eadf

Release 0.5

EMS Group, TU Ilmenau

May 28, 2020

CONTENTS

CHAPTER

ONE

MOTIVATION

Geometry-based MIMO channel modelling and a high-resolution parameter estimation are applications in which a precise description of the radiation pattern of the antenna arrays is required. In this package we implement an efficient representation of the polarimetric antenna response, which we refer to as the Effective Aperture Distribution Function (EADF). High-resolution parameter estimation are applications in which this reduced description permits us to efficiently interpolate the beam pattern to gather the antenna response for an arbitrary direction in azimuth and elevation. Moreover, the EADF provides a continuous description of the array manifold and its derivatives with respect to azimuth and elevation. The latter is valuable for the performance evaluation of an antenna array as well as for gradient-based parameter estimation techniques.

References

Full 3D Antenna Pattern Interpolation Using Fourier Transform Based Wavefield Modelling; S. Haefner, R. Mueller, R. S. Thomae; WSA 2016; 20th International ITG Workshop on Smart Antennas; Munich, Germany; pp. 1-8

Impact of Incomplete and Inaccurate Data Models on High Resolution Parameter Estimation in Multidimensional Channel Sounding, M. Landmann, M. Käske and R.S. Thomä; IEEE Trans. on Antennas and Propagation, vol. 60, no 2, February 2012, pp. 557-573

Efficient antenna description for MIMO channel modelling and estimation, M. Landmann, G. Del Galdo; 7th European Conference on Wireless Technology; Amsterdam; 2004; IEEE Link

Geometry-based Channel Modeling for Multi-User MIMO Systems and Applications, G. Del Galdo; Dissertation, Research Reports from the Communications Research Laboratory at TU Ilmenau; Ilmenau; 2007 Download1

Limitations of Experimental Channel Characterisation, M. Landmann; Dissertation; Ilmenau; 2008 Download2

cupy Cupy Documentation

Intel Python Intelpython Documentation

CHAPTER

TWO

EADF MAIN MODULE

This class is the central object where everything else is centered around. It has convenient methods and properties to handle:

- Fourier interpolation along Co-Elevation and Azimuth of the provided measurement data, see *eadf.eadf.EADF.pattern*,
- Fourier interpolation of the measurement data's first and second order derivatives, see *eadf.eadf.EADF.gradient* and *eadf.eadf.EADF.hessian*,
- Spline interpolation along excitation frequency of the provided measurement data, see *eadf.eadf.EADF.operatingFrequencies*,
- reduction of the coefficients used for the Fourier interpolation to allow denoising and/or faster computations, see *eadf.eadf.EADF.blockSize* and *eadf.eadf.EADF.optimizeBlockSize*,
- convenient definition of stationary subbands via *eadf.eadf.EADF.defineSubBands* and *eadf.eadf.EADF.subBandIntervals*,
- execution control of the calculations with respect to used datatypes, lower memory consumption and the used computation device, see *eadf.backend* and
- easy storage and loading from disk, see *eadf.eadf.EADF.save* and *eadf.eadf.EADF.load*.

EADF.**__init__**(*arrData: numpy.ndarray*, *arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *arrFreq: numpy.ndarray*, *arrPos: numpy.ndarray*, ***options*) → None

Initialize an EADF Object

Here we assume that the input data is given in the internal data format already. If you have antenna data, which is not in the internat data format, we advice you to use one of the importers, or implement your own.

In direction of co-elevation, we assume that both the north and the south pole were sampled, where the first sample represents the north pole and the last one the south pole. So arrCoEle must run from 0 to pi. In azimuth direction, we truncate the last sample, if we detect in arrAzi that both the first and last sample match. Both arrays have to contain values that are evenly spaced and ascending in value.

Parameters

- **arrData** (*np.ndarray*) – Co-Ele x Azi x Freq x Pol x Element
- **arrCoEle** (*np.ndarray*) – Co-elevation sampling positions in radians, both poles should be sampled
- **arrAzi** (*np.ndarray*) – Azimuth sampling positions in radians.
- **arrFreq** (*np.ndarray*) – Frequencies sampled at.

- **arrPos** (*np.ndarray*) – (3 x numElements) Positions of the single antenna elements. this is just for vizualisation purposes.

class eadf.eadf.**EADF**(*arrData: numpy.ndarray*, *arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *arrFreq: numpy.ndarray*, *arrPos: numpy.ndarray*, ***options*)

Bases: object

_calcActiveSubBands() → numpy.ndarray

_calcSubBandAssignment() → numpy.ndarray
Assing operating frequencies to their stationary subbands

This function iterates through the currently set operating frequencies and assigns them to the respective subband, just by simple search for the right interval. This way we can avoid calculating the antenna response several times for a single stationary subband.

Returns Indexing array.

Return type np.ndarray

_eval(*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *funCall*) → numpy.ndarray
Unified Evaluation Function

This function allows to calculate the Hessian, Jacobian and the values themselves with respect to the parameters angle and frequency.

Parameters

- **arrCoEle** (*np.ndarray*) – Sample at these elevations in radians
- **arrAzi** (*np.ndarray*) – Sample at these azimuths in radians
- **funCall** (*function*) – evaluatePattern, evaluateGradient, evaluateHessian

Returns pattern, gradient or Hessian array

Return type np.ndarray

_interpolateFourierData() → numpy.ndarray
Interpolate the Fourier Data along Frequency

Given the currently set operating frequencies, we evaluate splines of order 5, which were generated from the measurement data.

Returns CoEle x Azi x operatingFreq x Pol x Elem

Return type np.ndarray

property activeSubBands
Actually used stationary subbands

If the operation frequencies are such that some stationary subbands do not contain any operation frequency, we can save some computation time.

Returns indices of active subbands.

