# e13Tools documentation

**Ellert van der Velden**

**Mar 25, 2021**

# User Documentation

This is the documentation for the *e13Tools* package, a collection of utility functions that were created by @1313e. It is written in pure Python 3 and publicly available on GitHub.

---

The documentation of *e13Tools* is spread out over several sections:

CHAPTER 1

# Getting started

## 1.1 Installation

*e13Tools* can be easily installed by either cloning the repository and installing it manually:

```
$ git clone https://github.com/1313e/e13Tools
$ cd e13Tools
$ pip install .
```

or by installing it directly from PyPI with:

```
$ pip install e13tools
```

*e13Tools* can now be imported as a package with `import e13tools`. For using some functions in *e13Tools*, `astropy >= 1.3` is required (not installed automatically).

CHAPTER 2

# Community guidelines

*e13Tools* is an open-source and free-to-use software package (and it always will be), provided under the BSD-3 license (see below for the full license).

Users are highly encouraged to make contributions to the package or request new features by opening a GitHub issue. If you would like to contribute to the package, but do not know what, then there are quite a few ToDos in the code that may give you some inspiration. As with contributions, if you find a problem or issue with *e13Tools*, please do not hesitate to open a GitHub issue about it.

## 2.1 License

```
BSD 3-Clause License

Copyright (c) 2017-2021, Ellert van der Velden
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
```

(continues on next page)

(continued from previous page)

```
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

# CHAPTER 3

# Core

Provides a collection of functions that are core to **e13Tools** and are imported automatically.

**exception** e13tools.core.**InputError**
Generic exception raised for errors in the function input arguments.

General purpose exception class, raised whenever the function input arguments prevent the correct execution of the function without specifying the type of error (eg. ValueError, TypeError, etc).

**__weakref__**
list of weak references to the object (if defined)

**exception** e13tools.core.**ShapeError**
Inappropriate argument shape (of correct type).

**__weakref__**
list of weak references to the object (if defined)

e13tools.core.**compare_versions**(*a*, *b*)
Compares provided versions *a* and *b* with each other, and returns *True* if version *a* is later than or equal to version *b*.

# CHAPTER 4

# Math

Provides a collection of functions useful in various mathematical calculations.

e13tools.math.**gcd**(**args*)

Returns the greatest common divisor of the provided sequence of integers.

**Parameters** **args** (*tuple of int*) – Integers to calculate the greatest common divisor for.

**Returns** **gcd** (*int*) – Greatest common divisor of input integers.

### Example

```
>>> gcd(18, 60, 72, 138)
6
```

**See also:**

***lcm()*** Least common multiple for sequence of integers.

e13tools.math.**is_PD**(*matrix*)

Checks if *matrix* is positive-definite or not, by using the `cholesky()` function. It is required for *matrix* to be Hermitian.

**Parameters** **matrix** (*2D array_like*) – Matrix that requires checking.

**Returns** **out** (*bool*) – *True* if *matrix* is positive-definite, *False* if it is not.

### Examples

Using a real matrix that is positive-definite (like the identity matrix):

```
>>> matrix = np.eye(3)
>>> matrix
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> is_PD(matrix)
True
```

Using a real matrix that is not symmetric (Hermitian):

```
>>> matrix = np.array([[1, 2], [3, 4]])
>>> matrix
array([[1, 2],
       [3, 4]])
>>> is_PD(matrix)
Traceback (most recent call last):
    ...
ValueError: Input argument 'matrix' must be Hermitian!
```

Using a complex matrix that is positive-definite:

```
>>> matrix = np.array([[4, 1.5+1j], [1.5-1j, 3]])
>>> matrix
array([[ 4.0+0.j,  1.5+1.j],
       [ 1.5-1.j,  3.0+0.j]])
>>> is_PD(matrix)
True
```

**See also:**

***nearest_PD()*** Find the nearest positive-definite matrix to the input *matrix*.

e13tools.math.**lcm**(**args*)

Returns the least common multiple of the provided sequence of integers. If at least one integer is zero, the output will also be zero.

**Parameters** **args** (*tuple of int*) – Integers to calculate the least common multiple for.

**Returns** **lcm** (*int*) – Least common multiple of input integers.

### Example

```
>>> lcm(8, 9, 21)
504
```

**See also:**

***gcd()*** Greatest common divisor for sequence of integers.

e13tools.math.**nCr**(*n*, *r*, *repeat=False*)

For a given set S of *n* elements, returns the number of unordered arrangements ("combinations") of length *r* one can make with S. Returns zero if $r > n$ and *repeat* is *False*.

**Parameters**

- **n** (*int*) – Number of elements in the set S.

- **r** (*int*) – Number of elements in the sub-set of set S.

**Other Parameters** **repeat** (*bool. Default: False*) – If *False*, each element in S can only be chosen once. If *True*, they can be chosen more than once.

**Returns** **n_comb** (*int*) – Number of "combinations" that can be made with S.

### Examples

