# PyAbel Documentation

***Release 0.8.3***

**PyAbel team**

**Jan 28, 2020**

# Contents

Start by having a look at the *README*.

Contents:

CHAPTER 1

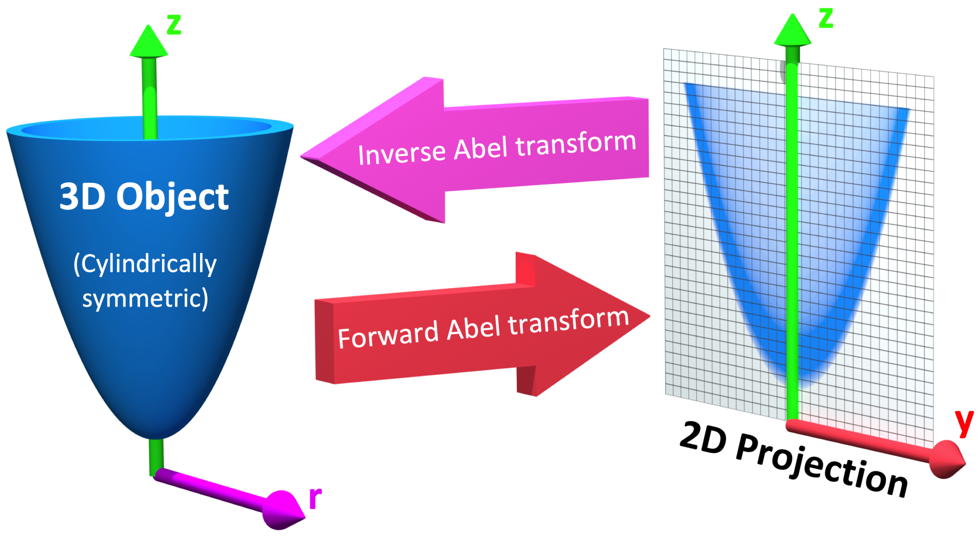
# PyAbel README

BUILD PASSING

**Note:** This readme is best viewed as part of the PyAbel Documentation.

## 1.1 Introduction

`PyAbel` is a Python package that provides functions for the forward and inverse Abel transforms. The forward Abel transform takes a slice of a cylindrically symmetric 3D object and provides the 2D projection of that object. The inverse Abel transform takes a 2D projection and reconstructs a slice of the cylindrically symmetric 3D distribution.



Inverse Abel transforms play an important role in analyzing the projections of angle-resolved photoelectron/photoion spectra, plasma plumes, flames, and solar occultation.

PyAbel provides efficient implementations of several Abel transform algorithms, as well as related tools for centering images, symmetrizing images, and calculating properties such as the radial intensity distribution and the anisotropy parameters.

## 1.2 Transform Methods

The outcome of the numerical Abel transform depends on the exact method used. So far, PyAbel includes the following transform methods:

1. `basex` - Gaussian basis set expansion of Dribinski and co-workers.
2. `hansenlaw` - recursive method of Hansen and Law.
3. `direct` - numerical integration of the analytical Abel transform equations.
4. `two_point` - the “two point” method of Dasch and co-workers.
5. `three_point` - the “three point” method of Dasch and co-workers.
6. `onion_peeling` - the “onion peeling” deconvolution method of Dasch and co-workers.
7. `onion_bordas` - “onion peeling” or “back projection” method of Bordas *et al.* based on the MatLab code by Rallis and Wells *et al.*
8. `linbasex` - the 1D-spherical basis set expansion of Gerber *et al.*

## 1.3 Installation

PyAbel requires Python 2.7 or 3.5-3.7. NumPy and SciPy are also required, and Matplotlib is required to run the examples. If you don’t already have Python, we recommend an “all in one” Python package such as the Anaconda Python Distribution, which is available for free.

### 1.3.1 With pip

The latest release can be installed from PyPi with

```
pip install PyAbel
```

### 1.3.2 With setuptools

If you prefer the development version from GitHub, download it here, *cd* to the PyAbel directory, and use

```
python setup.py install
```

Or, if you wish to edit the PyAbel source code without re-installing each time

```
python setup.py develop
```

## 1.4 Example of use

Using PyAbel can be simple. The following Python code imports the PyAbel package, generates a sample image, performs a forward transform using the Hansen–Law method, and then a reverse transform using the Three Point method:

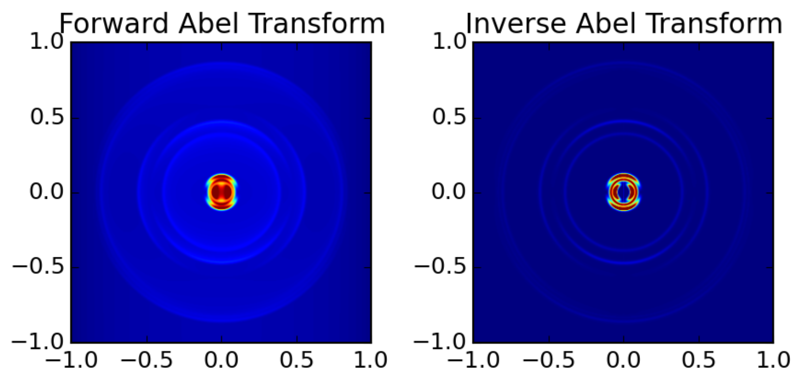
```
import abel
original     = abel.tools.analytical.SampleImage().image
forward_abel = abel.Transform(original, direction='forward', method='hansenlaw').
↪transform
inverse_abel = abel.Transform(forward_abel, direction='inverse', method='three_point
↪').transform
```

Note: the `abel.Transform()` class returns a Python `class` object, where the 2D Abel transform is accessed through the `.transform` attribute.

