|  |
| --- |
|  |
| %% COMPACT HIGHER-ORDER AMBISONIC LIBRARY |
|  |  |
|  | %% |
|  | %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% |
|  | % |
|  | % Archontis Politis, 2015 |
|  | % Department of Signal Processing and Acoustics, Aalto University, Finland |
|  | % archontis.politis@aalto.fi |
|  | % |
|  | %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% |
|  |  |
|  | %% |
|  | % This is a compact Matlab/Octave library implementing most common operations |
|  | % associated with higher-order ambisonics (HOA), which refer to a set of |
|  | % spatial audio techniques for capturing, manipulating and reproducing |
|  | % sound scenes, based on a spherical Fourier expansion of the sound field. |
|  | % |
|  | % The included functions implement HOA encoding of directional sounds, |
|  | % decoding using various decoding approaches, and rotation of HOA sound |
|  | % scenes. All operations are defined in terms of orthonormalized real |
|  | % Spherical Harmonics (N3D in ambisonic slang) and channel indexing |
|  | % according to $q = n^2+n+m+1$, where n is the order and m is the degree (ACN |
|  | % in ambisonic slang). However, functions are included to convert to and |
|  | % from N3D/ACN to some other established conventions (namely |
|  | % semi-normalized SHs (SN3D) and an alternative channel indexing, termed |
|  | % SID). |
|  | % |
|  | % Ambisonic decoding can be approached from various sides, more physically |
|  | % inspired or more perceptually inspired. Five approaches are implemented |
|  | % |
|  | % \* 1) Sampling or projection decoding (transpose) |
|  | % \* 2) Mode-matching decoding (pseudo-inverse) |
|  | % \* 3) Energy-preserving decoding [ref.1] |
|  | % \* 4) All-round ambisonic decoding [ref.2] |
|  | % \* 5) Constant-angular spread decoding [ref.3] |
|  | % |
|  | % Apart from the two first traditional approaches, the three last are more |
|  | % recent and more perceptually motivated. They are also more |
|  | % flexible and robust, in terms of loudspeaker layouts. |
|  | % |
|  | % Additionally, a function evaluating and visualizing the popular ambisonic |
|  | % performance measures, velocity and energy vectors, along with overall |
|  | % energy and amplitude preservation, is included. |
|  | % |
|  | % Max-rE weighting for the decoder [ref.4 & ref.2] can be optionally enabled. |
|  | % |
|  | % ALLRAD and CSAD decoders require computation of amplitude and energy |
|  | % panning gains, and large spherical uniform sampling schemes (t-Designs). |
|  | % Both of these can be found firs in the Matlab/Octave VBAP library in |
|  | % |
|  | % <https://github.com/polarch/Vector-Base-Amplitude-Panning> |
|  | % |
|  | % and the general Spherical harmonic transform library by the author found in |
|  | % |
|  | % <https://github.com/polarch/Spherical-Harmonic-Transform>, |
|  | % |
|  | % These two libraries should be added to the Matlab path before executing |
|  | % this script. |
|  | % |
|  | % Rotation, apart from the case of simple B-format, also depends on the |
|  | % larger spherical harmonic transform library, which contains many |
|  | % other operations that may be of interest to ambisonics, like directional |
|  | % smoothing (spherical convolution) and directional weighting/shaping |
|  | % (spherical multiplication). |
|  | % |
|  | % The library contains the following main functions: |
|  | % |
|  | % \* ambiDecoder: Compute a HOA decoding matrix for a specified order and |
|  | % a specified method, with or without max-rE weighting |
|  | % \* analyzeDecoder: Analyze amplitude, energy, velocity and energy vector |
|  | % magnitudes and directional errors and spread, for an |
|  | % ambisonic decoder, or from panning gains |
|  | % \* getRSH: returns values of real orthonormal spherical harmonics |
|  | % vectors of directions |
|  | % \* encodeHOA\_N3D: encode a number of source signals from various directions |
|  | % to arbitrary-order HOA signals (N3D, ACN) |
|  | % \* encodeBformat: encode a number of source signals from various directions |
|  | % to traditional B-format signals |
|  | % \* decodeHOA\_N3D: decode HOA signals to a loudspeaker setup, using a |
|  | % certain decoding matrix (frequency-dependent decoding |
|  | % possible, see below) |
|  | % \* decodeBformat: decode B-format signals to a loudspeaker setup, using a |
|  | % certain decoding matrix |
|  | % \* rotateHOA\_N3D: rotate sound scenes encoded or recorded in HOA signals, |
|  | % using a yaw-pitch-roll convention |
|  | % \* rotateBformat: rotate sound scenes encoded or recorded to B-format signals, |
|  | % using a yaw-pitch-roll convention |
|  | % \* allrad: implements the all-round ambisonic decoder |
|  | % \* csad: implements the constant-angular spread decoder |
|  | % \* getLayoutAmbisonicOrder: Compute the equivalent ambisonic order for |
|  | % regular and irregular speaker arrangements |
|  | % \* getMaxREweights: Compute the max-rE weights for a certain order, for |
|  | % decoding (or encoding) weighting |
|  | % \* getTheoreticalEVmag: Compute the theoretical energy vector magnitude |
|  | % of a plane-wave encoded to a certain order, |
|  | % with max-rE weighting |
|  | % \* plotSphericalGrid: Plots angular quantities in a 2D azimuth-elevation |
|  | % grid, with loudspeaker positions superimposed |
|  | % |
|  | % |
|  | % For any questions, comments, corrections, or general feedback, please |