```
>>> nCr(4, 2)
6
```

```
>>> nCr(4, 2, repeat=True)
10
```

```
>>> nCr(2, 4, repeat=True)
5
```

```
>>> nCr(2, 4)
0
```

**See also:**

***nPr()*** Returns the number of ordered arrangements.

e13tools.math.**nearest_PD**(*matrix*)

Find the nearest positive-definite matrix to the input *matrix*.

**Parameters** **matrix** (*2D array_like*) – Input matrix that requires its nearest positive-definite variant.

**Returns** **mat_PD** (2D `ndarray` object) – The nearest positive-definite matrix to the input *matrix*.

---

**Notes**

This is a Python port of John D'Errico's *nearestSPD* code[1], which is a MATLAB implementation of Higham (1988)[2].

According to Higham (1988), the nearest positive semi-definite matrix in the Frobenius norm to an arbitrary real matrix $A$ is shown to be

$$\frac{B + H}{2},$$

with $H$ being the symmetric polar factor of

$$B = \frac{A + A^T}{2}.$$

On page 2, the author mentions that all matrices $A$ are assumed to be real, but that the method can be very easily extended to the complex case. This can indeed be done easily by taking the conjugate transpose instead of the normal transpose in the formula on the above.

---

[1] https://www.mathworks.com/matlabcentral/fileexchange/42885-nearestspd

[2] N.J. Higham, "Computing a Nearest Symmetric Positive Semidefinite Matrix" (1988): https://doi.org/10.1016/0024-3795(88)90223-6

### References

### Examples

Requesting the nearest PD variant of a matrix that is already PD results in it being returned immediately:

```
>>> matrix = np.eye(3)
>>> matrix
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> is_PD(matrix)
True
>>> nearest_PD(matrix)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Using a real non-PD matrix results in it being transformed into an PD-matrix:

```
>>> matrix = np.array([[1, 2], [3, 4]])
>>> matrix
array([[1, 2],
       [3, 4]])
>>> is_PD(matrix)
Traceback (most recent call last):
    ...
ValueError: Input argument 'matrix' must be Hermitian!
>>> mat_PD = nearest_PD(matrix)
>>> mat_PD
array([[ 1.31461828,  2.32186616],
       [ 2.32186616,  4.10085767]])
>>> is_PD(mat_PD)
True
```

Using a complex non-PD matrix converts it into the nearest complex PD-matrix:

```
>>> matrix = np.array([[4, 2+1j], [1+3j, 3]])
>>> matrix
array([[ 4.+0.j,  2.+1.j],
       [ 1.+3.j,  3.+0.j]])
>>> mat_PD = nearest_PD(matrix)
>>> mat_PD
array([[ 4.0+0.j,  1.5-1.j],
       [ 1.5+1.j,  3.0+0.j]])
>>> is_PD(mat_PD)
True
```

**See also:**

***is_PD()*** Checks if *matrix* is positive-definite or not.

e13tools.math.**nPr**(*n*, *r*, *repeat=False*)

For a given set S of *n* elements, returns the number of ordered arrangements ("permutations") of length *r* one can make with S. Returns zero if $r > n$ and *repeat* is *False*.

**Parameters**

- **n** (*int*) – Number of elements in the set S.
- **r** (*int*) – Number of elements in the sub-set of set S.

**Other Parameters** **repeat** (*bool. Default: False*) – If *False*, each element in S can only be chosen once. If *True*, they can be chosen more than once.

**Returns** **n_perm** (*int*) – Number of "permutations" that can be made with S.

### Examples

```
>>> nPr(4, 2)
12
```

```
>>> nPr(4, 2, repeat=True)
16
```

```
>>> nPr(2, 4, repeat=True)
16
```

```
>>> nPr(2, 4)
0
```

**See also:**

***nCr()*** Returns the number of unordered arrangements.

# CHAPTER 5

# NumPy

Provides a collection of functions useful in manipulating *NumPy* arrays.

e13tools.numpy.**diff**(*array1*, *array2=None*, *axis=0*, *flatten=True*)

Calculates the pair-wise differences between inputs *array1* and *array2* over the given *axis*.

**Parameters** **array1** (*array_like*) – One of the inputs used to calculate the pair-wise differences.

**Other Parameters**

- **array2** (*array_like or None. Default: None*) – The other input used to calculate the pair-wise differences. If *None*, *array2* is equal to *array1*. If not *None*, the length of all axes except *axis* must be equal for both arrays.
- **axis** (*int. Default: 0*) – Over which axis to calculate the pair-wise differences. Default is over the first axis. A negative value counts from the last to the first axis.
- **flatten** (*bool. Default: True*) – If *array2* is *None*, whether or not to calculate all pair-wise differences. If *True*, a flattened array containing all above-diagonal pair-wise differences is returned. This is useful if only off-diagonal terms are required and the sign is not important. If *False*, an array with all pair-wise differences is returned.

**Returns** **diff_array** (ndarray object) – Depending on the input parameters, an array with n dimensions containing the pair-wise differences between *array1* and *array2* over the given *axis*.

### Examples

Using two matrices returns the pair-wise differences in row-vectors:

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[4, 5, 6], [7, 8, 9]])
>>> diff(mat1, mat2)
array([[[-3., -3., -3.],
        [-6., -6., -6.]],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
       [[ 0.,  0.,  0.],
        [-3., -3., -3.]]])
```

Setting *axis* to 1 returns the pair-wise differences in column-vectors:

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[4, 5, 6], [7, 8, 9]])
>>> diff(mat1, mat2, axis=1)
array([[[-3., -3.],
        [-4., -4.],
        [-5., -5.]],
<BLANKLINE>
       [[-2., -2.],
        [-3., -3.],
        [-4., -4.]],
<BLANKLINE>
       [[-1., -1.],
        [-2., -2.],
        [-3., -3.]]])
```

Only using a single matrix returns the pair-wise differences in row-vectors in that matrix (either flattened or not):