Return type np.ndarray

property arrAzi
Return Array Containing the sampled Azimuth Angles

Note: This property is read only.

Returns Sampled Azimuth Angles in radians

Return type np.ndarray

property arrCoEle
Return Array Containing the sampled Co-Elevation Angles

Note: This property is read only.

Returns Sampled Co-Elevation Angles in radians

Return type np.ndarray

property arrDataCalc
Calculation Data used for the pattern and derivatives

This data is ready to be fed into the pattern and derivatives routine. As such it is the periodified measurement data Fourier transformed along Co-Elevation and azimuth, possibly interpolated along measurement frequency and restricted to the indices determined by the compression factor. Moreover it is already in the correct datatype and also has been transferred to the computation device if necessary.

Note: This property is cacheable.

Returns Description of returned object.

Return type np.ndarray

property arrFourierData
Return the Fourier Data used to represent the antenna.

Note: It is important to know that this property's meaning depends on the fact if the user has defined stat. subbands or not. If there are no defined subbands, this property returns the Fourier data that is used to calculate the beampatterns at the specified *eadf.eadf.EADF.operatingFrequencies*. In this case it is the result of an interpolation process along the measured frequencies.

In case when there are stationary subbands, each slice along axis=2 of this property represents the data that is used to represent the beampattern on the subband with the same slice index.

Note: This property is read only.

Returns CoEle x Azi x Freq x Pol x Port shaped Fourier Data

Return type np.ndarray

property arrFreq
Return Array Containing the Sampled Frequencies

Note: This property is read only.

Returns Sampled Frequencies in Hertz

Return type np.ndarray

property arrIndAziCompress
Subselection indices for the compressed array in azimuth

This is influenced by *eadf.eadf.EADF.compressionFactor*

Note: This property is cacheable.

Returns Subselection in spatial Fourier domain in azimuth

Return type np.ndarray

property arrIndCoEleCompress
Subselection indices for the compressed array in elevation (ro)

This is influenced by *eadf.eadf.EADF.compressionFactor*.

Note: This property is cacheable.

Returns Subselection in spatial Fourier domain in elevation

Return type np.ndarray

property arrPos
Positions of the Elements as 3 x numElements

Cartesian coordinates.

Note: This property is read only.

Returns Positions of the Elements as 3 x numElements

Return type np.ndarray

property arrRawData
Return the Raw Data used during construction.

This is the already correctly along Co-Elevation and Azimuth periodified data. It is basically the measurement data in such a form that we can Fourier transform it conveniently along Co-Elevation and azimuth.

Note: This property is read only.

Returns Raw Data in 2 * Co-Ele x Azi x Freq x Pol x Element

Return type np.ndarray

property blockSize
Block Size for the Evaluation Functions

See *eadf.eadf.EADF.optimizeBlockSize* and `eadf.eadf.EADF.lowMem`.

Returns block size

Return type int

property compressionFactor
Compression Factor

Returns Compression factor in (0,1]

Return type float

defineSubBands(*intervals: numpy.ndarray*, *data: numpy.ndarray*) → None
Define Stationary SubBands

Stationary subband define intervals in frequency over which we can describe the antenna response reasonably well with one single set of CoEle x Azi x Pol x Port data values (not necessarily measurement data). This happens if the pattern changes only negligibly for the application at hand and as such can be considered constant/stationary.

If now several operating frequencies reside in the same stationary subband, one can save memory and computation time when calculating the pattern or its derivatives.

Note: Using this function triggers the spline generation and the regeneration of all the data used for calculating the pattern and so forth. Be patient.

Parameters

- **intervals** (*np.ndarray*) – Array containing the intervals bounds. Its first element must coincide with arrFreq[0] and its last with arrFreq[-1]. It must be sorted.
- **data** (*np.ndarray*) – Description of parameter *data*.

property dtype
Data Type to use during calculations

Especially if calculations are done on the GPU, switching to single floating precision can speed up calculations tremendously. Obviously one also saves some memory. It is set by the shell environment variable EADF_LOWMEM=single/double.

Note: This property is read cacheable.

Returns either 'single' for single precision or 'double' for double precision

Return type str

property frequencySplines
Frequency Interpolation Splines

These splines are used to get the pattern data for the specified *eadf.eadf.EADF.operatingFrequencies*. Since these may or may not be resided on the grid used during measurement of the array we use these splines to interpolate this data.

Note: This property is cacheable.

Returns CoEle x Azi x Pol x Elem ndarray of Splines

Return type np.ndarray

gradient (*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*) → numpy.ndarray
Sample the Beampattern Gradient at given Angles

The supplied arrays need to have the same length. The returned array has again the same length. This method samples the EADF object for given angles at operating frequencies, all polarizations and array elements. So it yields a Ang x Freq x Pol x Element ndarray.

Note: If the GPU is used for calculation a cupy.ndarray is returned, so for further processing on the host, you need to copy ot yourself. otherwise you can simply continue on the GPU device. Moreover, if you supply cupy.ndarrays with the right data types, this also speeds up the computation, since no copying or conversion have to be done.

Parameters

- **arrCoEle** (*np.ndarray*) – Sample at these elevations in radians
- **arrAzi** (*np.ndarray*) – Sample at these azimuths in radians

Returns (Ang x Freq x Pol x Elem x 2)

Return type np.ndarray

hessian (*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*) → numpy.ndarray
Sample the Beampattern Hessian at given Angles

The supplied arrays need to have the same length. The returned array has again the same length. This method samples the EADF object for given angles at operating frequencies, all polarizations and array elements. So it yields a Ang x Freq x Pol x Element ndarray.

Note: If the GPU is used for calculation a cupy.ndarray is returned, so for further processing on the host, you need to copy ot yourself. otherwise you can simply continue on the GPU device. Moreover, if you supply cupy.ndarrays with the right data types, this also speeds up the computation, since no copying or conversion have to be done.