|  | % contact archontis.politis@aalto.fi |
|  |  |
|  |  |
|  | %% ENERGY AND VELOCITY VECTOR ANALYSIS |
|  | % |
|  | % The velocity (or Makita) vector [ref.5] and the energy (or Gerzon) vector |
|  | % [ref.6], computed from the loudspeaker layout and the panning gains for a |
|  | % certain panning direction, are believed to indicate perceived directions |
|  | % along with localization blur and source spread. These two vectors |
|  | % are actually related to the more general acoustic intensity vector, and |
|  | % the diffuseness or reactivity of the soundfield at the listening spot, |
|  | % but they are simpler to compute (see [ref.7]). They have been fundamental |
|  | % in the design of ambisonic systems. Psychoacoustical validation of their |
|  | % relation to localization attributes has only recently been studied [ref.8]. |
|  | % |
|  | % Generally, loudspeaker gains have to preserve the energy of a sound coming |
|  | % from a certain direction, so that the loudspeaker setup does not affect |
|  | % its loudness. This energy preservation property can be given as: |
|  | % |
|  | % $$E(\theta,\phi) = \sum\_{l=1}^L g\_l^2(\theta,\phi) = \mathrm{constant}$$ |
|  | % |
|  | % while at low frequencies it is more appropriate to preserve the total |
|  | % amplitude, since a coherent summation model at the listener's |
|  | % ears is more appropriate (see [ref.9]) |
|  | % |
|  | % $$A(\theta,\phi) = \sum\_{l=1}^L g\_l(\theta,\phi) = \mathrm{constant}$. |
|  | % |
|  | % Here $\theta,\phi$ denote azimuth and elevation, and $g\_l$ are the |
|  | % loudspeaker gains for the specific direction. Velocity and energy vectors |
|  | % are then given by |
|  | % |
|  | % $$\vec{r}\_v(\theta,\phi) = \frac{\sum\_{l=1}^L g\_l(\theta,\phi) \vec{u}\_l}{A(\theta,\phi)}$$ |
|  | % |
|  | % and |
|  | % |
|  | % $$\vec{r}\_e(\theta,\phi) = \frac{\sum\_{l=1}^L g\_l^2(\theta,\phi) \vec{u}\_l}{E(\theta,\phi)}$. |
|  | % |
|  | % where $\vec{u}\_l$ are the unit vectors pointing to the speakers. Analysis |
|  | % of a panner or decoder in terms of energy/amplitude preservation and |
|  | % velocity/energy vectors can be done with analyzeDecoder(). |
|  | % |
|  | % In terms of ambisonic decoding, for a uniform arrangement of loudspeakers, |
|  | % the ambisonic order of the layout can be readily given by |
|  | % |
|  | % $$N = \lfloor \sqrt{L} - 1 \rfloor$. |
|  | % |
|  | % For irregular layouts, evaluating an effective order is more complicated, |
|  | % Zotter & Frank in [ref.2] propose an equivalent ambisonic order having to |
|  | % do with the average spread of the layout. The equivalent ambisonic order |
|  | % can be evaluated here by getLayoutAmbisonicOrder(). |
|  | % |
|  | % A relation of the source spread/localization blur, with repect to the |
|  | % magnitude of the energy vector is given by Daniel or Zotter in [ref.4 & 2] |
|  | % |
|  | % $$\gamma = 2\arccos(||\vec{r}\_e||)180/\pi$. |
|  | % |
|  | % while an alternative definition is given by Epain et al. in [ref.3] |
|  | % |
|  | % $$\gamma = 2\arccos(2||\vec{r}\_e||-1)180/\pi$. |
|  | % |
|  | % Finally, Frank in [ref.10] proposes a psychoacoustic curve between the |
|  | % relation of the enegry vector magnitude and perceived spread, from |
|  | % listening tests: |
|  | % |
|  | % $$\gamma = 186.4(1-||\vec{r}\_e||)+10.7$. |
|  | % |
|  | % In any case, the best energy vector magnitude, and so minimal spread, |
|  | % that can be achieved for a certain decoding order (due to a theoretical |
|  | % continuous ambisonic loudspeaker setup) is given by [ref.4 & 2] |
|  | % |
|  | % $$||\vec{r}\_e|| = \frac{2 \sum\_{n=0}^N (n+1)a\_n a\_{n+1}} {\sum\_{n=0}^N (2n+1)a^2\_n }$. |
|  | % |
|  | % where $a\_n$ are the per-order decoding weights, such as the max-rE ones. |
|  | % These metrics for a certain panner/decoder are displayed if the flag |
|  | % INFO\_ON on the analyzeDecoder() is set to true. |
|  | % |
|  | % A first example is shown for pure amplitude panning (and energy panning) |
|  | % functions, which are not ambisonic but they serve as basis for some of |
|  | % the more flexible ambisonic decoders included here. |
|  |  |
|  | % define a 3D layout with the shape of an icosahedron |
|  | [~, ls\_dirs\_rad] = platonicSolid('icosahedron'); |
|  | ls\_dirs = ls\_dirs\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs,1); |
|  |  |
|  | aziRes = 5; |
|  | polarRes = 5; |
|  | ang\_res = [aziRes polarRes]; |
|  | g\_vbap = getGainTable(ls\_dirs, ang\_res, 0, 'vbap'); |
|  | [g\_vbip, g\_dirs] = getGainTable(ls\_dirs, ang\_res, 0, 'vbip'); |
|  |  |
|  | % reshape the 1D array of gains (times ls\_num) returned by getGainTable() |
|  | % to a 2D azimuth-elevation matrix (times ls\_num) |
|  | G\_vbap = permute(reshape(g\_vbap, [(360/aziRes+1) (180/polarRes+1) ls\_num]), [2 1 3]); |
|  | G\_vbip = permute(reshape(g\_vbip, [(360/aziRes+1) (180/polarRes+1) ls\_num]), [2 1 3]); |
|  |  |
|  | % analyze and plot panners for the grid of panning directions on the gain |
|  | % table |
|  | analyzeDecoder(G\_vbap, ls\_dirs, 'panner', ang\_res, 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle('Vector-base Amplitude Panning analysis - power normalization') |
|  | %% |
|  | analyzeDecoder(G\_vbip, ls\_dirs, 'panner', ang\_res, 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle('Vector-base Intensity Panning analysis') |
|  |  |
|  | %% |
|  |  |