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> diff(mat, flatten=True)
array([[-3., -3., -3.]])
>>> diff(mat, flatten=False)
array([[[ 0.,  0.,  0.],
        [-3., -3., -3.]],
<BLANKLINE>
       [[ 3.,  3.,  3.],
        [ 0.,  0.,  0.]]])
```

Using a matrix and a vector returns the pair-wise differences in row-vectors:

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> vec = np.array([7, 8, 9])
>>> diff(mat, vec)
array([[-6, -6, -6],
       [-3, -3, -3]])
```

Using two vectors returns the pair-wise differences in scalars:

```
>>> vec1 = np.array([1, 2, 3])
>>> vec2 = np.array([4, 5, 6])
>>> diff(vec1, vec2)
array([[-3., -4., -5.],
       [-2., -3., -4.],
       [-1., -2., -3.]])
```

e13tools.numpy.**intersect**(*array1*, *array2*, *axis=0*, *assume_unique=False*)

Finds the intersection between given arrays *array1* and *array2* over provided *axis* and returns the unique elements that are both in *array1* and *array2*.

This is an nD-version of NumPy's `intersect1d()` function.

**Parameters**

- **array1** (*array_like*) – Input array.
- **array2** (*array_like*) – Comparison array with same shape as *array1* except in given *axis*.

**Other Parameters**

- **axis** (*int or None. Default: 0*) – Axis over which elements must be checked in both arrays. A negative value counts from the last to the first axis. If *None*, both arrays are flattened first (this is the functionality of `intersect1d()`).
- **assume_unique** (*bool. Default: False*) – Whether to assume that the elements in both arrays are unique, which can speed up the calculation.

**Returns** **intersect_array** (`ndarray` object) – Array containing the unique elements found both in *array1* and *array2* over given *axis*.

### Example

```
>>> array1 = np.array([[1, 2], [1, 3], [2, 1]])
>>> array2 = np.array([[1, 2], [1, 3]])
>>> intersect(array1, array2)
array([[1, 2], [1, 3]])
```

e13tools.numpy.**isin**(*array1*, *array2*, *axis=0*, *assume_unique=False*, *invert=False*)

Checks over the provided *axis* which elements of given *array1* are also in given *array2* and returns it.

This is an nD-version of NumPy's `isin()` function.

**Parameters**

- **array1** (*array_like*) – Input array.
- **array2** (*array_like*) – Comparison array with same shape as *array1* except in given *axis*.

**Other Parameters**

- **axis** (*int or None. Default: 0*) – Axis over which elements must be checked in both arrays. A negative value counts from the last to the first axis. If *None*, both arrays are compared element-wise (this is the functionality of `isin()`).
- **assume_unique** (*bool. Default: False*) – Whether to assume that the elements in both arrays are unique, which can speed up the calculation.
- **invert** (*bool. Default: False*) – Whether to invert the returned boolean values. If *True*, the values in *bool_array* are as if calculating `array1 not in array2`.

**Returns** **bool_array** (`ndarray` object of bool) – Bool array containing the elements found in *array1* that are in *array2* over given *axis*.

### Example

```
>>> array1 = np.array([[1, 2], [1, 3], [2, 1]])
>>> array2 = np.array([[1, 2], [1, 3]])
>>> isin(array1, array2)
array([True, True, False])
```

e13tools.numpy.**rot90**(*array*, *axes=(0, 1)*, *rot_axis='center'*, *n_rot=1*)

Rotates the given *array* by 90 degrees around the point *rot_axis* in the given *axes*. This function is different from NumPy's `rot90()` function in that every column (2nd axis) defines a different dimension instead of every individual axis.

**Parameters** **array** (*2D array_like*) – Array with shape [*n_pts*, *n_dim*] with *n_pts* the number of points and *n_dim* the number of dimensions. Requires: *n_dim* > 1.

**Other Parameters**

- **axes** (*1D array_like with 2 ints. Default: (0, 1)*) – Array containing the axes defining the rotation plane. Rotation is from the first axis towards the second. Can be omitted if *rot_axis* has length *n_dim*.
- **rot_axis** (1D array_like of length 2/*n_dim* or 'center'. Default: 'center') – If 'center', the rotation axis is chosen in the center of the minimum and maximum values found in the given *axes*. If 1D array of length 2, the rotation axis is chosen around the given values in the given *axes*. If 1D array of length *n_dim*, the rotation axis is chosen around the first two non-zero values.
- **n_rot** (*int. Default: 1*) – Number of times to rotate *array* by 90 degrees.

**Returns** **array_rot** (2D `ndarray` object) – Array with shape [*n_pts*, *n_dim*] that has been rotated by 90 degrees *n_rot* times.

### Examples

Using an array with just two dimensions:

```
>>> array = np.array([[0.75, 0], [0.25, 1], [1, 0.75], [0, 0.25]])
>>> rot90(array)
array([[ 1.  ,  0.75],
       [ 0.  ,  0.25],
       [ 0.25,  1.  ],
       [ 0.75,  0.  ]])
```

Using the same array, but rotating it around a different point:

```
>>> array = np.array([[0.75, 0], [0.25, 1], [1, 0.75], [0, 0.25]])
>>> rot90(array, rot_axis=[0.2, 0.7])
array([[ 0.9 ,  1.25],
       [-0.1 ,  0.75],
       [ 0.15,  1.5 ],
       [ 0.65,  0.5 ]])
```

`e13tools.numpy.`**`setdiff`**(*array1*, *array2*, *axis=0*, *assume_unique=False*)

Finds the set difference between given arrays *array1* and *array2* over provided *axis* and returns the unique elements in *array1* that are not in *array2*.