The results can then be plotted using Matplotlib:

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(1, 2, figsize=(6, 4))

axs[0].imshow(forward_abel, clim=(0, np.max(forward_abel)*0.6), origin='lower',␣
↪extent=(-1,1,-1,1))
axs[1].imshow(inverse_abel, clim=(0, np.max(inverse_abel)*0.4), origin='lower',␣
↪extent=(-1,1,-1,1))

axs[0].set_title('Forward Abel Transform')
axs[1].set_title('Inverse Abel Transform')

plt.tight_layout()
plt.show()
```

Output:



**Note:** Additional examples can be viewed on the PyAbel examples page and even more are found in the PyAbel/examples directory.

## 1.5 Documentation

General information about the various Abel transforms available in PyAbel is available at the links above. The complete documentation for all of the methods in PyAbel is hosted at https://pyabel.readthedocs.io.

## 1.6 Conventions

The PyAbel code adheres to the following conventions:

- **Image orientation:** PyAbel adopts the "television" convention, where `IM[0,0]` refers to the **upper** left corner of the image. (This means that `plt.imshow(IM)` should display the image in the proper orientation, without the need to use the `origin='lower'` keyword.) As an example, the x,y-grid for a 5x5 image can be generated using:

```
x = np.linspace(-2,2,5)
X,Y = np.meshgrid(x, -x) # notice the minus sign in front of the y-coordinate
```

- **Angle:** All angles in PyAbel are measured in radians. When an absolute angle is defined, zero-angle corresponds to the upwards, vertical direction. Positive values are on the right side, and negative values on the left side. The range of angles is from -Pi to +Pi. The polar grid for a 5x5 image can be generated (following the code above) using:

```
R = np.sqrt(X**2 + Y**2)
THETA = np.arctan2(X, Y)
```

  where the usual `(Y, X)` convention of `arctan2` has been reversed in order to place zero-angle in the vertical direction. Consequently, to convert the angular grid back to the Cartesian grid, we use:

```
X = R*np.sin(THETA)
Y = R*np.cos(THETA)
```

- **Image center:** Fundamentally, the Abel and inverse-Abel transforms in PyAbel consider the center of the image to be located in the center of a pixel. This means that, for a symmetric image, the image will have a width that is an odd number of pixels. (The center pixel is effectively "shared" between both halves of the image.) In most situations, the center is specified using the `center` keyword in `abel.Transform` (or directly using `abel.center.center_image` to find the true center of your image. This processing step takes care of locating the center of the image in the middle of the central pixel. However, if the individual Abel transforms methods are used directly, care must be taken to supply a properly centered image.

## 1.7 Support

If you have a question or suggestion about PyAbel, the best way to contact the PyAbel Developers Team is to open a new issue.

## 1.8 Contributing

We welcome suggestions for improvement, together with any interesting images that demonstrate application of PyAbel.

Either open a new Issue or make a Pull Request.

CONTRIBUTING.rst has more information on how to contribute, such as how to run the unit tests and how to build the documentation.

## 1.9 License

PyAble is licensed under the MIT license, so it can be used for pretty much whatever you want! Of course, it is provided "as is" with absolutely no warranty.

## 1.10 Citation

First and foremost, please cite the paper(s) corresponding to the implementation of the Abel transform that you use in your work. The references can be found at the links above.

If you find PyAbel useful in you work, it would bring us great joy if you would cite the project. You can find the DOI for the lastest verison here

Additionally, we have written a scientific paper comparing various Abel transform methods. You can find the manuscript at the Review of Scientific Instruments (DOI: doi.org/10.1063/1.5092635) or on arxiv (arxiv.org/abs/1902.09007).

**Have fun!**

CHAPTER 2

# abel package

## 2.1 abel.transform module

**class** abel.transform.**Transform**(*IM*, *direction=u'inverse'*, *method=u'three_point'*, *center=u'none'*, *symmetry_axis=None*, *use_quadrants=(True, True, True, True)*, *symmetrize_method=u'average'*, *angular_integration=False*, *transform_options={}*, *center_options={}*, *angular_integration_options={}*, *recast_as_float64=True*, *verbose=False*)

Bases: `object`

Abel transform image class.

This class provides whole image forward and inverse Abel transformations, together with preprocessing (centering, symmetrizing) and post processing (integration) functions.

The following class attributes are available, depending on the calculation.

**Returns**

- **transform** (*numpy 2D array*) – the 2D forward/reverse Abel transform.
- **angular_integration** (*tuple*) – (radial-grid, radial-intensity) radial coordinates, and the radial intensity (speed) distribution, evaluated using `abel.tools.vmi.angular_integration()`.
- **residual** (*numpy 2D array*) – residual image (not currently implemented).
- **IM** (*numpy 2D array*) – the input image, re-centered (optional) with an odd-size width.
- **method** (*str*) – transform method, as specified by the input option.