|  | % an example of re-normalized vbap gains for amplitude normalization, e.g. |
|  | % suitable for low frequencies |
|  | g\_vbap\_renorm = g\_vbap./( sum(g\_vbap,2)\*ones(1,ls\_num) ); |
|  | G\_vbap\_renorm = permute(reshape(g\_vbap\_renorm, [(360/aziRes+1) (180/polarRes+1) ls\_num]), [2 1 3]); |
|  | analyzeDecoder(G\_vbap\_renorm, ls\_dirs, 'panner', ang\_res, 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle('Vector-base Amplitude Panning analysis - amplitude normalization') |
|  |  |
|  | %% |
|  | % It is evident that VBAP maximises the velocity vector, with zero |
|  | % directional error, while VBIP maximises the energy vector, with zero |
|  | % directional error. Both of them produce very high magnitudes for the two |
|  | % vectors since they use the minimal number of loudspeakers, at the expense |
|  | % of a variable spread with direction (which is otherwie also the minimum |
|  | % possible for each direction). |
|  |  |
|  | %% SAMPLING DECODING (SAD) |
|  | % |
|  | % The sampling, or projection, decoder, is the simplest, and essentially |
|  | % corresponds to a plane-wave decomposition at the direction of each |
|  | % loudspeaker, band-limited to the supported order of the layout (an alternative |
|  | % interpretation is that the decoder forms higher-order hypercardioids, or |
|  | % virtual microphones in ambisonic slang, to the direction of the loudspeakers). |
|  | % It is easy to see how this decoder would be robust to irregular |
|  | % loudspeaker layouts, but it won't preserve the energy of a source or |
|  | % localization cues for all directions. |
|  | % |
|  | % In the following example, analysis plots are generated for a uniform minimal |
|  | % 1st-order layout, a uniform 3rd-order layout, and a 22.0 irregular setup. |
|  |  |
|  | % tetrahedral setup |
|  | [~, ls\_dirs4\_rad] = platonicSolid('tetra'); |
|  | ls\_dirs4 = ls\_dirs4\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs4,1); |
|  |  |
|  | % get order (1 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a sampling decoder |
|  | D\_sad4 = ambiDecoder(ls\_dirs4, 'sad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_sad4, ls\_dirs4, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Sampling Decoder - ' num2str(N) 'st-order - tetrahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % dodecahedral setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a projection (sampling) decoder |
|  | D\_sad20 = ambiDecoder(ls\_dirs20, 'sad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_sad20, ls\_dirs20, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Sampling Decoder - ' num2str(N) 'st-order - dodecahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % 22.2 style loudspeaker layout - quite irregular |
|  | ls\_dirs22 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0; |
|  | 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30]'; |
|  | ls\_num = size(ls\_dirs22,1); |
|  | % get order |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get ambisonic equivalent order, for comparison |
|  | Neq = getLayoutAmbisonicOrder(ls\_dirs); |
|  | % get a projection (sampling) decoder |
|  | D\_sad22 = ambiDecoder(ls\_dirs22, 'sad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_sad22, ls\_dirs22, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Sampling Decoder - ' num2str(N) 'st-order - 22.0 layout']) |
|  |  |
|  | %% MODE-MATCHING DECODING (MMD) |
|  | % |
|  | % The mode-matching decoder is the most 'physically'-based decoder, and it |
|  | % results from equating the spherical expansion of a plane wave for an |
|  | % arbitrary direction, to a weighted spherical expansion of the plane waves |
|  | % emitted by the loudspeaker setup, solving for the weights in a |
|  | % least-square sense. The solution is simply the pseudo-inverse of the SHs |
|  | % on the direction of the speakers. Even though, mode-matching is an exact |
|  | % solution at some small region (order and frequency-dependent) around the |
|  | % sweet-spot, in practice it is useful only at low frequencies, and only |
|  | % for regular layouts, as it is very sensitive to irregularities. |
|  | % |
|  | % In the following example, analysis plots are generated for a uniform minimal |
|  | % 1st-order layout, a uniform 3rd-order layout, and a 22.0 irregular setup. |
|  |  |
|  | % tetrahedral setup |
|  | [~, ls\_dirs4\_rad] = platonicSolid('tetra'); |
|  | ls\_dirs4 = ls\_dirs4\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs4,1); |
|  |  |
|  | % get order (1 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a MMD |
|  | D\_mmd4 = ambiDecoder(ls\_dirs4, 'mmd', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_mmd4, ls\_dirs4, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Mode-matching Decoder - ' num2str(N) 'st-order - tetrahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % dodecahedral setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a MMD |
|  | D\_mmd20 = ambiDecoder(ls\_dirs20, 'mmd', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_mmd20, ls\_dirs20, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Mode-matching Decoder - ' num2str(N) 'st-order - dodecahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % 22.2 style loudspeaker layout - quite irregular |
|  | ls\_dirs22 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0; |
|  | 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30]'; |
|  | ls\_num = size(ls\_dirs22,1); |