Parameters

- **arrCoEle** (*np.ndarray*) – Sample at these elevations in radians
- **arrAzi** (*np.ndarray*) – Sample at these azimuths in radians

Returns (Ang x Freq x Pol x Elem x 2 x 2) result array, hermitian along the last two 2x2 dimensions.

Return type np.ndarray

classmethod load (*path: str*) → object
Load the Class from Serialized Data

Note: This is not safe! Make sure the requirements for pickling are met. Among these are different CPU architectures, Python versions, Numpy versions and so forth.

However we at least check the eadf package version when reading back from disk and issue a warning if the versions don't match. then you are on your own!

Parameters **path** (*str*) – Path to load from

Returns The EADF object

Return type object

property lowMemory

Does this EADF object operate in Low-Memory mode?

Low memory mode can be switched on in order to split up the Antenna repsonse calculation in blocks along the requested angles. This way only the currently needed blocks in equation (7) in the EADF paper by Landman are used to calculate a block. It should always be used together with *eadf.eadf.EADF.blockSize* in order to maximize the possibly lower computation speed. It is set by the shell environment variable EADF_LOWMEM=1/0.

Note: This property is read cacheable.

Returns Flag if low memory mode is switched on.

Return type bool

property muAziCalc

Spatial Frequencies along Azimuth

These are already in the right data type, on the right device and subselected accoring to the chosen compression factor.

Note: This property is cacheable.

Returns Description of returned object.

Return type np.ndarray

property muCoEleCalc

Spatial Frequencies along Co-Elevation

These are already in the right data type, on the right device and subselected accoring to the chosen compression factor.

Note: This property is cacheable.

Returns Description of returned object.

Return type np.ndarray

property numElements

Number of Array Elements

Note: This property is read only.

Returns Number of Antenna Elements / Ports

Return type int

property operatingFrequencies
Frequencies where we whish to evaluate the pattern

Note: Setting this triggers the recalculation of all data used for calculating the pattern.

Note: This property is writeable.

Returns Description of returned object.

Return type np.ndarray

optimizeBlockSize (*maxSize: int*) → None
Optimize the Blocksize during the Calculation

Instead of processing all angles and (possibly) frequencies all at once, we process them in blocks. This can produce a decent speedup and makes the transform scale nicer with increasing number of angles.

Simply call this function, which might take some time to determine the best block size. See *eadf.eadf.EADF.blockSize* and eadf.eadf.EADF.lowMem.

Parameters **maxSize** (*int*) – Largest Block Size?

pattern (*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*) → numpy.ndarray
Sample the Beampattern at given Angles

The supplied arrays need to have the same length. The returned array has again the same length. This method samples the EADF object for given angles at operating frequencies, all polarizations and array elements. So it yields a Ang x Freq x Pol x Element ndarray.

Note: If the GPU is used for calculation a cupy.ndarray is returned, so for further processing on the host, you need to copy ot yourself. otherwise you can simply continue on the GPU device. Moreover, if you supply cupy.ndarrays with the right data types, this also speeds up the computation, since no copying or conversion have to be done.

Parameters

- **arrCoEle** (*np.ndarray*) – Sample at these elevations in radians
- **arrAzi** (*np.ndarray*) – Sample at these azimuths in radians

Returns (Ang x Freq x Pol x Elem) result array

Return type np.ndarray

save (*path: str*) → None
Save the object to disk in a serialized way

Note: This is not safe! Make sure the requirements for pickling are met. Among these are different CPU architectures, Python versions, Numpy versions and so forth.

However we at least check the eadf package version when reading back from disk.

Parameters **path** (*str*) – Path to write to

property subBandAssignment
Assigment of the operating frequencies to stationary subbands

This property returns a list of indices, which tells us which operating frequency is in which subband. See *eadf.eadf.EADF.subBandIntervals* and *eadf.eadf.EADF.operatingFrequencies*

Note: This property is cacheable.

Returns Description of returned object.

Return type np.ndarray

property subBandIntervals
Defining array of the stationary subbands

See *eadf.eadf.EADF.defineSubBands*.

Note: This property is read only.

Returns Defining array of the stationary subbands

Return type np.ndarray

truncateCoefficients(*numCoEle: int*, *numAzi: int*) → None
Manually Truncate the Fourier Coefficients

Parameters

- **numCoEle** (*int*) – Truncation size in co-elevation
- **numAzi** (*int*) – Truncation size in azimuth

property version
Return the version of the EADF package used to create the object

This is important, if we pickle an EADF object and recreate it from disk with a possible API break between two versions of the package. Right now we only use the property to issue a warning to the user when the versions dont match when reading an object from disk.

CHAPTER

THREE

SUBMODULES

3.1 Routines to Create Synthetic Arrays

Here we provide a convenient way to generate `EADF` objects from synthetic arrays. We assume the arrays elements to be uniformly sensitive in all directions.

3.1.1 Making your Own

If you want to add your of synthetic configurations, this is the place to be. We suggest to use the generateArbitraryDipole function by just passing the locations of the elements to it and let it handle the rest.

3.1.2 The Functions

`eadf.arrays.`**`generateURA`**(*numElementsY: int*, *numElementsZ: int*, *numSpacingY: float*, *numSpacingZ: float*, *elementSize: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Uniform Rectangular Array in y-z-plane

Creates an URA in y-z-plane according to the given parameters. Depending on the dimension of elementSize the basic elements are dipoles or patches. For dipoles the elementSize has to be (1 x 1). For patches the elementSize requires a shape of (3 x 1).

The raw dipoles are located in the z-axis and the predominant polarization is vertical. The raw patches are located in the y-z-plane and the predominant polarization is horizontal.