- **direction** (*str*) – transform direction, as specified by the input option.
- **Beta** (*numpy 2D array*) – with `linbasex` `transform_options=dict(return_Beta=True)()` Beta array coefficients of Newton sphere spherical harmonics

  Beta[0] - the radial intensity variation

Beta[1] - the anisotropy parameter variation

. . . Beta[n] - higher order terms up to *legedre_orders = [0, . . . , n]*

- **radial** (*numpy 1d array*) – with `linbasex transform_options=dict(return_Beta=True)()` radial-grid for Beta array
- *projection* – with `linbasex transform_options=dict(return_Beta=True)()` radial projection profiles at angles *proj_angles*

**`__init__`**(*IM*, *direction=u'inverse'*, *method=u'three_point'*, *center=u'none'*, *symmetry_axis=None*, *use_quadrants=(True*, *True*, *True*, *True)*, *symmetrize_method=u'average'*, *angular_integration=False*, *transform_options={}*, *center_options={}*, *angular_integration_options={}*, *recast_as_float64=True*, *verbose=False*)

The one stop transform function.

**Parameters**

- **IM** (*a NxM numpy array*) – This is the image to be transformed
- **direction** (*str*) – The type of Abel transform to be performed.

  **`forward`** A 'forward' Abel transform takes a (2D) slice of a 3D image and returns the 2D projection.

  **`inverse`** An 'inverse' Abel transform takes a 2D projection and reconstructs a 2D slice of the 3D image.

  The default is `inverse`.

- **method** (*str*) – specifies which numerical approximation to the Abel transform should be employed (see below). The options are

  **`hansenlaw`** the recursive algorithm described by Hansen and Law.

  **`basex`** the Gaussian "basis set expansion" method of Dribinski et al.

  **`direct`** a naive implementation of the analytical formula by Roman Yurchuk.

  **`two_point`** the two-point transform of Dasch (1992).

  **`three_point`** the three-point transform of Dasch (1992).

  **`onion_bordas`** the algorithm of Bordas and co-workers (1996), re-implemented by Rallis, Wells and co-workers (2014).

  **`onion_peeling`** the onion peeling deconvolution as described by Dasch (1992).

  **`linbasex`** the 1d-projections of VM-images in terms of 1d spherical functions by Gerber et al. (2013).

- **center** (*tuple or str*) – If a tuple (float, float) is provided, this specifies the image center in (y,x) (row, column) format. A value *None* can be supplied if no centering is desired in one dimension, for example 'center=(None, 250)'. If a string is provided, an automatic centering algorithm is used

  **`image_center`** center is assumed to be the center of the image.

  **`convolution`** center the image by convolution of two projections along each axis.

  **`slice`** the center is found my comparing slices in the horizontal and vertical directions

  **`com`** the center is calculated as the center of mass

  **`gaussian`** the center is found using a fit to a Gaussian function. This only makes sense if your data looks like a Gaussian.

**none** (Default) No centering is performed. An image with an odd number of columns must be provided.

- **symmetry_axis** (*None, int or tuple*) – Symmetrize the image about the numpy axis 0 (vertical), 1 (horizontal), (0,1) (both axes)
- **use_quadrants** (*tuple of 4 booleans*) – select quadrants to be used in the analysis: (Q0,Q1,Q2,Q3). Quadrants are numbered counter-clockwide from upper right. See note below for description of quadrants. Default is `(True, True, True, True)`, which uses all quadrants.
- **symmetrize_method** (*str*) – Method used for symmetrizing the image.

  **average** average the quadrants, in accordance with the *symmetry_axis*

  **fourier** axial symmetry implies that the Fourier components of the 2-D projection should be real. Removing the imaginary components in reciprocal space leaves a symmetric projection. ref: Overstreet, K., et al. "Multiple scattering and the density distribution of a Cs MOT." Optics express 13.24 (2005): 9672-9682. http://dx.doi.org/10.1364/OPEX.13.009672
- **angular_integration** (*boolean*) – integrate the image over angle to give the radial (speed) intensity distribution
- **transform_options** (*tuple*) – Additional arguments passed to the individual transform functions. See the documentation for the individual transform method for options.
- **center_options** (*tuple*) – Additional arguments to be passed to the centering function.
- **angular_integration_options** (*tuple (or dict)*) – Additional arguments passed to the angular_integration transform functions. See the documentation for angular_integration for options.
- **recast_as_float64** (*boolean*) – True/False that determines if the input image should be recast to `float64`. Many images are imported in other formats (such as `uint8` or `uint16`) and this does not always play well with the transorm algorithms. This should probably always be set to True. (Default is True.)
- **verbose** (*boolean*) – True/False to determine if non-critical output should be printed.

**Note:**

**Quadrant combining** The quadrants can be combined (averaged) using the `use_quadrants` keyword in order to provide better data quality.

The quadrants are numbered starting from Q0 in the upper right and proceeding counter-clockwise:

```
+--------+--------+
| Q1   * | *   Q0 |
|   *    |    *   |
|  *     |     *  |                               AQ1 | AQ0
+--------o--------+ --(inverse Abel transform)--> ----o----
|  *     |     *  |                               AQ2 | AQ3
|   *    |    *   |
| Q2  *  | *   Q3 |          AQi == inverse Abel transform
+--------+--------+                 of quadrant Qi
```

Three cases are possible:

1) symmetry_axis = 0 (vertical):

```
Combine:  Q01 = Q0 + Q1, Q23 = Q2 + Q3
inverse image   AQ01 | AQ01
                -----o----- (left and right sides equivalent)
                AQ23 | AQ23
```

2) symmetry_axis = 1 (horizontal):

```
Combine: Q12 = Q1 + Q2, Q03 = Q0 + Q3
inverse image   AQ12 | AQ03
                -----o----- (top and bottom equivalent)
                AQ12 | AQ03
```

3) symmetry_axis = (0, 1) (both):

```
Combine: Q = Q0 + Q1 + Q2 + Q3
inverse image   AQ | AQ
                ---o---  (all quadrants equivalent)
                AQ | AQ
```

## Notes

As mentioned above, PyAbel offers several different approximations to the the exact abel transform. All the the methods should produce similar results, but depending on the level and type of noise found in the image, certain methods may perform better than others. Please see the "Transform Methods" section of the documentation for complete information.

**hansenlaw** This "recursive algorithm" produces reliable results and is quite fast (~0.1 sec for a 1001x1001 image). It makes no assumptions about the data (apart from cylindrical symmetry). It tends to require that the data is finely sampled for good convergence.

E. W. Hansen and P.-L. Law "Recursive methods for computing the Abel transform and its inverse" J. Opt. Soc. A*2, 510-520 (1985) http://dx.doi.org/10.1364/JOSAA.2.000510

**basex** * The "basis set exapansion" algorithm describes the data in terms of gaussian functions, which themselves can be abel transformed analytically. Because the gaussian functions are approximately the size of each pixel, this method also does not make any assumption about the shape of the data. This method is one of the de-facto standards in photoelectron/photoion imaging.

Dribinski et al, 2002 (Rev. Sci. Instrum. 73, 2634) http://dx.doi.org/10.1063/1.1482156

**direct** This method attempts a direct integration of the Abel transform integral. It makes no assumptions about the data (apart from cylindrical symmetry), but it typically requires fine sampling to converge. Such methods are typically inefficient, but thanks to this Cython implementation (by Roman Yurchuk), this 'direct' method is competitive with the other methods.

**linbasex** * VM-images are composed of projected Newton spheres with a common centre. The 2D images are usually evaluated by a decomposition into base vectors each representing the 2D projection of a set of particles starting from a centre with a specific velocity distribution. *linbasex* evaluate 1D projections of VM-images in terms of 1D projections of spherical functions, instead.

..Rev. Sci. Instrum. 84, 033101 (2013): <http://scitation.aip.org/content/aip/journal/rsi/84/3/10.1063/1.4793404>

**onion_bordas**

The onion peeling method, also known as "back projection", originates from Bordas *et al.* Rev. Sci. Instrum. 67, 2257 (1996).

The algorithm was subsequently coded in MatLab by Rallis, Wells and co-workers, Rev. Sci. Instrum. 85, 113105 (2014).

which was used as the basis of this Python port. See issue #56.

**`onion_peeling`** *

This is one of the most compact and fast algorithms, with the inverse Abel transfrom achieved in one Python code-line, PR #155. See also `three_point` is the onion peeling algorithm as described by Dasch (1992), reference below.

**`two_point`** * Another Dasch method. Simple, and fast, but not as accurate as the other methods.

**`three_point`** * The "Three Point" Abel transform method exploits the observation that the value of the Abel inverted data at any radial position r is primarily determined from changes in the projection data in the neighborhood of r. This method is also very efficient once it has generated the basis sets.

Dasch, 1992 (Applied Optics, Vol 31, No 8, March 1992, Pg 1146-1152).

* The methods marked with a * indicate methods that generate basis sets. The first time they are run for a new image size, it takes seconds to minutes to generate the basis set. However, this basis set is saved to disk can can be reloaded, meaning that future transforms are performed much more quickly.

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

## 2.2 abel.basex module

`abel.basex.`**`basex_transform`**(*data*, *sigma=1.0*, *reg=0.0*, *correction=True*, *basis_dir=u'./'*, *dr=1.0*, *verbose=True*, *direction=u'inverse'*)

This function performs the BASEX (BAsis Set EXpansion) Abel transform. It works on a "right side" image. I.e., it works on just half of a cylindrically symmetric object, and `data[0,0]` should correspond to a central pixel. To perform a BASEX transform on a whole image, use

