitrarily defined.

`src` an S-by-3 array containing the x,y,z position information of the S sound sources.

`freq` is a scalar or a vector with lenght S, either refering to an identical frequency of all monochromatic sound sources, or specifying the frequency of each source independently.

`type` is a string which determines the radiation characteristics of the sound sources. Possible choices are `'spherical'` or `'sph'` or `'s'`, and `'plane'` or `'pl'` or `'p'`. If `type` is a single string, all sound sources will be of that type. If can also be a cell array of S strings, specifying the type for each of the sources independently.

`X`, `Y`, and `Z` are arrays specifying the x, y, and coordinates of the sinks, i.e. the measuring points of the soundfield. Use the output of Matlab's `meshgrid` for these arrays, i.e. 3D arrays for cubic, 2D arrays for plane, or vectors for line sink definition. `soundfielder` uses the number of dimensions for gradient evaluation of the output vector fields, thus you have to follow the described scheme to get useful output for the three output arguments `v`, VV, and uV. In all cases, the arrays have to be of identical size, and an equal number of dimensions is used in the output arrays to represent them. The only exception is given when `X` and `Y` are vectors and `Z` is a scalar, in which case `soundfielder` does the meshgriding itself such that all combinations of x and y at constant height z are calculated, i.e. the soundfield on a horizontal plane. Accordingly, two dimensions are used in the output arrays to represent the sinks.

`time` is a vector of the points in time at which the soundfield is rendered, which have to be specified in seconds.

`gain` determines the gain factors of the S sound sources. It can either be a scalar, refering to identical gains for all sources at each point of time for which the field is rendered, or a vector of length S, representing time-constant gains for each source, or an S-by-N array, where the element (i,j) represents the gain of the i-th sound source at the j-th point in time. Note that `gain` can be complex, specifying the amplitude as well as the phase of a wavefront.

`sum` is a scalar determining whether the soundfields created by the S sources are superponed in the output arrays `P` and `V`, which is the case if `sum` is a non-negative scalar.

`dir` specifies the direction of the wavefronts. Non-negative numbers refer to the incoming wavefront, while negative numbers specify an outgoing wavefront. `dir` can either be a scalar or a vector of length S, depending on whether all S sound sources share the same direction or not.

`T` is the time-invariant and homogeneous temperature at which the soundwaves propagate. It is specified in Kelvin and defaults to 273.15 Kelvin (0°C) if not specified.

`p` is the complex sound pressure field. The first dimensions of `p` refer to the dimensions of `X`, `Y`, `Z` (unless those were specified as two vectors and a scalar -> see `X, Y, Z`), followed by another dimension representing the various points of time for which the field is rendered. If `sum~=0`, an additional dimension refers to the fields of the different sound sources. Thus, generally `size(p) = [size(X),lenght(time),size(src,1)]`. However, any singleton dimension will be removed, e.g. if you calculate the soundfields caused by three sources in a cubic sink grid at a single point of time, `size(p) = [size(X),3]`. If you additionally superpone the fields created by the sources, `size(p) = size(X)`.

`v` is the complex sound velocity field as discussed in [6]. As in `p`, its first dimensions refer to the dimensions of `X`, `Y`, `Z`. The first dimension after these represents the vector components of the gradient field and thus has as many elements as there are non-singleton dimensions in `X`, `Y`, `Z`, i.e. three if the sink data is cubic, two if it is plane, and one if it represents a line. The next dimension represents the elements in `time`, and if `sum~=0`, another dimension refers to the velocity fields due to the different sources of the field. Thus, generally `size(V) = [size(X),numNonSingletonSinkDims,length(TIME),size(SRC,1)]`, but as in `p`, all singleton dimensions are removed, so if you calculate the superponed field of S sources along a line of 10 points in space for 17 points in time, `size(V) = [10 17]`.

`VV` is the complex velocity vector field, as defined in [7]. It is an array with the same size and properties as `v`. Its real part is associated with the perception of direction in a soundfield, and its imaginary part is often refered to in the literature as 'phasiness'. As in `p`, all singleton dimensions will be removed.

`UV` is the complex u velocity as defined in [6]. It is an array with the same size and properties as `V`. Its real part is refered to in the literature as 'active velocity', and is associated with the perception of direction in a soundfield. Its imaginary part is refered to as 'reactive velocity' and does not relate to sound energy transport. As in `P`, all singleton dimensions will be removed.