This is an nD-version of NumPy's `setdiff1d()` function.

**Parameters**

- **array1** (*array_like*) – Input array.
- **array2** (*array_like*) – Comparison array with same shape as *array1* except in given *axis*.

**Other Parameters**

- **axis** (*int or None. Default: 0*) – Axis over which elements must be checked in both arrays. A negative value counts from the last to the first axis. If *None*, both arrays are flattened first (this is the functionality of `setdiff1d()`).
- **assume_unique** (*bool. Default: False*) – Whether to assume that the elements in both arrays are unique, which can speed up the calculation.

**Returns** **diff_array** (`ndarray` object) – Array containing the unique elements found in *array1* but not in *array2* over given *axis*.

### Example

```
>>> array1 = np.array([[1, 2], [1, 3], [2, 1]])
>>> array2 = np.array([[1, 2], [1, 3]])
>>> setdiff(array1, array2)
array([[2, 1]])
```

e13tools.numpy.**setxor**(*array1*, *array2*, *axis=0*, *assume_unique=False*)

Finds the set exclusive-or between given arrays *array1* and *array2* over provided *axis* and returns the unique elements that are in either *array1* or *array2* (but not both).

This is an nD-version of NumPy's `setxor1d()` function.

**Parameters**

- **array1** (*array_like*) – Input array.
- **array2** (*array_like*) – Comparison array with same shape as *array1* except in given *axis*.

**Other Parameters**

- **axis** (*int or None. Default: 0*) – Axis over which elements must be checked in both arrays. A negative value counts from the last to the first axis. If *None*, both arrays are flattened first (this is the functionality of `setxor1d()`).
- **assume_unique** (*bool. Default: False*) – Whether to assume that the elements in both arrays are unique, which can speed up the calculation.

**Returns** **xor_array** (`ndarray` object) – Array containing the unique elements found in either *array1* or *array2* (but not both) over given *axis*.

### Example

```
>>> array1 = np.array([[1, 2], [1, 3], [2, 1]])
>>> array2 = np.array([[1, 2], [1, 3], [3, 1]])
>>> setxor(array1, array2)
array([[2, 1], [3, 1]])
```

e13tools.numpy.**sort2D**(*array*, *axis=-1*, *order=None*)

Sorts a 2D *array* over a given *axis* in the specified *order*. This function is different from NumPy's `sort()` function in that it sorts over a given axis rather than along it, and the order can be given as integers rather than field strings.

**Parameters** **array** (*2D array_like*) – Input array that requires sorting.

**Other Parameters**

- **axis** (*int. Default: -1*) – Axis over which to sort the elements. Default is to sort all elements over the last axis. A negative value counts from the last to the first axis.
- **order** (*int, 1D array_like of int or None. Default: None*) – The order in which the vectors in the given *axis* need to be sorted. Negative values count from the last to the first vector. If *None*, all vectors in the given *axis* are sorted individually.

**Returns** **array_sort** (2D `ndarray` object) – Input *array* with its *axis* sorted in the specified *order*.

### Examples

Sorting the column elements of a given 2D array with no order specified:

```
>>> array = np.array([[0, 5, 1], [7, 4, 9], [3, 13, 6], [0, 1, 8]])
>>> array
array([[ 0,  5,  1],
       [ 7,  4,  9],
       [ 3, 13,  6],
       [ 0,  1,  8]])
>>> sort2D(array)
array([[ 0,  1,  1],
       [ 0,  4,  6],
       [ 3,  5,  8],
       [ 7, 13,  9]])
```

Sorting the same array in only the first column:

```
>>> sort2D(array, order=0)
array([[ 0,  5,  1],
       [ 0,  1,  8],
       [ 3, 13,  6],
       [ 7,  4,  9]])
```

Sorting all three columns in order:

```
>>> sort2D(array, order=(0, 1, 2))
array([[ 0,  1,  8],
       [ 0,  5,  1],
       [ 3, 13,  6],
       [ 7,  4,  9]])
```

Sorting all three columns in a different order:

```
>>> sort2D(array, order=(0, 2, 1))
array([[ 0,  5,  1],
       [ 0,  1,  8],
       [ 3, 13,  6],
       [ 7,  4,  9]])
```

e13tools.numpy.**transposeC**(*array*, *axes=None*)

Returns the (conjugate) transpose of the input *array*.

**Parameters** **array** (*array_like*) – Input array that needs to be transposed.

**Other Parameters** **axes** (*1D array_like of int or None. Default: None*) – If *None*, reverse the dimensions. Else, permute the axes according to the values given.

**Returns** **array_t** (ndarray object) – Input *array* with its axes transposed.

### Examples

Using an array with only real values returns its transposed variant:

```
>>> array = np.array([[1, 2.5], [3.5, 5]])
>>> array
array([[ 1. ,  2.5],
```

(continues on next page)

(continued from previous page)

```
       [ 3.5,  5. ]])
>>> transposeC(array)
array([[ 1. ,  3.5],
       [ 2.5,  5. ]])
```

And using an array containing complex values returns its conjugate transposed:

```
>>> array = np.array([[1, -2+4j], [7.5j, 0]])
>>> array
array([[ 1.+0.j , -2.+4.j ],
       [ 0.+7.5j,  0.+0.j ]])
>>> transposeC(array)
array([[ 1.-0.j ,  0.-7.5j],
       [-2.-4.j ,  0.-0.j ]])
```

e13tools.numpy.**union**(*array1*, *array2*, *axis=0*)

Finds the union between given arrays *array1* and *array2* over provided *axis* and returns the unique elements in *array1* and *array2*.