By setting the addCrosspolPort true, each element will be duplicated with a crosspol element. So the number of elements in the EADF is also doubled. The cross-pol element is rotated about 90 degree around the x-axis to get the cross-polarization.

Example

```
>>> import eadf
>>> import numpy as np
>>> elementSize = np.ones((3, 1))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateURA(7, 5, 1.5, 0.5, elementSize, arrFreq)
```

Parameters

- **`numElementsY`** (*int*) – number of array elements in x-direction

- **numElementsZ** (*int*) – number of array elements in y-direction
- **numSpacingY** (*float*) – spacing between the first and last element in meter
- **numSpacingZ** (*float*) – spacing between the first and last element in meter
- **elementSize** (*np.ndarray*) – (dim x 1) array with size of single antenna element in meter (1 x 1) only length for dipole (3 x 1) length, width and thickness of patch element
- **arrFreq** (*np.ndarray*) – array of frequencies for EADF calculation in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant polarization.

  Defaults to false.
- ****options** – get passed to the EADF constructor

Returns URA

Return type *EADF*

eadf.arrays.**generateULA**(*numElements: int*, *numSpacing: float*, *elementSize: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Uniform Linear Array (ULA) along y-axis

Creates an ULA along y-axis according to the given parameters. Depending on the dimension of elementSize the basic elements are dipoles or patches. For dipoles the elementSize has to be (1 x 1). For patches the elementSize requires a shape of (3 x 1).

The raw dipoles are located in the z-axis and the predominant polarization is vertical. The raw patches are located in the y-z-plane and the predominant polarization is horizontal.

By setting the addCrosspolPort true, each element will be duplicated with a crosspol element. So the number of elements in the EADF is also doubled. The cross-pol element is rotated about 90 degree around the x-axis to get the cross-polarization.

Example

```
>>> import eadf
>>> import numpy as np
>>> elementSize = np.ones((3, 1))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateULA(11, 1.5, elementSize, arrFreq)
```

Parameters

- **numElements** (*int*) – number of array elements
- **numSpacing** (*float*) – spacing between the first and last element in meter
- **elementSize** (*np.ndarray*) – (dim x 1) array with size of single antenna element in meter (1 x 1) only length for dipole (3 x 1) length, width and thickness of patch element
- **arrFreq** (*np.ndarray*) – array of frequencies for EADF calculation in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant polarization.

Defaults to false.

Returns ULA

Return type *EADF*

eadf.arrays.**generateUCA**(*numElements: int*, *numRadius: float*, *elementSize: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Uniform Circular Array (UCA) in the x-y-plane

Creates an UCA in the x-y-plane according to the given parameters. Depending on the dimension of elementSize the basic elements are dipoles or patches. For dipoles the elementSize has to be (1 x 1). For patches the elementSize requires a shape of (3 x 1).

The raw dipoles are located in the z-axis and the predominant polarization is vertical. The raw patches are located in the y-z-plane and the predominant polarization is horizontal.

By setting the addCrosspolPort true, each element will be duplicated with a crosspol element. So the number of elements in the EADF is also doubled. The cross-pol element is rotated about 90 degree around the x-axis to get the cross-polarization.

Example

```
>>> import eadf
>>> import numpy as np
>>> elementSize = np.ones((3, 1))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateUCA(11, 1.5, elementSize, arrFreq)
```

Parameters

- **numElements** (*int*) – Number of Elements
- **numRadius** (*float*) – Radius of the UCA in meter
- **elementSize** (*np.ndarray*) – (dim x 1) array with size of single antenna element in meter (1 x 1) only length for dipole (3 x 1) length, width and thickness of patch element
- **arrFreq** (*np.ndarray*) – array of frequencies for EADF calculation in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant polarization.

  Defaults to false.

Returns EADF object

Return type *EADF*

eadf.arrays.**generateStackedUCA**(*numElements: int*, *numStacks: int*, *numRadius: float*, *numHeight: float*, *elementSize: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Stacked Uniform Circular Array (SUCA)

Creates a SUCA according to the given parameters. Depending on the dimension of elementSize the basic elements are dipoles or patches. For dipoles the elementSize has to be (1 x 1). For patches the elementSize requires a shape of (3 x 1).

The raw dipoles are located in the z-axis and the predominant polarization is vertical. The raw patches are located in the y-z-plane and the predominant polarization is horizontal.

By setting the addCrosspolPort true, each element will be duplicated with a crosspol element. So the number of elements in the EADF is also doubled. The cross-pol element is rotated about 90 degree around the x-axis to get the cross-polarization.

Example

```
>>> import eadf
>>> import numpy as np
>>> elementSize = np.ones((3, 1))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateSteackedUCA(11, 3, 1.5, 0.5, elementSize, arrFreq)
```

Parameters

- **numElements** (*int*) – Number of Elements per Stack > 0
- **numStacks** (*int*) – Number of Stacks > 0
- **numRadius** (*float*) – Radius of the SUCA in meter
- **numHeight** (*float*) – Displacement height between two adjacent stacks in meter
- **elementSize** (*np.ndarray*) – (dim x 1) array with size of single antenna element in meter (1 x 1) only length for dipole (3 x 1) length, width and thickness of patch element
- **arrFreq** (*np.ndarray*) – array of frequencies for EADF calculation in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant polarization.

  Defaults to false.

Returns EADF object representing this very array

Return type *EADF*

eadf.arrays.**generateArbitraryDipole**(*arrPos: numpy.ndarray*, *arrRot: numpy.ndarray*, *arrLength: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Arbitrary dipole array EADF

One specifies a (3 x N) np.ndarray to specify the elements positions and rotations in 3D cartesian space. Furthermore, a (1 x N) np.ndarray specifies the dipole-lengths of the elements. The EADF can be created for multiple frequencies at one time. The elements themselves are assumed to be uniform emitters based on finite length dipoles. One can decide if only one single pol dipole is used or if two dipoles are combined to a dual-polarimetric antenna element. This function allows to create a vast amount of different antenna geometries for quick testing.