```
abel.Transform(image, method='basex', direction='inverse').transform
```

This BASEX implementation only works with images that have an odd-integer full width.

**Parameters**

- **data** (*m × n numpy array*) – the image to be transformed. `data[:,0]` should correspond to the central column of the image.
- **sigma** (*float*) – width parameter for basis functions, see equation (14) in the article. Determines the number of basis functions (**n/sigma** rounded). Can be any positive number, but using **sigma** < 1 is not very meaningful and requires regularization.
- **reg** (*float*) –

  regularization parameter, square of the Tikhonov factor.

  `reg=0` means no regularization,

  `reg=100` is a reasonable value for megapixel images.

  Forward transform requires regularization only if **sigma** < 1, and **reg** should be  1.
- **correction** (*boolean*) – apply intensity correction in order to reduce method artifacts (intensity normalization and oscillations)

- **basis_dir** (*str*) – path to the directory for saving / loading the basis sets. If `None`, the basis set will not be saved to disk.
- **dr** (*float*) – size of one pixel in the radial direction. This only affects the absolute scaling of the transformed image.
- **verbose** (*boolean*) – determines whether statements should be printed
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed

**Returns** **recon** – the transformed (half) image

**Return type** m × n numpy array

`abel.basex.`**`basex_core_transform`**(*rawdata*, *A*)

Internal function that does the actual BASEX transform. It requires that the transform matrix be passed.

**Parameters**

- **rawdata** (*m × n numpy array*) – right half (with the axis) of the input image.
- **A** (*n × n numpy array*) – 2D array given by the transform-calculation function

**Returns** **IM** – the Abel-transformed image

**Return type** m × n numpy array

`abel.basex.`**`get_bs_cached`**(*n*, *sigma=1.0*, *reg=0.0*, *correction=True*, *basis_dir=u'.'*, *dr=1.0*, *verbose=False*, *direction=u'inverse'*)

Internal function.

Gets BASEX basis sets, using the disk as a cache (i.e. load from disk if they exist, if not, calculate them and save a copy on disk) and calculates the transform matrix. To prevent saving the basis sets to disk, set `basis_dir=None`. Loaded/calculated matrices are also cached in memory.

**Parameters**

- **n** (*int*) – Abel transform will be performed on an **n** pixels wide area of the (half) image
- **sigma** (*float*) – width parameter for basis functions
- **reg** (*float*) – regularization parameter
- **correction** (*boolean*) – apply intensity correction. Corrects wrong intensity normalization (seen for narrow basis sets), intensity oscillations (seen for broad basis sets), and intensity drop-off near $r = 0$ due to regularization.
- **basis_dir** (*str*) – path to the directory for saving / loading the basis sets. If `None`, the basis sets will not be saved to disk.
- **dr** (*float*) – pixel size. This only affects the absolute scaling of the output.
- **verbose** (*boolean*) – determines whether statements should be printed
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed

**Returns** **A** – matrix of the Abel transform (forward or inverse)

**Return type** n × n numpy array

`abel.basex.`**`cache_cleanup`**(*select=u'all'*)

Utility function.

Frees the memory caches created by `get_bs_cached()`. This is usually pointless, but might be required after working with very large images, if more RAM is needed for further tasks.

**Parameters** **select** (*str*) – selects which caches to clean:

**`all` (default)** everything, including basis;

**`forward`** forward transform;

**`inverse`** inverse transform.

**Returns**

**Return type** None

`abel.basex.`**`get_basex_correction`**(*A*, *sigma*, *direction*)

Internal function.

The default BASEX basis and the way its projection is calculated leads to artifacts in the reconstructed distribution – incorrect overall intensity for **sigma** = 1, intensity oscillations for other **sigma** values, intensity fluctuations (and drop-off for **reg** > 0) near *r* = 0. This function generates the intensity correction profile from the BASEX result for a step function with a soft edge (to avoid ringing) aligned with the last basis function.

**Parameters**

- **A** (*n* × *n numpy array*) – matrix of the Abel transform
- **sigma** (*float*) – basis width parameter
- **direction** (str: `'forward'` or `'inverse'`) – type of the Abel transform

**Returns** **cor** – intensity correction profile

**Return type** 1 × n numpy array

## 2.3 abel.linbasex module

`abel.linbasex.`**`linbasex_transform`**(*IM, basis_dir=None, proj_angles=[0, 1.5707963267948966], legendre_orders=[0, 2], radial_step=1, smoothing=0, rcond=0.0005, threshold=0.2, return_Beta=False, clip=0, norm_range=(0, -1), direction=u'inverse', verbose=False, dr=None*)

wrapper function for linebasex to process supplied quadrant-image as a full-image.

PyAbel transform functions operate on the right side of an image. Here we follow the *basex* technique of duplicating the right side to the left re-forming the whole image.

Inverse Abel transform using 1d projections of images.

*Gerber, Thomas, Yuzhu Liu, Gregor Knopp, Patrick Hemberger, Andras Bodi, Peter Radi, and Yaroslav Sych. Charged Particle Velocity Map Image Reconstruction with One-Dimensional Projections of Spherical Functions.* Review of Scientific Instruments 84, no. 3 (March 1, 2013): 033101–033101 – 10. <http://dx.doi.org/10.1063/1.4793404>‘_

`linbasex` models the image using a sum of Legendre polynomials at each radial pixel, As such, it should only be applied to situations that can be adequately represented by Legendre polynomials, i.e., images that feature spherical-like structures. The reconstructed 3D object is obtained by adding all the contributions, from which slices are derived.

**Parameters**

- **IM** (*numpy 2D array*) – image data must be square shape of odd size
- **proj_angles** (*list*) – projection angles, in radians (default $[0, \pi/2]$) e.g. $[0, \pi/2]$ or $[0, 0.955, \pi/2]$ or $[0, \pi/4, \pi/2, 3\pi/4]$
- **legendre_orders** (*list*) – orders of Legendre polynomials to be used as the expansion

  - even polynomials [0, 2, . . . ] gerade
  - odd polynomials [1, 3, . . . ] ungerade
  - all orders [0, 1, 2, . . . ].

  In a single photon experiment there are only anisotropies up to second order. The interaction of 4 photons (four wave mixing) yields anisotropies up to order 8.

- **radial_step** (*int*) – number of pixels per Newton sphere (default 1)
- **smoothing** (*float*) – convolve Beta array with a Gaussian function of 1/e 1/2 width *smoothing*.
- **rcond** (*float*) – (default 0.0005) scipy.linalg.lstsq fit conditioning value. set rcond to zero to switch conditioning off. Note: In the presence of noise the equation system may be ill posed. Increasing rcond smoothes the result, lowering it beyond a minimum renders the solution unstable. Tweak rcond to get a "reasonable" solution with acceptable resolution.
- **clip** (*int*) – clip first vectors (smallest Newton spheres) to avoid singularities (default 0)
- **norm_range** (*tuple*) – (low, high) normalization of Newton spheres, maximum in range Beta[0, low:high]. Note: Beta[0, i] the total number of counts integrated over sphere i, becomes 1.