This is an nD-version of NumPy's `union1d()` function.

**Parameters**

- **array1** (*array_like*) – Input array.
- **array2** (*array_like*) – Comparison array with same shape as *array1* except in given *axis*.

**Other Parameters** **axis** (*int or None. Default: 0*) – Axis over which elements must be checked in both arrays. A negative value counts from the last to the first axis. If *None*, both arrays are flattened first (this is the functionality of `union1d()`).

**Returns** **union_array** (`ndarray` object) – Sorted array containing the unique elements found in *array1* and *array2* over given *axis*.

### Example

```
>>> array1 = np.array([[1, 2], [1, 3], [3, 1]])
>>> array2 = np.array([[1, 2], [1, 3], [2, 1]])
>>> union(array1, array2)
array([[1, 2], [1, 3], [2, 1], [3, 1]])
```

# CHAPTER 6

# PyPlot

Provides a collection of functions useful in various plotting routines.

e13tools.pyplot.**apu2tex**(*unit*, *unitfrac=False*)

Transform a `Unit` object into a (La)TeX string for usage in a `Figure` instance.

**Parameters** **unit** (`Unit` object) – Unit to be transformed.

**Other Parameters** **unitfrac** (*bool. Default: False*) – Whether or not to write *unit* as a LaTeX fraction.

**Returns** **out** (*string*) – String containing *unit* written in (La)TeX string.

### Examples

```
>>> import astropy.units as apu
>>> apu2tex(apu.solMass)
'\mathrm{M_{\odot}}'
```

```
>>> import astropy.units as apu
>>> apu2tex(apu.solMass/apu.yr, unitfrac=False)
'\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> import astropy.units as apu
>>> apu2tex(apu.solMass/apu.yr, unitfrac=True)
'\mathrm{\frac{M_{\odot}}{yr}}'
```

e13tools.pyplot.**center_spines**(*centerx=0*, *centery=0*, *set_xticker=False*, *set_yticker=False*, *ax=None*)

Centers the axis spines at <*centerx*, *centery*> on the axis *ax* in a `Figure` instance. Centers the axis spines at the origin by default.

**Other Parameters**

- **centerx** (*int or float. Default: 0*) – Centers x-axis at value *centerx*.

- **centery** (*int or float. Default: 0*) – Centers y-axis at value *centery*.
- **set_xticker** (*int, float or False. Default: False*) – If int or float, sets the x-axis ticker to *set_xticker*. If *False*, let `Figure` instance decide.
- **set_yticker** (*int, float or False. Default: False*) – If int or float, sets the y-axis ticker to *set_yticker*. If *False*, let `Figure` instance decide.
- **ax** (`Axes` object or None. Default: None) – If `Axes` object, centers the axis spines of specified `Figure` instance. If *None*, centers the axis spines of current `Figure` instance.

e13tools.pyplot.**draw_textline**(*text*, ***, *x=None*, *y=None*, *pos='start top'*, *ax=None*, *line_kwargs={}*, *text_kwargs={}*)

Draws a line on the axis *ax* in a `Figure` instance and prints *text* on top.

**Parameters**

- **text** (*str*) – Text to be printed on the line.
- **x** (*scalar or None*) – If scalar, text/line x-coordinate. If *None*, line covers complete x-axis. Either *x* or *y* needs to be *None*.
- **y** (*scalar or None*) – If scalar, text/line y-coordinate. If *None*, line covers complete y-axis. Either *x* or *y* needs to be *None*.

**Other Parameters**

- **pos** (*{'start', 'end'}{'top', 'bottom'}. Default: 'start top'*) – If 'start', prints the text at the start of the drawn line. If 'end', prints the text at the end of the drawn line. If 'top', prints the text above the drawn line. If 'bottom', prints the text below the drawn line. Arguments must be given as a single string.
- **ax** (`Axes` object or None. Default: None) – If `Axes` object, draws line in specified `Figure` instance. If *None*, draws line in current `Figure` instance.
- **line_kwargs** (dict of `Line2D` properties. Default: {}) – The keyword arguments used for drawing the line.
- **text_kwargs** (dict of `Text` properties. Default: {}) – The keyword arguments used for drawing the text.

e13tools.pyplot.**f2tex**(*value*, **errs*, *sdigits=4*, *power=3*, *nobase1=True*)

Transform a value into a (La)TeX string for usage in a `Figure` instance.

**Parameters** **value** (*int or float*) – Value to be transformed.

**Other Parameters**

- **errs** (*int or float*) – The upper and lower $1\sigma$-errors of the given *value*. If only a single value is given, *value* is assumed to have a centered error interval of *errs*.
- **sdigits** (*int. Default: 4*) – Number of significant digits any value is returned with.
- **power** (*int. Default: 3*) – Minimum abs(log10(*value*)) required before all values are written in scientific form. This value is ignored if *sdigits* forces scientific form to (not) be used.
- **nobase1** (*bool. Default: True*) – Whether or not to include *base* in scientific form if *base=1*. This is always *False* if *errs* contains at least one value.