|  | % get order |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a MMD |
|  | D\_mmd22 = ambiDecoder(ls\_dirs22, 'mmd', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_mmd22, ls\_dirs22, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Mode-matching Decoder - ' num2str(N) 'st-order - 22.0 layout']) |
|  |  |
|  | %% |
|  | % Mode-matching is problematic for irregular layouts, due to the inversion |
|  | % of the sampling matrix that becomes unstable - the effect can be seen at |
|  | % the energy amplification at directions for which the layout is quite |
|  | % sparse. |
|  |  |
|  | %% ENERGY-PRESERVING DECODING (EPAD) |
|  | % |
|  | % This approach has been devised by Zotter et al [ref.1] to address the |
|  | % energy-preserving issues of the previous two basic decoding approaches, |
|  | % especially for irregular layouts. It resembles the projection decoding, |
|  | % but with an additional singular value decomposition of the sampling |
|  | % matrix, appropriate truncation and omission of the singular values, |
|  | % resulting in a semi-orthogonal decoding matrix that preserves energy. |
|  | % |
|  | % In the following example, analysis plots are generated for a uniform minimal |
|  | % 1st-order layout, a uniform 3rd-order layout, and a 22.0 irregular setup. |
|  |  |
|  | % tetrahedral setup |
|  | [~, ls\_dirs4\_rad] = platonicSolid('tetra'); |
|  | ls\_dirs4 = ls\_dirs4\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs4,1); |
|  |  |
|  | % get order (1 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an EPAD |
|  | D\_epad4 = ambiDecoder(ls\_dirs4, 'epad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_epad4, ls\_dirs4, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Energy-preserving Decoder - ' num2str(N) 'st-order - tetrahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % dodecahedral setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an EPAD |
|  | D\_epad20 = ambiDecoder(ls\_dirs20, 'epad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_epad20, ls\_dirs20, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Energy-preserving Decoder - ' num2str(N) 'st-order - dodecahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % 22.2 style loudspeaker layout - quite irregular |
|  | ls\_dirs22 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0; |
|  | 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30]'; |
|  | ls\_num = size(ls\_dirs22,1); |
|  | % get order |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an EPAD |
|  | D\_epad22 = ambiDecoder(ls\_dirs22, 'epad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_epad22, ls\_dirs22, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Energy-preserving Decoder - ' num2str(N) 'st-order - 22.0 layout']) |
|  |  |
|  | %% ALL-ROUND AMBISONIC DECODING (ALLRAD) |
|  | % |
|  | % ALLRAD is one of the two more advanced and flexible decoding approaches, |
|  | % implemented here. It has been presented by Zotter anf Frank in [ref.2] and |
|  | % manages to handle well irregular loudspeaker setups, with low directional |
|  | % error and with good energy-preserving properties. This is achieved by a |
|  | % combination of amplitude panning (VBAP) stage rendering virtual sources |
|  | % that correspond to a uniform dense arrangement, which due to its uniformity |
|  | % satisfies all ambisonic analysis requirements. The VBAP renders this |
|  | % virtual ideal decoder to an arbitrary layout. |
|  | % |
|  | % In the following example, analysis plots are generated for a uniform minimal |
|  | % 1st-order layout, a uniform 3rd-order layout, and a 22.0 irregular setup. |
|  |  |
|  | % tetrahedral setup |
|  | [~, ls\_dirs4\_rad] = platonicSolid('tetra'); |
|  | ls\_dirs4 = ls\_dirs4\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs4,1); |
|  |  |
|  | % get order (1 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an ALLRAD |
|  | D\_allrad4 = ambiDecoder(ls\_dirs4, 'allrad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_allrad4, ls\_dirs4, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['All-round Ambisonic Decoder - ' num2str(N) 'st-order - tetrahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % dodecahedral setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an ALLRAD |
|  | D\_allrad20 = ambiDecoder(ls\_dirs20, 'allrad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_allrad20, ls\_dirs20, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['All-round Ambisonic Decoder - ' num2str(N) 'st-order - dodecahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % 22.2 style loudspeaker layout - quite irregular |
|  | ls\_dirs22 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0; |
|  | 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30]'; |
|  | ls\_num = size(ls\_dirs22,1); |
|  | % get order |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get an ALLRAD |
|  | D\_allrad22 = ambiDecoder(ls\_dirs22, 'allrad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_allrad22, ls\_dirs22, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['All-round Ambisonic Decoder - ' num2str(N) 'st-order - 22.0 layout']) |
|  |  |
|  | %% CONSTANT ANGULAR SPREAD DECODING (CSAD) |
|  | % |
|  | % CSAD is the most recent proposal for flexible and robust ambisonic decoding, |
|  | % proposed by Epain et al. in [ref.3]. Similar to ALLRAD it uses a |
|  | % panning stage, based on VBIP, an energy-based variant of VBAP, in order |