Example

```
>>> import eadf
>>> import numpy as np
>>> arrPos = np.random.uniform(-1, 1, (3, 10))
>>> arrRot = np.zeros((3, 10))
>>> arrLength = np.ones((1, 10))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateArbitraryDipole(arrPos, arrRot, arrLength, arrFreq)
```

Parameters

- **arrPos** (*np.ndarray*) – (3 x numElements) array of positions in meter
- **arrRot** (*np.ndarray*) – (3 x numElements) array of rotations of the elements in radians
- **arrLength** (*np.ndarray*) – (1 x numElements) array of length of the individual dipole lengths in meter
- **arrFreq** (*np.ndarray*) – array of frequencies to sample in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant vertical polarization.

  Defaults to false.

Returns EADF object representing this very array

Return type *EADF*

eadf.arrays.**generateArbitraryPatch**(*arrPos: numpy.ndarray*, *arrRot: numpy.ndarray*, *arrSize: numpy.ndarray*, *arrFreq: numpy.ndarray*, *addCrossPolPort: bool = False*, ***options*) → eadf.eadf.EADF

Creates an EADF of an analyical patch antenna and returns an EADF.

One specifies a (3 x N) np.ndarray to specify the elements positions and rotations in 3D cartesian space. Furthermore, a (3 x N) np.ndarray specifies the size of the patch elements. The EADF can be created for multiple frequencies at one time. One can decide if only one single pol patch is used or if two dipoles are combined to a dual-polarimetric antenna element. This function allows to create a vast amount of different antenna geometries for qucik testing.

Example

```
>>> import eadf
>>> import numpy as np
>>> arrPos = np.random.uniform(-1, 1, (3, 10))
>>> arrRot = np.zeros((3, 10))
>>> arrSize = np.ones((3, 10))
>>> arrFreq = np.arange(1,4)
>>> A = eadf.generateArbitraryPatch(arrPos, arrRot, arrSize, arrFreq)
```

Parameters

- **arrPos** (*np.ndarray*) – (3 x numElements) array of positions in meter
- **arrRot** (*np.ndarray*) – (3 x numElements) array of rotations of the elements in radians

- **arrSize** (*np.ndarray*) – (3 x numElements) array of sizes of the patches (length, width, thickness) in meter
- **arrFreq** (*np.ndarray*) – array of frequencies to sample in Hertz
- **addCrossPolPort** (*bool*) – Should we have appropriately rotated cross-pol ports? If true, the virtual elements will have two cross-polarized ports and are described as two elements in the EADF object. If false, the array has a predominant horizontal polarization.

  Defaults to false.

Returns EADF object

Return type *EADF*

3.2 The Backend

3.2.1 Memory Management

For some calculations it might be a smart idea to process the requested angles in chunks of a certain size. First, because it might result in fewer cache misses and second one might be running out of RAM to run the calculations. To this end, you might set the EADF_LOWMEM environment variable and use it together with the *blockSize* property and the *optimizeBlockSize* function.

3.2.2 Datatypes

We support single and double precision calculations. This can be set with the EADF_DTYPE environment variable. Most of the time single precision allows approximately twice the computation speed. This is reflected in the *dtype* environment variable.

3.2.3 Environment Variables

A list of all available shell environment variables.

- EADF_LOWMEM: yay or nay?
- EADF_DTYPE: single or double?

`eadf.backend.`**rDtype**
alias of `numpy.float64`

`eadf.backend.`**cDtype**
alias of `numpy.complex128`

3.3 Mathematical Core Routines

In this submodule we place all the mathematical and general core routines which are used throughout the package. These are not intended for direct use, but are still documented in order to allow new developers who are unfamiliar with the code base to get used to the internal structure.

eadf.core.**calcBlockSize**(*muCoEle: numpy.ndarray*, *muAzi: numpy.ndarray*, *arrData: numpy.ndarray*, *blockSizeMax: int*, *lowMem: bool*) → int

Calculate an optimized block size

This function steadily increases the blockSize during the pattern transform in order to optimze it for execution time. We only use a very crude metric and a very naive measurement for execution time. However, it is a starting point.

Parameters

- **muCoEle** (*np.ndarray*) – co-elevation DFT frequencies
- **muAzi** (*np.ndarray*) – azimuth DFT frequencies
- **arrData** (*np.ndarray*) – Fourier coefficients of the array
- **blockSizeMax** (*int*) – Maximum block Size
- **lowMem** (*bool*) – Is it the low memory mode?

Returns block size

Return type int

eadf.core.**inversePatternTransform**(*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *arrData: numpy.ndarray*, *blockSize: int*) → numpy.ndarray

Samples the Pattern by using the Fourier Coefficients

This function does the heavy lifting in the EADF evaluation process. It is used to sample the beampattern and the derivative itself, by evaluating d_phi * Gamma * d_theta^t as stated in (6) in the EADF paper by Landmann and delGaldo. It broadcasts this product over the last three coordinates of the fourier data, so across all wave frequency bins, polarisations and array elements.

By changing d_theta(arrCoEle) and d_phi (arrAzi) acordingly in the arguments one can calculate either the derivative or the pattern itself.