- **threshold** (*float*) – threshold for normalization of higher order Newton spheres (default 0.2) Set all Beta[j], j>=1 to zero if the associated Beta[0] is smaller than threshold.
- **return_Beta** (*bool*) – return the Beta array of Newton spheres, as the tuple: radial-grid, Beta for the case `legendre_orders=[0, 2]`

  Beta[0] vs radius -> speed distribution

  Beta[2] vs radius -> anisotropy of each Newton sphere

  see 'Returns'.
- **direction** (*str*) – "inverse" - only option for this method. Abel transform direction.
- **dr** (*None*) – dummy variable for call compatibility with the other methods
- **verbose** (*bool*) – print information about processing (normally used for debugging)

**Returns**

- **inv_IM** (*numpy 2D array*) – inverse Abel transformed image
- **radial, Beta, projections** (*tuple*) – (if `return_Beta=True`)

  contributions of each spherical harmonic $Y_{i0}$ to the 3D distribution contain all the information one can get from an experiment. For the case `legendre_orders=[0, 2]`:

  Beta[0] vs radius -> speed distribution

  Beta[1] vs radius -> anisotropy of each Newton sphere.

  projections : are the radial projection profiles at angles *proj_angles*

`abel.linbasex.`**`linbasex_transform_full`**(*IM, basis_dir=None, proj_angles=[0, 1.5707963267948966], legendre_orders=[0, 2], radial_step=1, smoothing=0, rcond=0.0005, threshold=0.2, clip=0, return_Beta=False, norm_range=(0, -1), direction=u'inverse', verbose=False*)

Inverse Abel transform using 1d projections of images.

*Gerber, Thomas, Yuzhu Liu, Gregor Knopp, Patrick Hemberger, Andras Bodi, Peter Radi, and Yaroslav Sych. Charged Particle Velocity Map Image Reconstruction with One-Dimensional Projections of Spherical Functions.* Review of Scientific Instruments 84, no. 3 (March 1, 2013): 033101–033101 – 10. <http://dx.doi.org/10.1063/1.4793404>‘_

`linbasex` models the image using a sum of Legendre polynomials at each radial pixel, As such, it should only be applied to situations that can be adequately represented by Legendre polynomials, i.e., images that feature spherical-like structures. The reconstructed 3D object is obtained by adding all the contributions, from which slices are derived.

**Parameters**

- **IM** (*numpy 2D array*) – image data must be square shape of odd size
- **proj_angles** (*list*) – projection angles, in radians (default $[0, \pi/2]$) e.g. $[0, \pi/2]$ or $[0, 0.955, \pi/2]$ or $[0, \pi/4, \pi/2, 3\pi/4]$
- **legendre_orders** (*list*) – orders of Legendre polynomials to be used as the expansion
  - even polynomials [0, 2, . . . ] gerade
  - odd polynomials [1, 3, . . . ] ungerade
  - all orders [0, 1, 2, . . . ].

  In a single photon experiment there are only anisotropies up to second order. The interaction of 4 photons (four wave mixing) yields anisotropies up to order 8.
- **radial_step** (*int*) – number of pixels per Newton sphere (default 1)
- **smoothing** (*float*) – convolve Beta array with a Gaussian function of 1/e 1/2 width *smoothing*.
- **rcond** (*float*) – (default 0.0005) scipy.linalg.lstsq fit conditioning value. set rcond to zero to switch conditioning off. Note: In the presence of noise the equation system may be ill posed. Increasing rcond smoothes the result, lowering it beyond a minimum renders the solution unstable. Tweak rcond to get a “reasonable” solution with acceptable resolution.
- **clip** (*int*) – clip first vectors (smallest Newton spheres) to avoid singularities (default 0)
- **norm_range** (*tuple*) – (low, high) normalization of Newton spheres, maximum in range Beta[0, low:high]. Note: Beta[0, i] the total number of counts integrated over sphere i, becomes 1.
- **threshold** (*float*) – threshold for normalization of higher order Newton spheres (default 0.2) Set all Beta[j], j>=1 to zero if the associated Beta[0] is smaller than threshold.
- **return_Beta** (*bool*) – return the Beta array of Newton spheres, as the tuple: radial-grid, Beta for the case `legendre_orders=[0, 2]`

  Beta[0] vs radius -> speed distribution

  Beta[2] vs radius -> anisotropy of each Newton sphere

  see ‘Returns’.
- **direction** (*str*) – “inverse” - only option for this method. Abel transform direction.
- **dr** (*None*) – dummy variable for call compatibility with the other methods
- **verbose** (*bool*) – print information about processing (normally used for debugging)

**Returns**

- **inv_IM** (*numpy 2D array*) – inverse Abel transformed image

- **radial, Beta, projections** (*tuple*) – (if `return_Beta=True`)

  contributions of each spherical harmonic $Y_{i0}$ to the 3D distribution contain all the information one can get from an experiment. For the case `legendre_orders=[0, 2]`:

  Beta[0] vs radius -> speed distribution

  Beta[1] vs radius -> anisotropy of each Newton sphere.

  projections : are the radial projection profiles at angles *proj_angles*

`abel.linbasex.`**`int_beta`**(*Beta*, *radial_step=1*, *threshold=0.1*, *regions=None*)

Integrate beta over a range of Newton spheres.

**Parameters**

- **Beta** (*numpy array*) – Newton spheres
- **radial_step** (*int*) – number of pixels per Newton sphere (default 1)
- **threshold** (*float*) – threshold for normalisation of higher orders, 0.0 . . . 1.0.
- **regions** (*list of tuple radial ranges*) – [(min0, max0), (min1, max1), . . . ]

**Returns** **Beta_in** – integrated normalized Beta array [Newton sphere, region]

**Return type** numpy array

`abel.linbasex.`**`get_bs_cached`**(*cols, basis_dir=None, legendre_orders=[0, 2], proj_angles=[0, 1.5707963267948966], radial_step=1, clip=0, verbose=False*)

load basis set from disk, generate and store if not available.

Checks whether file: `linbasex_basis_{cols}_{legendre_orders}_{proj_angles}_{radial_step}_{cli`
`npy` is present in *basis_dir*

Either, read basis array or generate basis, saving it to the file.

**Parameters**

- **cols** (*int*) – width of image
- **basis_dir** (*str*) – path to the directory for saving / loading the basis
- **legendre_orders** (*list*) – default [0, 2] = 0 order and 2nd order polynomials
- **proj_angles** (*list*) – default [0, np.pi/2] in radians
- **radial_step** (*int*) – pixel grid size, default 1
- **clip** (*int*) – image edge clipping, default 0 pixels
- **verbose** (*boolean*) – print information for debugging

**Returns**

- **D** (*tuple (B, Bpol)*) – of ndarrays B (pol, proj, cols, cols) Bpol (pol, proj)
- **file.npy** (*file*) – saves basis to file name `linbasex_basis_{cols}_{legendre_orders}_{proj_angle`
  `npy`

`abel.linbasex.`**`cache_cleanup`**()

Utility function.

Frees the memory caches created by `get_bs_cached()`. This is usually pointless, but might be required after working with very large images, if more RAM is needed for further tasks.

**Parameters** **None**

**Returns**

**Return type** None

## 2.4 abel.hansenlaw module

abel.hansenlaw.**hansenlaw_transform**(*image*, *dr=1*, *direction=u'inverse'*, *hold_order=0*, ***kwargs*)

Forward/Inverse Abel transformation using the algorithm of:

E. W. Hansen "Fast Hankel Transform" IEEE Trans. Acoust. Speech Signal Proc. 33, 666 (1985)

and

E. W. Hansen and P.-L. Law "Recursive methods for computing the Abel transform and its inverse" J. Opt. Soc. Am. A 2, 510-520 (1985)

This function performs the Hansen-Law transform on only one "right-side" image:

```
Abeltrans = abel.hansenlaw.hansenlaw_transform(image, direction='inverse')
```

**Note:** Image should be a right-side image, like this:

```
.          +--------      +--------+
.          |       *      | *      |
.          |   *          |     *  |  <---------- im
.          |  *           |      * |
.          +--------      o--------+
.          |  *           |      * |
.          |   *          |     *  |
.          |     *        | *      |
.          +--------      +--------+
```

In accordance with all PyAbel methods the image center o is defined to be mid-pixel i.e. an odd number of columns, for the full image.

For the full image transform, use the *abel.Transform*.

Inverse Abel transform:

```
iAbel = abel.Transform(image, method='hansenlaw').transform
```

Forward Abel transform:

```
fAbel = abel.Transform(image, direction='forward', method='hansenlaw').transform
```

**Parameters**

- **image** (*1D or 2D numpy array*) – Right-side half-image (or quadrant). See figure below.
- **dr** (*float*) – Sampling size, used for Jacobian scaling. Default: *1* (appliable for pixel images).
- **direction** (*string 'forward' or 'inverse'*) – `forward` or `inverse` Abel transform. Default: 'inverse'.
- **hold_order** (*int 0 or 1*) – The order of the hold approximation, used to evaluate the state equation integral. *0* assumes a constant intensity across a pixel (between grid points) for the driving function (the image gradient for the inverse transform, or the original image, for the

forward transform). *1* assumes a linear intensity variation between grid points, which may yield a more accurate transform for some functions (see PR 211). Default: *0*.

**Returns** **aim** – forward/inverse Abel transform half-image

**Return type** 1D or 2D numpy array

## 2.5 abel.dasch module

`abel.dasch.`**`two_point_transform`**(*IM*, *basis_dir=u'.'*, *dr=1*, *direction=u'inverse'*, *verbose=False*)

**two-point deconvolution** C. J. Dasch Applied Optics 31, 1146 (1992). http://dx.doi.org/10.1364/AO.31.001146

**Parameters**

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str*) – path to the directory for saving / loading the "two-point" deconvolution operator array. Here, called *basis_dir* for consistency with the other true basis methods. If *None*, the operator array will not be saved to disk.
- **dr** (*float*) – sampling size (=1 for pixel images), used for Jacobian scaling. The resulting inverse transform is simply scaled by 1/dr.
- **direction** (*str*) – only the *direction="inverse"* transform is currently implemented
- **verbose** (*bool*) – trace printing

**Returns** **inv_IM** – the "two-point" inverse Abel transformed half-image

**Return type** 1D or 2D numpy array

`abel.dasch.`**`three_point_transform`**(*IM*, *basis_dir=u'.'*, *dr=1*, *direction=u'inverse'*, *verbose=False*)

**three-point deconvolution** C. J. Dasch Applied Optics 31, 1146 (1992). http://dx.doi.org/10.1364/AO.31.001146

**Parameters**

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str*) – path to the directory for saving / loading the "three-point" deconvolution operator array. Here, called *basis_dir* for consistency with the other true basis methods. If *None*, the operator array will not be saved to disk.
- **dr** (*float*) – sampling size (=1 for pixel images), used for Jacobian scaling. The resulting inverse transform is simply scaled by 1/dr.
- **direction** (*str*) – only the *direction="inverse"* transform is currently implemented
- **verbose** (*bool*) – trace printing

**Returns** **inv_IM** – the "three-point" inverse Abel transformed half-image

**Return type** 1D or 2D numpy array

`abel.dasch.`**`onion_peeling_transform`**(*IM*, *basis_dir=u'.'*, *dr=1*, *direction=u'inverse'*, *verbose=False*)

**onion-peeling deconvolution** C. J. Dasch Applied Optics 31, 1146 (1992). http://dx.doi.org/10.1364/AO.31.001146

**Parameters**

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str*) – path to the directory for saving / loading the "onion-peeling" deconvolution operator array. Here, called *basis_dir* for consistency with the other true basis methods. If *None*, the operator array will not be saved to disk.
- **dr** (*float*) – sampling size (=1 for pixel images), used for Jacobian scaling. The resulting inverse transform is simply scaled by 1/dr.
- **direction** (*str*) – only the *direction="inverse"* transform is currently implemented
- **verbose** (*bool*) – trace printing

**Returns** **inv_IM** – the "onion-peeling" inverse Abel transformed half-image

**Return type** 1D or 2D numpy array

`abel.dasch.`**`dasch_transform`**(*IM*, *D*)

Inverse Abel transform using the given deconvolution D-operator array.

**Parameters**

- **IM** (*2D numpy array*) – image data
- **D** (*2D numpy array*) – deconvolution operator array, of shape (cols, cols)

**Returns** **inv_IM** – inverse Abel transform according to deconvolution operator D

**Return type** 2D numpy array

`abel.dasch.`**`get_bs_cached`**(*method*, *cols*, *basis_dir=u'.'*, *verbose=False*)

load Dasch method deconvolution operator array from cache, or disk. Generate and store if not available.

Checks whether `method` deconvolution array has been previously calculated, or whether the file `{method}_basis_{cols}.npy` is present in *basis_dir*.

Either, assign, read, or generate the deconvolution array (saving it to file).

**Parameters**

- **method** (*str*) – Abel transform method `onion_peeling`, `three_point`, or `two_point`
- **cols** (*int*) – width of image
- **basis_dir** (*str or None*) – path to the directory for saving or loading the deconvolution array. For *None* do not save the deconvolution operator array
- **verbose** (*boolean*) – print information (mainly for debugging purposes)

**Returns**

- **D** (*numpy 2D array of shape (cols, cols)*) – deconvolution operator array for the associated method
- **file.npy** (*file*) – saves *D*, the deconvolution array to file name: `{method}_basis_{cols}.npy`

`abel.dasch.`**`cache_cleanup`**()

Utility function.

Frees the memory caches created by `get_bs_cached()`. This is usually pointless, but might be required after working with very large images, if more RAM is needed for further tasks.

**Parameters** **None**

**Returns**

**Return type** None

## 2.6 abel.onion_bordas module

`abel.onion_bordas.`**`onion_bordas_transform`**(*IM*, *dr=1*, *direction=u'inverse'*, *shift_grid=True*, ***kwargs*)

Onion peeling (or back projection) inverse Abel transform.

This algorithm was adapted by Dan Hickstein from the original Matlab implementation, created by Chris Rallis and Eric Wells of Augustana University, and described in this paper:

http://scitation.aip.org/content/aip/journal/rsi/85/11/10.1063/1.4899267

The algorithm actually originates from this 1996 RSI paper by Bordas et al:

http://scitation.aip.org/content/aip/journal/rsi/67/6/10.1063/1.1147044

This function operates on the "right side" of an image. i.e. it works on just half of a cylindrically symmetric image. Unlike the other transforms, the left edge should be the image center, not mid-first pixel. This corresponds to an even-width full image.

However, shift_grid=True (default) provides the typical behavior, where the first pixel corresponds to the center pixel of the image.

To perform a onion-peeling transorm on a whole image, use