|  | % to derive gains that have zero energy-vector directional error, and |
|  | % additionally they exhibit a constant energy-vector magnitude, which |
|  | % presumable results in a perceptual angular spread/blur that does not |
|  | % change with direction. The resulting VBIP gains are approximated by an |
|  | % ambisonic decoding matrix in a least-squares sense. |
|  | % |
|  | % The version implemented here is a "lazy-man's" version, since it does not |
|  | % consider the more elaborate smooth windowing of the panning sources in |
|  | % the reference. Instead a plain rectangular angular window is used. |
|  | % However, it seems to perform well, with only small deviations compared to |
|  | % the published results. |
|  | % |
|  | % In the following example, analysis plots are generated for a uniform minimal |
|  | % 1st-order layout, a uniform 3rd-order layout, and a 22.0 irregular setup. |
|  |  |
|  | % tetrahedral setup |
|  | [~, ls\_dirs4\_rad] = platonicSolid('tetra'); |
|  | ls\_dirs4 = ls\_dirs4\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs4,1); |
|  |  |
|  | % get order (1 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a CSAD |
|  | D\_csad4 = ambiDecoder(ls\_dirs4, 'csad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_csad4, ls\_dirs4, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Constant Spread Decoder - ' num2str(N) 'st-order - tetrahedral layout']) |
|  |  |
|  | %% |
|  |  |
|  | % dodecahedral setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a CSAD |
|  | D\_csad20 = ambiDecoder(ls\_dirs20, 'csad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D\_csad20, ls\_dirs20, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Constant Spread Decoder - ' num2str(N) 'st-order - dodecahedral layout']) |
|  |  |
|  | %% |
|  | % Even though this decoder should be able to handle irregular setups, it |
|  | % seems that (at least the current implementation) the optimization part |
|  | % has difficulties with quite irregular setups like the 22.0 one. In this |
|  | % case, the decoding takes ages to return results and the results |
|  | % themselves are very poor, indicating probably that the optimization |
|  | % fails to converge. (REALLY SLOW!) |
|  |  |
|  | % % 22.2 style loudspeaker layout - quite irregular |
|  | % ls\_dirs22 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0; |
|  | % 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30]'; |
|  | % ls\_num = size(ls\_dirs22,1); |
|  | % % get order |
|  | % N = floor(sqrt(ls\_num) - 1); |
|  | % % get a CSAD |
|  | % D\_csad22 = ambiDecoder(ls\_dirs22, 'csad', 0, N); |
|  | % % analyze decoder properties |
|  | % analyzeDecoder(D\_csad22, ls\_dirs22, 'decoder', [5 5], 1, 1); |
|  |  |
|  | %% |
|  | % However, giving a more complete irregular setup, based on the 22.0 with |
|  | % only 3 additional speakers, returns fast results with excellent properties. |
|  |  |
|  | % 22.2 style loudspeaker layout with 3 extra speakers - more regular |
|  | ls\_dirs25 = [45 -45 0 135 -135 15 -15 90 -90 180 45 -45 0 135 -135 90 -90 180 0 45 -45 0 135 -135 0; |
|  | 0 0 0 0 0 0 0 0 0 0 45 45 45 45 45 45 45 45 90 -30 -30 -30 -30 -30 -90]'; |
|  | ls\_num = size(ls\_dirs25,1); |
|  | % get order |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get a CSAD |
|  | D25 = ambiDecoder(ls\_dirs25, 'csad', 0, N); |
|  | % analyze decoder properties |
|  | analyzeDecoder(D25, ls\_dirs25, 'decoder', [5 5], 1, 1); |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle(['Constant Spread Decoder - ' num2str(N) 'st-order - 22.0 layout with 3 extra speakers']) |
|  |  |
|  | %% THE MAX-rE WEIGHTING |
|  | % |
|  | % In ambisonic literature, it is generally believed that the properties of |
|  | % the energy vector are the most crucial for accurate rendering, with the |
|  | % magnitude of the vector related to localization blur, as it was mentioned |
|  | % above. Decoding at mid-high frequencies aims at optimizing the energy |
|  | % vector. Max-rE weighting (actually meaning "max energy vector magnitude" |
|  | % weighting) was introduced by Gerzon for first-order decoders, and |
|  | % formalized by Daniel [ref.4] for higher-order decoders. A derivation is |
|  | % also given by Zotter in [ref.2]. It corresponds to a per-order weighting |
|  | % of the decoder, that results in maximum-norm energy vectors, compared to |
|  | % the unweighted cases like all the previous examples. If a single decoder |
|  | % is used for all frequencies, then max-rE weighting should be preferred, |
|  | % if two (or more) decoding matrices are used for different ranges, then |
|  | % the lowest range should use unweighted decoding (which has maximum |
|  | % velocity vector, suitable for low frequencies) and weighted max-rE at all |
|  | % ranges above. Max-rE can be enabled as a flag in ambiDecoder(). |
|  | % |
|  | % Note the the Constant-angular Spread Decoder (CSAD) maximizes the energy |
|  | % vector (with constraints) by design, so max-rE weighting should not be |
|  | % applied in this case as it will affect its performance. |
|  | % |
|  | % The example below showcases the unweighted vs. max-rE magnitude of the |
|  | % energy vector, for all of the above decoders and the uniform 20-speaker |
|  | % layout. |
|  |  |
|  | ls\_num = size(ls\_dirs20,1); |
|  | % get order (3 in this case) |
|  | N = floor(sqrt(ls\_num) - 1); |
|  | % get max-rE weights up to N |