Parameters

- **arrCoEle** (*np.ndarray*) – array of fourier kernels in co-elevation direction
- **arrAzi** (*np.ndarray*) – array of fourier kernels in azimuth direction
- **arrData** (*np.ndarray*) – the Fourier coefficients to use
- **blockSize** (*int*) – number of angles to process at once

Returns beam pattern values at arrCoEle, arrAzi

Return type np.ndarray

eadf.core.**inversePatternTransformLowMem**(*arrCoEle: numpy.ndarray, arrAzi: numpy.ndarray, funCoEle: Callable[[numpy.ndarray], numpy.ndarray], funAzi: Callable[[numpy.ndarray], numpy.ndarray], arrData: numpy.ndarray, blockSize: int*) → numpy.ndarray

Samples the Pattern by using the Fourier Coefficients

This function does the heavy lifting in the EADF evaluation process. It is used to sample the beampattern and the derivative itself, by evaluating d_phi * Gamma * d_theta^t as stated in (6) in the EADF paper by Landmann and delGaldo. It broadcasts this product over the last three coordinates of the fourier data, so across all polarisations, wave frequency bins and array elements.

However, the matrices containing the complex exponentials are calculated block wise and on the fly.

By changing d_theta(arrCoEle) and d_phi (arrAzi) acordingly in the arguments one can calculate either the derivative or the pattern itself.

Parameters

- **arrCoEle** (*np.ndarray*) – array of fourier kernels in co-elevation direction
- **arrAzi** (*np.ndarray*) – array of fourier kernels in azimuth direction
- **funAzi** (*method*) – function that generates transform matrix in azimuth direction
- **funCoEle** (*method*) – function that generates transform matrix in frequency direction
- **arrData** (*np.ndarray*) – the Fourier coefficients to use
- **blockSize** (*int*) – number of blocks to transform at once

Returns beam pattern values at arrCoEle, arrAzi

Return type np.ndarray

eadf.core.**evaluatePattern**(*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *muCoEle: numpy.ndarray*, *muAzi: numpy.ndarray*, *arrData: numpy.ndarray*, *blockSize: int*, *lowMem: bool*) → numpy.ndarray

Sample the Beampattern at Arbitrary Angles

Parameters

- **arrCoEle** (*np.ndarray*) – co-elevation angles to sample at in radians
- **arrAzi** (*np.ndarray*) – azimuth angles to sample at in radians
- **muCoEle** (*np.ndarray*) – spatial frequency bins in co-elevation direction
- **muAzi** (*np.ndarray*) – spatial frequency bins in azimuth direction
- **arrData** (*np.ndarray*) – fourier coefficients
- **blockSize** (*int*) – number of angles / frequencies to process at once
- **lowMem** (*bool*) – should we save memory?

Returns sampled values

Return type np.ndarray

eadf.core.**evaluateGradient**(*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *muCoEle: numpy.ndarray*, *muAzi: numpy.ndarray*, *arrData: numpy.ndarray*, *blockSize: int*, *lowMem: bool*) → numpy.ndarray

Sample the Beampattern Gradients at Arbitrary Angles

Parameters

- **arrCoEle** (*np.ndarray*) – co-elevation angles to sample at in radians
- **arrAzi** (*np.ndarray*) – azimuth angles to sample at in radians
- **muCoEle** (*np.ndarray*) – spatial frequency bins in co-elevation direction
- **muAzi** (*np.ndarray*) – spatial frequency bins in azimuth direction
- **arrData** (*np.ndarray*) – fourier coefficients

- **blockSize** (*int*) – number of angles / frequencies to process at once
- **lowMem** (*bool*) – should we save memory?

Returns

Return type np.ndarray

eadf.core.**evaluateHessian**(*arrCoEle: numpy.ndarray*, *arrAzi: numpy.ndarray*, *muCoEle: numpy.ndarray*, *muAzi: numpy.ndarray*, *arrData: numpy.ndarray*, *blockSize: int*, *lowMem: bool*) → numpy.ndarray

Sample the Beampattern Hessian Matrix at Arbitrary Angles

Parameters

- **arrCoEle** (*np.ndarray*) – co-elevation angles to sample at in radians
- **arrAzi** (*np.ndarray*) – azimuth angles to sample at in radians
- **muCoEle** (*np.ndarray*) – spatial frequency bins in co-elevation direction
- **muAzi** (*np.ndarray*) – spatial frequency bins in azimuth direction
- **arrData** (*np.ndarray*) – fourier coefficients
- **blockSize** (*int*) – number of angles to process at once
- **lowMem** (*bool*) – should we save memory?

Returns

Return type np.ndarray

3.4 Importers

Here we provide a collection of several importers to conveniently create EADF objects from various data formats. For this purpose we provide a set of so called *handshake formats*, which can be seen as intermediate formats, which facilitate the construction of importers, since for these handshake formats we already provide tested conversion routines to the internal data format.

For these formats there are readily available and tested importers. See the respective importer methods for further details.

- **Regular (in space) Angular Data: 2*co-ele x azi x pol x freq x elem.** This format is simply handled by the EADF class initialization. So, if your data is already in that format, just call EADF() with it.
- Regular (in space) Spatial Fourier Data: 2*co-ele-sptfreq x azi-sptfreq x pol x freq x elem
- Angle List Data: Ang x Pol x Freq x Elem
- Regular (in space) Variation 2 Angular Data: ele x azi x elem x pol x freq.
- **Wideband Angular Data as struct from .mat file:** This format accounts for array pattern information stored with a .mat file containing a Matlab structured array (struct) with 'Value' field having dimensions; ele x azi x elem x pol x freq.
- **Narrowband Angular Data as struct from .mat file:** This format accounts for array pattern information stored with a .mat file containing a Matlab structured array (struct) with 'Value' field having dimensions; ele x azi x elem x pol.
- HFSS export (.csv Files)

eadf.importers.**fromAngleListData**(*arrCoEleData: numpy.ndarray*, *arrAziData: numpy.ndarray*, *arrAngleListData: numpy.ndarray*, *arrFreqData: numpy.ndarray*, *arrPos: numpy.ndarray*, *numCoEle: int*, *numAzi: int*, *numErrorTol=0.0001*, *method='SH'*, ***eadfOptions*) → eadf.eadf.EADF

Importer from the Angle List Data Handshake format

This format allows to specify a list of angles (ele, azi)_i and beam pattern values v_i = (pol, freq, elem)_i which are then interpolated along the two angular domains to get a regular grid in azimuth and co-elevation. By default this is done using vector spherical harmonics, since they can deal with irregular sampling patterns quite nicely. In this format for each angular sampling point, we need to have excited the array elements with the same frequencies.