**Returns** **out** (*string*) – String containing *value* and *errs* written in (La)TeX string.

### Examples

```
>>> f2tex(20.2935826592)
'20.29'
```

```
>>> f2tex(20.2935826592, sdigits=6)
'20.2936'
```

```
>>> f2tex(20.2935826592, power=1)
'2.029\cdot 10^{1}'
```

```
>>> f2tex(1e6, nobase1=True)
'10^{6}'
```

```
>>> f2tex(1e6, nobase1=False)
'1.000\cdot 10^{6}'
```

```
>>> f2tex(20.2935826592, 0.1)
'20.29\pm 0.10'
```

```
>>> f2tex(20.2935826592, 0.1, 0.2)
'20.29^{+0.10}_{-0.20}'
```

```
>>> f2tex(1e6, 12, 10)
'1.000^{+0.000}_{-0.000}\cdot 10^{6}'
```

```
>>> f2tex(1e6, 12, 10, sdigits=6)
'1.000^{+0.000}_{-0.000}\cdot 10^{6}'
```

e13tools.pyplot.**q2tex**(*quantity*, **errs*, *sdigits=4*, *power=3*, *nobase1=True*, *unitfrac=False*)

Combination of *f2tex()* and *apu2tex()*.

Transform a Quantity object into a (La)TeX string for usage in a Figure instance.

**Parameters** **quantity** (int, float or Quantity object) – Quantity to be transformed.

**Other Parameters**

- **errs** (int, float or Quantity object) – The upper and lower $1\sigma$-errors of the given *quantity*. If only a single value is given, *quantity* is assumed to have a centered error interval of *errs*. The unit of *errs* must be convertible to the unit of *quantity*.
- **sdigits** (*int. Default: 4*) – Maximum amount of significant digits any quantity is returned with.
- **power** (*int. Default: 3*) – Minimum abs(log10(*value*)) required before all quantities are written in scientific form. This value is ignored if *sdigits* forces scientific form to (not) be used.
- **nobase1** (*bool. Default: True*) – Whether or not to include *base* in scientific form if *base=1*. This is always *False* if *errs* contains a value.
- **unitfrac** (*bool. Default: False*) – Whether or not to write *unit* as a LaTeX fraction.

**Returns** **out** (*string*) – String containing *quantity* and *errs* written in (La)TeX string.

### Examples

```
>>> import astropy.units as apu
>>> q2tex(20.2935826592)
'20.29'
```

```
>>> q2tex(20.2935826592*apu.kg, 1500*apu.g)
'20.29\pm 1.50\,\mathrm{kg}'
```

```
>>> q2tex(20.2935826592*apu.solMass/apu.yr)
'20.29\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(20.2935826592*apu.solMass/apu.yr, sdigits=6)
'20.2936\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(20.2935826592*apu.solMass/apu.yr, power=1)
'2.029\cdot 10^{1}\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(1e6*apu.solMass/apu.yr, nobase1=True)
'10^{6}\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(1e6*apu.solMass/apu.yr, nobase1=False)
'1.000\cdot 10^{6}\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(20.2935826592*apu.solMass/apu.yr, unitfrac=False)
'20.29\,\mathrm{M_{\odot}\,yr^{-1}}'
```

```
>>> q2tex(20.2935826592*apu.solMass, 1*apu.solMass, unitfrac=True)
'20.29\pm 1.00\,\mathrm{M_{\odot}}'
```

# CHAPTER 7

# Sampling

e13tools.sampling.**lhd**(*n_sam*, *n_val*, *val_rng=None*, *method='random'*, *criterion=None*, *iterations=1000*, *get_score=False*, *quickscan=True*, *constraints=None*)

Generates a Latin Hypercube Design of *n_sam* samples, each with *n_val* values. Method for choosing the 'best' Latin Hypercube Design depends on the *method* and *criterion* that are used.

**Parameters**

- **n_sam** (*int*) – The number of samples to generate.
- **n_val** (*int*) – The number of values in a single sample.

**Other Parameters**

- **val_rng** (*2D array_like or None. Default: None*) – Array defining the lower and upper limits of every value in a sample. Requires: numpy.shape(val_rng) = (*n_val*, 2). If *None*, output is normalized.
- **method** (*{'random'; 'fixed'; 'center'}. Default: 'random'*) – String specifying the method used to construct the Latin Hypercube Design. See `Notes` for more details. If *n_sam* == 1 or *n_val* == 1, *method* is set to the closest corresponding method if necessary.
- **criterion** (*float, {'maximin'; 'correlation'; 'multi'} or None. Default: None*) – Float or string specifying the criterion the Latin Hypercube Design has to satisfy or *None* for no criterion. See `Notes` for more details. If *n_sam* == 1 or *n_val* == 1, *criterion* is set to the closest corresponding criterion if necessary.
- **iterations** (*int. Default: 1000*) – Number of iterations used for the criterion algorithm.
- **get_score** (*bool. Default: False*) – If *True*, the normalized maximin, correlation and multi scores are also returned if a criterion is used.
- **quickscan** (*bool. Default: True*) – If *True*, a faster but less precise algorithm will be used for the criteria.
- **constraints** (*2D array_like or None. Default: None*) – If *constraints* is not empty and *criterion* is not *None*, *sam_set* + *constraints* will satisfy the given criterion instead of *sam_set*. Providing this argument when *criterion* is *None* will discard it. **WARNING**: If *constraints* is not a 'fixed' or 'center' lay-out LHD, the output might contain errors.