See also: `pressure_errors` (3LD), `direction_deviation` (3LD), `gradient`, `meshgrid`.

direction_deviation

`dirdev = direction_deviation(refdir,synthdir)` computes errors among the directions of two vector fields. `refdir` and `synthdir` are two arrays of equal size, representing the vector fields which indicate the directions of a complex reference pressure field and a synthesized field. Direction indicators are for example the real part of the complex velocity or the real part of the u velocity as calculated by 3LD's `soundfielder`.

`dim` specifies the dimension which represents the `x`,y,and possibly `z` components of the vector field. This dimension will be missing in the output array `direction_deviation`: since the direction deviation is specified as a scalar in radians at each point of the field, the dimensions representing the vector field components becomes singleton and is removed by the function.

See also: `soundfielder` (3LD), `pressure_errors` (3LD).

pressure_errors

`[pe2,ae] = pressure_errors(ref,synth)` computes errors among two complex sound pressure fields `ref` and `synth`, where the first represents the original soundfield, and the latter represents the reconstructed soundfield. The size of the input arrays is arbitrary, but has to match. The error is calculated as a scalar field with the size of the two input fields.

`pe2` is the 'squared sound pressure error', i.e. the squared difference of the two fields. Note that it is complex. `ae` is the 'sound pressure amplitude error', which is the absolute value of the difference of the two fields and is real.

See also: `soundfielder` (3LD), `direction_deviation` (3LD).

3.5 Helper Functions

solospharm

`Y = solospharm(n,mTimesSig,az,elev[,norm])` computes the spherical harmonic functions of degree `n`, order `m`, and superscript `sig = ±1`. The functions are evaluated for each element of `az` and `elev`. `n` must be a scalar integer. `mTimesSig` must be a vector with each element i fullfilling the condition `abs(mTimesSig(i)) <= n.` `az` and `elev` must be arrays of identical size containing, the azimuth and elevation arguments in radians. `norm` is an optional argument, specifying different normalizations of the Legendre polynomials, which are included in the spherical harmonics. Legal terms are `'unnorm'`,`'sch'` or `'norm'`, and default is `'unnorm'`. `Y` returns the values of the spherical harmonic functions for each element in the `mTimesSig` vector and each pair `az`, `elev`. The first dimension of `Y` refers to the different spherical harmonic functions, whereas the other dimensions refer to those of the input arrays `az` and `elev`. If `mTimesSig` is a scalar, the first (singleton) dimension is removed, and `Y` has the same size as `az` and `elev`.

`solospharm` calls `spharmonic` (3LD), which calls `legendre` with the `norm` argument; more information on the normalization options can be found in `legendre`.

`spharmonic` always returns the spherical harmonic functions of all orders and both superscripts for the specified degree `n`, i.e. from `mTimesSig = -n:n`. To be able to access single harmonic function of a certain order, `solospharm` has been introduced. Note that it is inefficient, since `solospharm` simply throws away the functions returned by `spharmonic` which have not been requested. However, `spharmonic` uses the native Matlab function `legendre`, which does not return functions of a single order either.

See also: `spharmonic` (3LD), `legendre`.

handlespharm

`h = handlespharm(string)` returns a handle to a combination of spherical harmonic functions. `string` is a string defining this combination using terms `'Y(n,m*sig)'`, where `n` refers to the degree, `m` to the order, and `sig = ±1` to the superscript of the spherical harmonic function. Note that `abs(m*sig) <= n`. The output of `handlespharm` can be plotted directly with `ezspherical` (3LD). Note that `handlespharm` the spherical harmonics in `handlespharm` are always Schmidt-seminormalized.

`handlespharm` calls `solospharm` (3LD), which calls `spharmonic` (3LD), which calls Matlab's `legendre`. Refer to those functions for more information.