|  | a\_n = getMaxREweights(N); |
|  | % apply weighting to decoders |
|  | D\_sad20\_maxrE = D\_sad20\*diag(a\_n); |
|  | D\_mmd20\_maxrE = D\_mmd20\*diag(a\_n); |
|  | D\_epad20\_maxrE = D\_epad20\*diag(a\_n); |
|  | D\_allrad20\_maxrE = D\_allrad20\*diag(a\_n); |
|  |  |
|  | % Plots |
|  | figure |
|  | subplot(241) |
|  | [~, ~, ~, rE\_mag1] = analyzeDecoder(D\_sad20, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag1, ang\_res, ls\_dirs20, gca); |
|  | title('SAD - unweighted') |
|  | subplot(245) |
|  | [~, ~, ~, rE\_mag2] = analyzeDecoder(D\_sad20\_maxrE, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag2, ang\_res, ls\_dirs20, gca); |
|  | title('SAD - max-rE weighting') |
|  | subplot(242) |
|  | [~, ~, ~, rE\_mag1] = analyzeDecoder(D\_mmd20, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag1, ang\_res, ls\_dirs20, gca); |
|  | title('MMD - unweighted') |
|  | subplot(246) |
|  | [~, ~, ~, rE\_mag2] = analyzeDecoder(D\_mmd20\_maxrE, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag2, ang\_res, ls\_dirs20, gca); |
|  | title('MMD - max-rE weighting') |
|  | subplot(243) |
|  | [~, ~, ~, rE\_mag1] = analyzeDecoder(D\_epad20, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag1, ang\_res, ls\_dirs20, gca); |
|  | title('EPAD - unweighted') |
|  | subplot(247) |
|  | [~, ~, ~, rE\_mag2] = analyzeDecoder(D\_epad20\_maxrE, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag2, ang\_res, ls\_dirs20, gca); |
|  | title('EPAD - max-rE weighting') |
|  | subplot(244) |
|  | [~, ~, ~, rE\_mag1] = analyzeDecoder(D\_allrad20, ls\_dirs20, 'decoder', ang\_res); |
|  | plotSphericalGrid(rE\_mag1, ang\_res, ls\_dirs20, gca); |
|  | title('ALLRAD - unweighted') |
|  | subplot(248) |
|  | [~, ~, ~, rE\_mag2] = analyzeDecoder(D\_allrad20\_maxrE, ls\_dirs20, 'decoder', ang\_res, 0); |
|  | plotSphericalGrid(rE\_mag2, ang\_res, ls\_dirs20, gca); |
|  | title('ALLRAD - max-rE weighting') |
|  | h = gcf; h.Position(3) = 2.5\*h.Position(3); h.Position(4) = 1.3\*h.Position(4); |
|  | suptitle('Energy vector magnitude for unweighted and max-rE weighted decoders - dodecahedral layout') |
|  |  |
|  | %% |
|  | % The effect of the max-rE weighting is very pronounced in the case of the |
|  | % 'traditional' decoders, the sampling and mode-matching, while it is |
|  | % smaller in the case of the energy-preserving and all-round decoders. |
|  |  |
|  | %% ENCODING/DECODING AMBISONIC SIGNALS |
|  | % |
|  | % Encoding and decoding ambisonic signals is straightforward. The example |
|  | % below shows encoding two noise signals at two directions to 3rd-order |
|  | % HOA signals, and then decoding them at 84 uniformly arranged loudspeakers |
|  | % using an ALLRAD max-rE decoder. |
|  |  |
|  | % encode two signals of 5sec of noise, coming from the front and |
|  | % front-left-up |
|  | fs = 48000; |
|  | t = 5; |
|  | src\_sig = randn(t\*fs, 2); |
|  | src\_dir = [0 0; 90 30]; |
|  |  |
|  | order = 2; |
|  | hoasig = encodeHOA\_N3D(order, src\_sig, src\_dir); |
|  |  |
|  | % define a 12-speaker uniform setup |
|  | [u12, ls\_dirs12\_rad, mesh12] = platonicSolid('icosahedron'); |
|  | ls\_dirs12 = ls\_dirs12\_rad\*180/pi; |
|  | % get ALLRAD decoder |
|  | MAXRE\_ON = 1; |
|  | D\_allrad12 = ambiDecoder(ls\_dirs12, 'allrad', MAXRE\_ON, order); |
|  |  |
|  | % decode signals |
|  | lssig = decodeHOA\_N3D(hoasig, D\_allrad12); |
|  | % alternatively |
|  | % lssig = hoasig \* D\_allrad12.'; |
|  |  |
|  | % plot RMS distribution of the decoded signals, along speaker directions |
|  | Psig = sqrt(mean(lssig.^2)).'; |
|  | Sx = zeros(2,12); Sx(2,:) = u12(:,1); % speaker lines |
|  | Sy = zeros(2,12); Sy(2,:) = u12(:,2); % speaker lines |
|  | Sz = zeros(2,12); Sz(2,:) = u12(:,3); % speaker lines |
|  | figure |
|  | patch('vertices', mesh12.vertices .\* (Psig\*ones(1,3)), 'faces', mesh12.faces, 'facecolor', 'm') |
|  | line([0 1.5\*max(Psig)],[0 0],[0 0],'color','r') % axis lines |
|  | line([0 0],[0 1.5\*max(Psig)],[0 0],'color','g') % axis lines |
|  | line([0 0],[0 0],[0 1.5\*max(Psig)],'color','b') % axis lines |
|  | line(Sx,Sy,Sz,'color','k') % plot speakers |
|  | axis equal |
|  | xlabel('x'), ylabel('y'), zlabel('z'), grid, view(100,20) |
|  | h = gcf; h.Position(3:4) = 2\*h.Position(3:4); |
|  | suptitle('RMS signal power of speaker channels for two decoded sources - icosahedral layout') |
|  |  |
|  | %% FREQUENCY DEPENDENT-DECODING |
|  | % |
|  | % When HOA-encoded sound scenes are used, like in the previous example, the |
|  | % HOA signals are broadband and frequency considerations depend only on the |
|  | % reproduction side. A common approach is to use an amplitude preserving |
|  | % unweighted decoder at low frequencies, with a cutoff frequency between |
|  | % 400~700Hz depending on the room, and an energy preserving max-rE weighted |
|  | % decoder at higher frequencies (termed dual-band decoding in ambisonic slang). |
|  | % |
|  | % Decoding ambisonic recordings, however, captured with some spherical |
|  | % microphone array, is more complicated because the HOA signals themselves |
|  | % become frequency-dependent due to the microphone array properties. |
|  | % This is not very obvious with 1st-order signals (B-format), but it |
|  | % becomes very pronounced for higher-orders. E.g. the Eigenmike array |
|  | % delivers 2nd-order signals above ~500Hz and 3rd-order signals above |