**Returns** **sam_set** (2D `ndarray` object) – Sample set array of shape [*n_sam*, *n_val*].

**Notes**

The 'method' argument specifies the way in which the values should be distributed within the value intervals.

The following methods can be used:

| method | interval lay-out |
|---|---|
| 'random' | Values are randomized |
| 'fixed' | Values are fixed to maximize spread |
| 'center' | Values are centered |
| 'r' | Same as 'random' |
| 'f' | Same as 'fixed' |
| 'c' | Same as 'center' |

The 'fixed' method chooses values in such a way, that the distance between the values is maxed out.

The 'criterion' argument specifies how much priority should be given to maximizing the minimum distance and minimizing the correlation between samples. Strings specify basic priority cases, while a value between 0 and 1 specifies a custom case.

The following criteria can be used (last column shows the equivalent float value):

| criterion | effect/priority | equiv |
|---|---|---|
| None | No priority | – |
| 'maximin' | Maximum priority for maximizing the minimum distance | 0.0 |
| 'correlation' | Maximum priority for minimizing the correlation | 1.0 |
| 'multi' | Equal priority for both | 0.5 |
| [0, 1] | Priority is given according to value provided | – |

## Examples

Latin Hypercube with 5 samples with each 2 random, fixed or centered values:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> lhd(5, 2, method='random')
array([[ 0.34303787,  0.55834501],
       [ 0.70897664,  0.70577898],
       [ 0.88473096,  0.19273255],
       [ 0.1097627 ,  0.91360891],
       [ 0.52055268,  0.2766883 ]])
>>> lhd(5, 2, method='fixed')
array([[ 0.5 ,  0.75],
       [ 0.25,  0.25],
       [ 0.  ,  1.  ],
       [ 0.75,  0.5 ],
       [ 1.  ,  0.  ]])
>>> lhd(5, 2, method='center')
array([[ 0.1,  0.9],
       [ 0.9,  0.5],
       [ 0.5,  0.7],
```

(continues on next page)

(continued from previous page)

```
       [ 0.3,  0.3],
       [ 0.7,  0.1]])
```

Latin Hypercube with 4 samples, 3 values in a specified value range:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> val_rng = [[0, 2], [1, 4], [0.3, 0.5]]
>>> lhd(4, 3, val_rng=val_rng)
array([[ 1.30138169,  2.41882975,  0.41686981],
       [ 0.27440675,  1.32819041,  0.48240859],
       [ 1.77244159,  3.53758114,  0.39180394],
       [ 0.85759468,  3.22274707,  0.31963924]])
```

Latin Hypercubes can also be created by specifying a criterion with either a string or a normalized float. The strings identify basic float values.

```
>>> import numpy as np
>>> np.random.seed(0)
>>> lhd(4, 3, method='fixed', criterion=0)
array([[ 0.66666667,  0.        ,  0.66666667],
       [ 1.        ,  0.66666667,  0.        ],
       [ 0.33333333,  1.        ,  1.        ],
       [ 0.        ,  0.33333333,  0.33333333]])
>>> np.random.seed(0)
>>> lhd(4, 3, method='fixed', criterion='maximin')
array([[ 0.66666667,  0.        ,  0.66666667],
       [ 1.        ,  0.66666667,  0.        ],
       [ 0.33333333,  1.        ,  1.        ],
       [ 0.        ,  0.33333333,  0.33333333]])
```

CHAPTER 8

# Utilities

Provides several useful utility functions.

e13tools.utils.**add_to_all**(*obj*)
Custom decorator that allows for the name of the provided object *obj* to be automatically added to the *__all__* attribute of the frame this decorator is used in. The provided *obj* must have a *__name__* attribute.

e13tools.utils.**check_instance**(*instance*, *cls*)
Checks if provided *instance* has been initialized from a proper *cls* (sub)class. Raises a `TypeError` if *instance* is not an instance of *cls*.

**Parameters**

- **instance** (*object*) – Class instance that needs to be checked.
- **cls** (*class*) – The class which *instance* needs to be properly initialized from.

**Returns** **result** (*bool*) – Bool indicating whether or not the provided *instance* was initialized from a proper *cls* (sub)class.

e13tools.utils.**delist**(*list_obj*)
Returns a copy of *list_obj* with all empty lists and tuples removed.

**Parameters** **list_obj** (*list*) – A list object that requires its empty list/tuple elements to be removed.

**Returns** **delisted_copy** (*list*) – Copy of *list_obj* with all empty lists/tuples removed.

e13tools.utils.**docstring_append**(*addendum*, *join=''*)
Custom decorator that allows a given string *addendum* to be appended to the docstring of the target function/class, separated by a given string *join*.

If *addendum* is not a string, its `__doc__` attribute is used instead.

e13tools.utils.**docstring_copy**(*source*)
Custom decorator that allows the docstring of a function/class *source* to be copied to the target function/class.

e13tools.utils.**docstring_substitute**(**args*, ***kwargs*)
Custom decorator that allows either given positional arguments *args* or keyword arguments *kwargs* to be substituted into the docstring of the target function/class.

Both % and *.format()* string formatting styles are supported. Keep in mind that this decorator will always attempt to do %-formatting first, and only uses *.format()* if the first fails.

e13tools.utils.**get_main_desc**(*source*)
Retrieves the main description of the provided object *source* and returns it.