Examples:

- Plot the sum of the spherical harmonic of degree 3, order 2, and superscript +1, and the absolute value of the function of degree 7, order 1, and superscript -1:
  ```
  h = handlespharm('Y(3,2) + abs(Y(7,-1))');
  ezspherical(h);
  ```
- Butterfly demo
  ```
  h = handlespharm('(Y(4,3)*Y(5,5)) / abs(Y(1,0)+2)');
  ezspherical(h);
  ```

See also: `spharmonic` (3LD), `solospharm` (3LD), `ezspherical` (3LD), `legendre`.

cart3sph

`[az,elev,r] = cart3sph(xyz)` This function is identical to Matlab's `cart2sph`, but takes a single N-by-3 matrix as an input argument, rather than three separate arrays. The columns of `xyz` represent the `x`, `y`, and `z` components of N different points. The function returns the `az` (azimuth), `elev` (elevation), and `r` (radius) components as three vectors. This is convenient to evaluate the spherical coordinates of a vertices array as returned by `platonicsolid`, `geosphere`, or `bucky2` (all 3LD) without prior separation into x,y,z vectors.

<u>See also:</u> `sph3cart` (3LD).

sph3cart

`xyz = sph3cart(az,elev,r)`. This function is identical to Matlab's `sph2cart`, but returns a single N-by-3 matrix as an input argument, rather than three separate arrays. The columns of `xyz` represent the `x`, `y`, and `z` components of the N different points. The azimuth, elavtion and radius components have to be provided as three independent input arguments. This function is convenient for converting the spherical coordinates of a vertices array back to their cartesian representation after edits like radius mapping.

<u>See also:</u> `cart3sph` (3LD).

deg2rad

`rad = deg2rad(deg)`

Convert an arbitrary numeric input array `deg` from degrees to radians.

<u>See also:</u> `rad2deg` (3LD).

rad2deg

`deg = rad2deg(rad)`

Convert an arbitrary numeric input array `rad` from radians to degrees.

See also: `deg2rad` (3LD).

plot3LD

`h = plot3LD(thing[,lock])` is a straightforward-to-use plotter for vertices/faces structures representing loudspeaker layouts and spherical functions representing radius or loudspeaker density functions.

In the case of loudspeaker layouts, `thing` is a structure with fields 'vertices' and 'faces'. Additionally, you can specify the `lock` status of the speakers (used also in `minenergyconf`), which will be represented with the color of the loudspeaker index in the plot: IDs of unlocked speakers are black, whereas locked loudspeaker indices appear red. The plotting is done by Matlab's `patch` function. 3LD functions which return a structure that can be directly plotted with `plot3LD` are `platonicsolid`, `bucky2`, and `geosphere`.

For plotting spherical functions, `thing` is a function handle depending on two variables, the first of which is interpreted as the azimuth and the second one of which as the elevation. The plotting is done by 3LD's `ezspherical`. The output of 3LD's `handlespharm` can be plotted directly using `plot3LD`.

See also: `ezspherical` (3LD), `platonicsolid` (3LD), `bucky2` (3LD), `geosphere` (3LD), `patch`.

3.6 Demo Scripts

The following demo scripts are included in the 3LD distribution:

- demo_density
- demo_minenergy
- demo_geosphere

- demo_soundfielder

and can serve as useful starting points for exploring the library.

References

[1] Florian Hollerweger. *Periphonic Sound Spatialization in Multi-User Virtual Environments*. Master's thesis, University of Music and Dramatic Arts Graz, Austria, 2006

[2] Jérôme Daniel. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia*. PhD thesis, Université Paris 6, 2000.

[3] Alois Sontacchi. *Dreidimensionale Schallfeldreproduktion für Lautsprecher- und Kopfhöreranwendungen*. PhD thesis, Technische Universität Graz, Austria, 2003.

[4] Ville Pulkki. *Virtual Sound Source Positioning Using Vector Base Amplitude Panning*. Journal of the Audio Engineering Society, Vol.45, No.6, June 1997, pp. 456-466

[5] Jean-Marie Pernaux, Patrick Boussard, Jean-Marc Jot: Virtual. *Sound Source Positioning and Mixing in 5.1. Implementation on the Real-Time System Genesis*. Proceedings DAFX98, Barcelona

[6] M. A. Poletti. *A Unified Theory of Horizontal Holographic Sound Systems.* Journal of the Audio Engineering Society, Vol.48, No.12, December 2000, pp.1155-1182.

[7] J. Daniel. *Acoustic Properties and Perceptive Implications of Stereophonic Phenomena. Corrected version, 03/29/99.* AES 16th International Conference on Spatial Sound Reproduction, 1999.