Parameters

- **arrCoEleData** (*np.ndarray*) – Sampled Co-elevation Angles in radians
- **arrAziData** (*np.ndarray*) – Sampled Azimuth Angles in radians
- **arrAngleListData** (*np.ndarray*) – List in Angle x Freq x Pol x Element format
- **arrFreqData** (*np.ndarray*) – Frequencies the array was excited with in ascending order
- **arrPos** (*np.ndarray*) – Positions of the array elements
- **numCoEle** (*int*) – number of regular elevation samples used during interpolation > 0
- **numAzi** (*int*) – number of regular azimuth samples used during interpolation > 0
- **numErrorTol** (*float*) – error tolerance for coefficients fitting > 0
- **method** (*string*) – Interpolation Method, default='SH'
- **eadfOptions** – Things to tell the EADF constructor

Returns Created Array

Return type *EADF*

eadf.importers.**fromWidebandAngData**(*path: str*, ***kwargs*) → eadf.eadf.EADF

This format defines angular data over uniformly sampled ele and azi for a range of antenna ports elem, with v and/or h-polarization pol for frequency range freq. Using the importer it is possible to choose the respective indices for antenna ports, polarization and frequency to derive the EADF.

The .mat file which utilizes this importer is expected to consist of a struct named 'pattern' which includes the fields: 'Dim', 'Value', 'Description', 'Unit', and 'Date'.

The 'Dim' field is itself a struct which describes the dimensions of the 5-dimensional 'Value' tensor. 'Dim' contains Name-Value pairs corresponding to and ordered as 'Elevation', 'Azimuth', 'Element', 'Polarization', and 'Frequency'.

The 'Elevation' and 'Azimuth' fields contain the vectors with the range of their respective angles.The 'Element' field contains the indices of the measured array ports starting from 1, while the 'Polarization' field is a cell array structured as {'h'} and/or {'v'}. The 'Frequency' field contains an array of the available frequency points.

The 'Value' field is a tensor containing complex-double type data and is of the size (numEle x numAzi x numElem x numPol x numFreq).

The 'Description' field contains a string describing what data is included in the file. Typically, it is 'sqrt of abs of complex realised gain'.

The 'Unit' field is a string denoting if the data is in the linear or logarithmic units. Typically, it is expected to be 'linear'.

The 'Date' field is a string denoting the date and time when the measurement campaign was completed and the data was stored.

Parameters

- **path** (*string*) – Directory location and name of .mat file containing measurement data
- **arrPorInd** (*numpy.array, default = [], optional*) – Indices of antenna ports
- **arrPol** (*numpy.array, default = ['h','v'], optional*) – Strings corresponding to polarization ('h' and/or 'v')
- **arrFreq** (*numpy.array, default = 0, optional*) – Frequency points in Hertz

Returns Created Array

Return type *EADF*

eadf.importers.**fromNarrowBandAngData**(*path: str*, ***kwargs*) → eadf.eadf.EADF

This format defines angular data over uniformly sampled ele and azi for a range of antenna ports elem, with v and/or h-polarization pol. Using the importer it is possible to choose the respective indices for antenna ports, polarization to derive the EADF.

The .mat file which utilizes this importer is expected to consist of a struct named 'pattern' which includes the fields: 'Dim', 'Value'.

The 'Dim' field is itself a struct which describes the dimensions of the 4-dimensional 'Value' tensor. 'Dim' contains Name-Value pairs corresponding to and ordered as 'Elevation', 'Azimuth', 'Element', and 'Polarization'.

The 'Elevation' and 'Azimuth' fields contain the vectors with the range of their respective angles.The 'Element' field contains the indices of the measured array ports starting from 1, while the 'Polarization' field is a cell array structured as {'h'} and/or {'v'}.

The 'Value' field is a tensor containing complex-double type data and is of the size (numEle x numAzi x numElem x numPol).

Parameters

- **path** (*string*) – Directory location and name of .mat file containing measurement data
- **arrPorInd** (*numpy.array, default = [], optional*) – Indices of antenna ports
- **arrPol** (*numpy.array, default = ['h','v'], optional*) – Strings corresponding to polarization ('h' and/or 'v')

Returns Created Array

Return type *EADF*

eadf.importers.**fromArraydefData**(*path: str*) → eadf.eadf.EADF

Import from Matlab arraydef structure used for calibrated arrays at TUI.

Parameters **path** (*str*) – path to .mat-file

Returns EADF object from the respective data

Return type *EADF*

eadf.importers.**fromHFSS**(**filenames*, ***options*)

Importer from HFSS .csv exports.

This format allows to specify a list of angles (azi, ele)_i and beam pattern values v_i = (pol, freq, elem)_i which are then interpolated along the two angular domains to get a regular grid in azimuth and co-elevation. By

default this is done using vector spherical harmonics, since they can deal with irregular sampling patterns quite nicely. In this format for each angular sampling point, we need to have excited the array elements with the same frequencies.

Parameters

- ***filenames** (*str*) – One or multiple filenames, corresponding to .csv HFSS output files.
- **key_CoElevation** (*str, optional*) – The HFSS parameter name corresponding to the Azimuth dimension.