The main description is defined as the first paragraph of its docstring.

**Parameters** **source** (*object*) – The object whose main description must be retrieved.

**Returns** **main_desc** (*str or None*) – The main description string of the provided *source* or *None* if *source* has not docstring.

e13tools.utils.**get_outer_frame**(*func*)
Checks whether or not the calling function contains an outer frame corresponding to *func* and returns it if so. If this frame cannot be found, returns *None* instead.

**Parameters** **func** (*function*) – The function or method whose frame must be located in the outer frames.

**Returns** **outer_frame** (*frame or None*) – The requested outer frame if it was found, or *None* if it was not.

e13tools.utils.**raise_error**(*err_msg*, *err_type=<class 'Exception'>*, *logger=None*, *err_traceback=None*)
Raises a given error *err_msg* of type *err_type* and logs the error using the provided *logger*.

**Parameters** **err_msg** (*str*) – The message included in the error.

**Other Parameters**

- **err_type** (`Exception` subclass. Default: `Exception`) – The type of error that needs to be raised.
- **logger** (`Logger` object or None. Default: None) – The logger to which the error message must be written. If *None*, the `RootLogger` logger is used instead.
- **err_traceback** (*traceback object or None. Default: None*) – The traceback object that must be used for this exception, useful for when this function is used for reraising a caught exception. If *None*, no additional traceback is used.

  New in version 0.6.17.

**See also:**

***raise_warning()*** Raises and logs a given warning.

e13tools.utils.**raise_warning**(*warn_msg*, *warn_type=<class 'UserWarning'>*, *logger=None*, *stacklevel=1*)
Raises/issues a given warning *warn_msg* of type *warn_type* and logs the warning using the provided *logger*.

**Parameters** **warn_msg** (*str*) – The message included in the warning.

**Other Parameters**

- **warn_type** (`Warning` subclass. Default: `UserWarning`) – The type of warning that needs to be raised/issued.
- **logger** (`Logger` object or None. Default: None) – The logger to which the warning message must be written. If *None*, the `RootLogger` logger is used instead.
- **stacklevel** (*int. Default: 1*) – The stack level of the warning message at the location of this function call. The actual used stack level is increased by one to account for this function call.

**See also:**

*raise_error()* Raises and logs a given error.

e13tools.utils.**split_seq**(**seq*)
Converts a provided sequence *seq* to a string, removes all auxiliary characters from it, splits it up into individual elements and converts all elements back to booleans; floats; integers; and/or strings.

The auxiliary characters are given by `aux_char_set`. One can add, change and remove characters from the set if required. If one wishes to keep an auxiliary character that is in *seq*, it must be escaped by a backslash (note that backslashes themselves also need to be escaped).

This function can be used to easily unpack a large sequence of nested iterables into a single list, or to convert a formatted string to a list of elements.

**Parameters** **seq** (*str, array_like or tuple of arguments*) – The sequence that needs to be split into individual elements. If array_like, *seq* is first unpacked into a string. It is possible for *seq* to be a nested iterable.

**Returns** **new_seq** (*list*) – A list with all individual elements converted to booleans; floats; integers; and/or strings.

### Examples

The following function calls all produce the same output:

```
>>> split_seq('A', 1, 20.0, 'B')
['A', 1, 20.0, 'B']
>>> split_seq(['A', 1, 2e1, 'B'])
['A', 1, 20.0, 'B']
>>> split_seq("A 1 20. B")
['A', 1, 20.0, 'B']
>>> split_seq([("A", 1), (["20."], "B")])
['A', 1, 20.0, 'B']
>>> split_seq("[(A / }| ; <1{}) , ,>20.0000 !! < )?% \B")
['A', 1, 20.0, 'B']
```

If one wants to keep the '?' in the last string above, it must be escaped:

```
>>> split_seq("[(A / }| ; <1{}) , ,>20.0000 !! < )\?% \B")
['A', 1, 20.0, '?', 'B']
```

**See also:**

*unpack_str_seq()* Unpacks a provided (nested) sequence into a single string.

e13tools.utils.**unpack_str_seq**(**seq*, *sep=', '*)
Unpacks a provided sequence *seq* of elements and iterables, and converts it to a single string separated by *sep*.

Use *split_seq()* if it is instead required to unpack *seq* into a single list while maintaining the types of all elements.

**Parameters** **seq** (*str, array_like or tuple of arguments*) – The sequence that needs to be unpacked into a single string. If *seq* contains nested iterables, this function is used recursively to unpack them as well.

**Other Parameters** **sep** (*str. Default: ', '*) – The string to use for separating the elements in the unpacked string.

**Returns** **unpacked_seq** (*str*) – A string containing all elements in *seq* unpacked and converted to a string, separated by *sep*.

### Examples

The following function calls all produce the same output:

```
>>> unpack_str_seq('A', 1, 20.0, 'B')
'A, 1, 20.0, B'
>>> unpack_str_seq(['A', 1, 2e1, 'B'])
'A, 1, 20.0, B'
>>> unpack_str_seq("A, 1, 20.0, B")
'A, 1, 20.0, B'
>>> unpack_str_seq([("A", 1), (["20.0"], "B")])
'A, 1, 20.0, B'
```

**See also:**

***split_seq()*** Splits up a provided (nested) sequence into a list of individual elements.

# Python Module Index

# Index

## Symbols



## A



## C



## D



## E



## F



## G



## I



## L



## N



## Q



## R



## S



## T



## U