|  | % ~1300Hz. Using a single decoding matrix for all mid-high frequencies may |
|  | % sound ok and it's definitely the simplest solution, but technically there |
|  | % will be some loss of power and colouration for a source captured from |
|  | % some direction and then decoded, due to the decoding matrix tuned to |
|  | % preserve energy using all HOA signals. Since certain HOA signals vanish |
|  | % at certain ranges, an alternative approach is to use as many decoding |
|  | % matrices as HOA frequency ranges, e.g. for the Eigenmike case: an |
|  | % amplitude-preserving unweighted 1st-order decoder f<500Hz, an |
|  | % energy-preserving max-rE 2nd-order decoder 500Hz<f<1200Hz, an |
|  | % energy-preserving max-rE 3rd-order decoder 1200Hz<f. |
|  | % |
|  | % Frequency dependent-decoding can be done using decodeHOA\_N3D() function, |
|  | % and passing an additional argument specifying the cutoff frequencies for |
|  | % the HOA ranges. As many decoding matrices as ranges should be defined in |
|  | % this case. A filterbank is applied internally to split the signals, |
|  | % decode the different ranges and combine the speaker outputs. |
|  |  |
|  | % encode one noise signal of 5sec of noise, coming from the front |
|  | fs = 48000; |
|  | t = 5; |
|  | src\_sig = randn(t\*fs, 1); |
|  | src\_dir = [0 0]; |
|  | order = 3; |
|  | hoasig = encodeHOA\_N3D(order, src\_sig, src\_dir); |
|  |  |
|  | % Dual-band decoding at a 20-speaker uniform setup |
|  | [~, ls\_dirs20\_rad] = platonicSolid('dodecahedron'); % dodecahedral setup |
|  | ls\_dirs20 = ls\_dirs20\_rad\*180/pi; |
|  | ls\_num = size(ls\_dirs20,1); |
|  | cutoff = 500; |
|  | order = 3; |
|  | D\_low = ambiDecoder(ls\_dirs20, 'mmd', 0, order); |
|  | D\_high = ambiDecoder(ls\_dirs20, 'allrad', 1, order); |
|  | D\_dualband = cat(3, D\_low, D\_high); |
|  | y\_dualband = decodeHOA\_N3D(hoasig, D\_dualband, cutoff, fs); |
|  |  |
|  | % Frequency/order-dependent decoding for the Eigenmike at a 20-speaker uniform setup |
|  | cutoffs = [500 1200]; |
|  | max\_order = 3; |
|  | D\_eigen = zeros(ls\_num,(max\_order+1)^2, max\_order); |
|  | % f<500Hz, 1st-order |
|  | D\_eigen(:,1:2^2,1) = ambiDecoder(ls\_dirs20, 'mmd', 0, 1); |
|  | % 500<f<1200Hz, 2nd-order |
|  | D\_eigen(:,1:3^2,2) = ambiDecoder(ls\_dirs20, 'allrad', 1, 2); |
|  | % 500<f<1200Hz, 3rd-order |
|  | D\_eigen(:,1:4^2,3) = ambiDecoder(ls\_dirs20, 'allrad', 1, 3); |
|  | y\_eigen = decodeHOA\_N3D(hoasig, D\_eigen, cutoffs, fs); |
|  |  |
|  | %% ROTATION OF AMBISONIC SOUND SCENES (REQUIRES THE SHT-LIB) |
|  | % |
|  | % Rotation of the HOA sound scene can be achieved with appropriate rotation |
|  | % matrices, designed directly in the spherical harmonic domain. Apart from |
|  | % the case of the B-format rotation, which corresponds directly to standard |
|  | % rotation matrices, HO rotation matrices can be computed through the more |
|  | % general spherical harmonic transform library by the author (see |
|  | % introduction above for the link). |
|  | % |
|  | % After the library is added to the Matlab search path, rotations can be |
|  | % performed using the rotateHOA\_N3D() function, specifying three angles on |
|  | % a yaw-pitch-roll convention. |
|  | % |
|  | % The following example encodes two sources to HOA signals, plots the |
|  | % decoding spatial power distribution, rotates the sound scene, and replots |
|  | % the rotated distribution. |
|  |  |
|  | % encode two signals of 5sec of noise, coming from the front and |
|  | % front-left-up |
|  | fs = 48000; |
|  | t = 5; |
|  | src\_sig = randn(t\*fs, 2); |
|  | src\_dir = [0 0; 0 90]; |
|  | order = 3; |
|  | hoasig = encodeHOA\_N3D(order, src\_sig, src\_dir); |
|  | % rotate the sound scene, first around Z-axis by 90deg (yaw), then around |
|  | % the new Y'-axis by 45deg (pitch), then around the new X''-axis by 45deg |
|  | % (roll). Compute each step individually for plotting. |
|  | yaw = 90; |
|  | pitch = 45; |
|  | roll = 45; |
|  | hoasig\_rot\_y = rotateHOA\_N3D(hoasig, yaw, 0, 0); |
|  | hoasig\_rot\_yp = rotateHOA\_N3D(hoasig, yaw, pitch, 0); |
|  | hoasig\_rot\_ypr = rotateHOA\_N3D(hoasig, yaw, pitch, roll); |
|  |  |
|  | % define an 84-speaker uniform setup |
|  | [u84, ls\_dirs84\_rad] = getTdesign(12); |
|  | ls\_dirs84 = ls\_dirs84\_rad\*180/pi; |
|  | mesh84.vertices = u84; |
|  | mesh84.faces = sphDelaunay(ls\_dirs84\_rad); |
|  | % get a sampling decoder |
|  | MAXRE\_ON = 1; |
|  | D\_sad84 = ambiDecoder(ls\_dirs84, 'sad', MAXRE\_ON, order); |
|  | % decode signals |
|  | LSsig84 = decodeHOA\_N3D(hoasig, D\_sad84); |
|  | % decode rotated signals |
|  | LSsig84\_rot\_y = decodeHOA\_N3D(hoasig\_rot\_y, D\_sad84); |
|  | LSsig84\_rot\_yp = decodeHOA\_N3D(hoasig\_rot\_yp, D\_sad84); |
|  | LSsig84\_rot\_ypr = decodeHOA\_N3D(hoasig\_rot\_ypr, D\_sad84); |
|  |  |
|  | % plot RMS power distribution of the decoded signals |
|  | Psig = sqrt(mean(LSsig84.^2)).'; |
|  | figure |
|  | subplot(141) |
|  | patch('vertices', mesh84.vertices .\* (Psig\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal |
|  | xlabel('x'), ylabel('y'), zlabel('z'), grid |
|  | line([0 max(Psig)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig)],'color','b') |
|  | title('unrotated') |
|  | subplot(142) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_y.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal |
|  | xlabel('x'), ylabel('y'), zlabel('z'), grid |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg)') |