  Defaults to *Theta*.
- **key_Azimuth** (*str, optional*) – The HFSS parameter name corresponding to the Co-Elevation dimension.

  Defaults to *phi*.
- **key_Frequency** (*str, optional*) – The HFSS parameter name corresponding to the frequency dimension.

  Defaults to *freq*.
- **key_Polarization** (*tuple of str, optional*) – An immutable tuple of HFSS type identifiers, that shall populate the polarization dimension of the EADF tensor.

  Defaults to *('rEPhi', 'rETheta')*.
- **position** (numpy.ndarray, optional) – The position of the antenna, given in [3 x 1] coordinates.
- ****options** (*kwargs*) – Further key-worded arguments will be passed on to the construction of the EADF object

Returns Created Array

Return type *EADF*

3.5 Preprocessing Methods

This module hosts several preprocessing methods that can be used during and before construction of an EADF object.

eadf.preprocess.**sampledToFourier**(*arrData: numpy.ndarray*) → tuple

Transform the regularly sampled data in frequency domain

Here we assume that the data is already flipped along co-elevation, rotated along azimuth as described in the EADF paper and in the wideband case it is also periodified in excitation frequency direction such that we can just calculate the respective 2D/3D FFT from this along the first two /three axes.

Parameters **data** (*np.ndarray*) – Raw sampled and periodified data in the form 2 * co-ele x azi x freq x pol x elem

Returns Fourier Transform and the respective sample frequencies

Return type (np.ndarray, np.ndarray, np.ndarray)

eadf.preprocess.**setCompressionFactor**(*arrFourierData: numpy.ndarray*, *numCoEleInit: int*, *numAziInit: int*, *numValue: float*) → tuple

Calculate Subselection-Indices

This method takes the supplied compression factor, which is not with respect to the number of Fourier coefficients to use but rather the amount of energy still present in them. this is achieved by analysing the spatial spectrum of the whole array in the following way:

1. Flip the spectrum all into one quadrant
2. normalize it with respect to the complete energy
3. find all combinations of subsizes in azimuth and elevation such that the energy is lower than numValue
4. find the pair of elevation and azimuth index such that it minimizes the execution time during sampling

Parameters

- **arrFourierData** (*np.ndarray*) – the description of the array in frequency domain.
- **numCoEleInit** (*int*) – number of coelevation samples
- **numAziInit** (*int*) – number of azimuth samples
- **numValue** (*float*) – Compression Factor we would like to have

Returns compression factor and subselection indices.

Return type tuple

eadf.preprocess.**splineInterpolateFrequency**(*arrFreq: numpy.ndarray*, *arrData: numpy.ndarray*) → numpy.ndarray

Spline Interpolation of Pattern Data in Frequency

Splines!

Parameters **arrData** (*np.ndarray*) – data to pad

Returns of interpolation splines

Return type np.ndarray

eadf.preprocess.**symmetrizeData**(*arrA: numpy.ndarray*) → numpy.ndarray

Generate a symmetrized version of a regularly sampled array data

This function assumes that we are given the beam pattern sampled in co-elevation and azimuth on a regular grid, as well as for at most 2 polarizations and all the same wave-frequency bins. Then this function applies (2) in the original EADF paper. So the resulting array has the same dimensions but 2*n-1 the size in co-elevation direction, if n was the original co-elevation size.

Parameters **arrA** (*np.ndarray*) – Input data (co-elevation x azimuth x pol x freq x elem).

Returns Output data (2*co-elevation - 2 x azimuth x pol x freq x elem).

Return type np.ndarray

eadf.preprocess.**regularSamplingToGrid**(*arrA: numpy.ndarray*, *numCoEle: int*, *numAzi: int*) → numpy.ndarray

Reshape an array sampled on a 2D grid to actual 2D data

Parameters

- **arrA** (*np.ndarray*) – Input data *arrA* (2D angle x pol x freq x elem).
- **numCoEle** (*int*) – Number of samples in co-elevation direction.
- **numAzi** (*int*) – Number of samples in azimuth direction.

Returns Output data (co-elevation x azimuth x freq x pol x elem).

Return type np.ndarray

eadf.sphericalharm.**interpolateDataSphere**(*arrCoEleSample: numpy.ndarray*, *arrAziSample: numpy.ndarray*, *arrValues: numpy.ndarray*, *arrCoEleInter: numpy.ndarray*, *arrAziInter: numpy.ndarray*, ***options*) → numpy.ndarray

Interpolate Data located on a Sphere

This method can be used for interpolating a function of the form f : S^2 -> C which is sampled on N arbitrary positions on the sphere. The input data is assumed to be in the format N x M1 x ... and the interpolation is broadcasted along M1 x ... The interpolation is always done using least squares, so for noisy data or overdetermined data with respect to the basis you should not encounter any problems.

Methods

- *SH* (Spherical Harmonics), see dissertation delGaldo, For these you have to supply *numN* as a kwarg, which determines the order of the SH basis. The number of total basis functions is then calculated via *numN x (numN + 1) + 1*. default=6

Parameters

- **arrCoEleSample** (*np.ndarray*) – Sampled Co-Elevation positions in radians
- **arrAziSample** (*np.ndarray*) – Sampled Azimuth positions in radians
- **arrValues** (*np.ndarray*) – Sampled values
- **arrCoEleInter** (*np.ndarray*) – CoElevation positions we want the function to be evaluated in radians
- **arrAziInter** (*np.ndarray*) – Azimuth Positions we want the function to be evaluated in radians
- **method** (*type, optional, default='SH'*) – 'SH'(default) for spherical harmonics
- ****options** (*type*) – Depends on method, see above

Returns Description of returned object.

Return type np.ndarray

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V