|  | subplot(143) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_yp.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal |
|  | xlabel('x'), ylabel('y'), zlabel('z'), grid |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg) - pitch (45deg)') |
|  | subplot(144) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_ypr.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal |
|  | xlabel('x'), ylabel('y'), zlabel('z'), grid |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg) - pitch (45deg) - roll (45deg)'), view(3) |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); |
|  | suptitle('Successive yaw-pitch-roll rotations of decoded 3rd-order HOA signals') |
|  |  |
|  | %% |
|  | % B-format rotation requires just a regular rotation matrix, for this case |
|  | % use the rotateBformat() function. The example below duplicates the above |
|  | % HOA case, but doing everything with the traditional 1st-order B-format. |
|  |  |
|  | % encode two signals of 5sec of noise, coming from the front and |
|  | % front-left-up |
|  | fs = 48000; |
|  | t = 5; |
|  | src\_sig = randn(t\*fs, 2); |
|  | src\_dir = [0 0; 0 90]; |
|  | bfsig = encodeBformat(src\_sig, src\_dir); |
|  | % rotate the sound scene, first around Z-axis by 90deg (yaw), then around |
|  | % the new Y'-axis by 45deg (pitch), then around the new X''-axis by 45deg |
|  | % (roll). Compute each step individually for plotting. |
|  | yaw = 90; |
|  | pitch = 45; |
|  | roll = 45; |
|  | bfsig\_rot\_y = rotateBformat(bfsig, yaw, 0, 0); |
|  | bfsig\_rot\_yp = rotateBformat(bfsig, yaw, pitch, 0); |
|  | bfsig\_rot\_ypr = rotateBformat(bfsig, yaw, pitch, roll); |
|  |  |
|  | % decode signals |
|  | LSsig84 = decodeBformat(bfsig, D\_sad84); |
|  | % decode rotated signals |
|  | LSsig84\_rot\_y = decodeBformat(bfsig\_rot\_y, D\_sad84); |
|  | LSsig84\_rot\_yp = decodeBformat(bfsig\_rot\_yp, D\_sad84); |
|  | LSsig84\_rot\_ypr = decodeBformat(bfsig\_rot\_ypr, D\_sad84); |
|  |  |
|  | % plot RMS power distribution of the decoded signals |
|  | Psig = sqrt(mean(LSsig84.^2)).'; |
|  | figure |
|  | subplot(141) |
|  | patch('vertices', mesh84.vertices .\* (Psig\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal, grid |
|  | xlabel('x'), ylabel('y'), zlabel('z') |
|  | line([0 max(Psig)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig)],'color','b') |
|  | title('unrotated') |
|  | subplot(142) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_y.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal, grid |
|  | xlabel('x'), ylabel('y'), zlabel('z') |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg)') |
|  | subplot(143) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_yp.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal, grid |
|  | xlabel('x'), ylabel('y'), zlabel('z') |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg) - pitch (45deg)') |
|  | subplot(144) |
|  | Psig\_rot = sqrt(mean(LSsig84\_rot\_ypr.^2)).'; |
|  | patch('vertices', mesh84.vertices .\* (Psig\_rot\*ones(1,3)), 'faces', mesh84.faces, 'facecolor', 'm') |
|  | axis(max(Psig)\*[-1 1 -1 1 -1 1]), axis equal, grid |
|  | xlabel('x'), ylabel('y'), zlabel('z') |
|  | line([0 max(Psig\_rot)],[0 0],[0 0],'color','r'), line([0 0],[0 -max(Psig\_rot)],[0 0],'color','g'), line([0 0],[0 0],[0 -max(Psig\_rot)],'color','b') |
|  | title('yaw (90deg) - pitch (45deg) - roll (45deg)') |
|  | h = gcf; h.Position(3) = 2\*h.Position(3); h.Position(4) = 1.5\*h.Position(4); |
|  | suptitle('Successive yaw-pitch-roll rotations of decoded B-format signals') |
|  |  |
|  | %% CONVERSION BETWEEN DIFFERENT FORMATS |
|  | % |
|  | % All the HOA processing here assumes orthonormal real SHs, for the exact |
|  | % convention's details check the code of getRSH() or the documentation of |
|  | % the SHT-lib. Furthermore, indexing of HOA channels and SHs follows the |
|  | % rational single number indexing of SH components found in all other |
|  | % fields. These two conventions correspond to N3D normalization and ACN |
|  | % channel indexing in ambisonic slang. There are a few functions in the |
|  | % library that convert between these and two other common conventions - in |
|  | % case you obtain signals following them, you can use these functions to |
|  | % convert them to N3D\_ACN and apply the operations of the library. |
|  | % Similarly, you can convert back to these other conventions if you need to |
|  | % share HOA signals in that format for any reason. |
|  | % One can convert from N3D to the Schmidt semi-normalization (SN3D), and |
|  | % back, and one can convert from the ACN channel indexing to the SID |
|  | % (see [ref.11]) indexing and back. |
|  |  |
|  | % N3D to SN3D and back |
|  | hoa\_N3D\_ACN = encodeHOA\_N3D(3, 1, [0 0]); |
|  | hoa\_SN3D\_ACN = convert\_N3D\_SN3D(hoa\_N3D\_ACN, 'n2sn'); |
|  | hoa\_N3D\_ACN\_2 = convert\_N3D\_SN3D(hoa\_SN3D\_ACN, 'sn2n'); |
|  |  |
|  | % ACN to SID and back |
|  | hoa\_N3D\_SID = convert\_ACN\_SID(hoa\_N3D\_ACN, 'acn2sid'); |
|  | hoa\_N3D\_ACN\_3 = convert\_ACN\_SID(hoa\_N3D\_SID, 'sid2acn'); |
|  |  |
|  | % ACN to Bformat (1st-order only) and back. Here the sqrt(2) factor of the |
|  | % B-format is assumed to be on the dipoles. |
|  | foa\_N3D\_ACN = encodeHOA\_N3D(1, 1, [0 0]); |
|  | bf = convert\_N3D\_Bformat(foa\_N3D\_ACN, 'n2b'); |
|  | foa\_N3D\_ACN\_2 = convert\_N3D\_Bformat(bf, 'b2n'); |
|  |  |