```
abel.Transform(image, method='onion_bordas').transform
```

**Parameters**

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **dr** (*float*) – sampling size (=1 for pixel images), used for Jacobian scaling. The resulting inverse transform is simply scaled by 1/dr.
- **direction** (*str*) – only the *direction="inverse"* transform is currently implemented
- **shift_grid** (*boolean*) – place width-center on grid (bottom left pixel) by shifting image center (0, -1/2) pixel

**Returns** **AIM** – the inverse Abel transformed half-image

**Return type** 1D or 2D numpy array

## 2.7 abel.direct module

`abel.direct.`**`direct_transform`**(*fr*, *dr=None*, *r=None*, *direction=u'inverse'*, *derivative=<function gradient>*, *int_func=<function trapz>*, *correction=True*, *backend=u'C'*, ***kwargs*)

This algorithm performs a direct computation of the Abel transform integrals. When correction=False, the pixel at the lower bound of the integral (where y=r) is skipped, which causes a systematic error in the Abel transform. However, if correction=True is used, then an analytical transform transform is applied to this pixel, which makes the approximation that the function is linear across this pixel. With correction=True, the Direct method produces reasonable results.

The Direct method is implemented in both Python and, if Cython is available during PyAbel's installation, a compiled C version, which is much faster. The implementation can be selected using the backend argument.

By default, integration at all other pixels is performed using the Trapezoidal rule.

**Parameters**

- **fr** (*1d or 2d numpy array*) – input array to which direct/inverse Abel transform will be applied. For a 2d array, the first dimension is assumed to be the z axis and the second the r axis.
- **dr** (*float*) – spatial mesh resolution (optional, default to 1.0)
- **r** (*1D ndarray*) – the spatial mesh (optional). Unusually, direct_transform should, in principle, be able to handle non-uniform data. However, this has not been regorously tested.
- **direction** (*string*) – Determines if a forward or inverse Abel transform will be applied. can be 'forward' or 'inverse'.
- **derivative** (*callable*) – a function that can return the derivative of the fr array with respect to r. (only used in the inverse Abel transform).
- **int_func** (*function*) – This function is used to complete the integration. It should resemble np.trapz, in that it must be callable using axis=, x=, and dx= keyword arguments.
- **correction** (*boolean*) – If False the pixel where the weighting function has a singular value (where r==y) is simply skipped, causing a systematic under-estimation of the Abel transform. If True, integration near the singular value is performed analytically, by assuming that the data is linear across that pixel. The accuracy of this approximation will depend on how the data is sampled.
- **backend** (*string*) – There are currently two implementations of the Direct transform, one in pure Python and one in Cython. The backend paremeter selects which method is used. The Cython code is converted to C and compiled, so this is faster. Can be 'C' or 'python' (case insensitive). 'C' is the default, but 'python' will be used if the C-library is not available.

**Returns** **out** – with either the direct or the inverse abel transform.

**Return type** 1d or 2d numpy array of the same shape as fr

`abel.direct.`**`is_uniform_sampling`**(*r*)

Returns True if the array is uniformly spaced to within 1e-13. Otherwise False.

# CHAPTER 3

# Image processing tools

## 3.1 abel.tools.analytical module

**class** abel.tools.analytical.**BaseAnalytical**(*n*, *r_max*, *symmetric=True*, ***args*)
Bases: object

**class** abel.tools.analytical.**StepAnalytical**(*n*, *r_max*, *r1*, *r2*, *A0=1.0*, *ratio_valid_step=1.0*, *symmetric=True*)
Bases: *abel.tools.analytical.BaseAnalytical*

Define a symmetric step function and calculate its analytical Abel transform. See examples/example_step.py

**Parameters**

- **n** (*int*) – number of points along the r axis
- **r_max** (*float*) – range of the r interval
- **symmetric** (*boolean*) – if True, the r interval is [-r_max, r_max] (and n should be odd), otherwise the r interval is [0, r_max]
- **r1, r2** (*float*) – bounds of the step function if r > 0 (symmetric function is constructed for r < 0)
- **A0** (*float*) – height of the step
- **ratio_valid_step** (*float*) – in the benchmark take only the central ratio*100% of the step (exclude possible artefacts on the edges)

**abel_step_analytical**(*r*, *A0*, *r0*, *r1*)
Forward Abel transform of a step function located between r0 and r1, with a height A0

```
A0 +                  +-------------+
   |                  |             |
   |                  |             |
 0 | -----------------+             +-------------
   +------------------+-------------+------------>
   0                  r0            r1            r axis
```

**Parameters**

- **r1** (*1D array*) – vecor of positions along the r axis. Must start with 0.
- **r0, r1** (*float*) – positions of the step along the r axis
- **A0** (*float or 1D array*) – height of the step. If 1D array, the height can be variable along the z axis

**Returns**

**Return type** 1D array, if A0 is a float, a 2D array otherwise

**`sym_abel_step_1d`**(*r*, *A0*, *r0*, *r1*)
Produces a symmetrical analytical transform of a 1D step

**class** `abel.tools.analytical.`**`Polynomial`**(*n*, *r_max*, *r_1*, *r_2*, *c*, *r_0=0.0*, *s=1.0*, *reduced=False*, *symmetric=True*)
Bases: *`abel.tools.analytical.BaseAnalytical`*

Define a polynomial function and calculate its analytical Abel transform.

(See *Polynomials* for details and examples.)

**Parameters**

- **n** (*int*) – number of points along the *r* axis
- **r_max** (*float*) – range of the *r* interval
- **symmetric** (*boolean*) – if `True`, the *r* interval is [**r_max**, +**r_max**] (and **n** should be odd), otherwise the *r* interval is [0, **r_max**]
- **r_1, r_2** (*float*) – *r* bounds of the polynomial function if $r > 0$; outside [**r_1**, **r_2**] the function is set to zero (symmetric function is constructed for $r < 0$)
- **c** (*numpy array*) – polynomial coefficients in order of increasing degree: [$c_0$, $c_1$, $c_2$] means $c_0 + c_1\,r + c_2\,r^2$
- **r_0** (*float, optional*) – origin shift: the polynomial is defined as $c_0 + c_1\,(r\ \mathbf{r\_0}) + c_2\,(r\ \mathbf{r\_0})^2 + \ldots$
- **s** (*float, optional*) – *r* stretching factor (around **r_0**): the polynomial is defined as $c_0 + c_1\,(r/s) + c_2\,(r/s)^2 + \ldots$
- **reduced** (*boolean, optional*) – internally rescale the *r* range to [0, 1]; useful to avoid floating-point overflows for high degrees at large *r* (and might improve numerical accuracy)

**class** `abel.tools.analytical.`**`PiecewisePolynomial`**(*n*, *r_max*, *ranges*, *symmetric=True*)
Bases: *`abel.tools.analytical.BaseAnalytical`*

Define a piecewise polynomial function (sum of `Polynomials`) and calculate its analytical Abel transform.

**Parameters**

- **n** (*int*) – number of points along the *r* axis
- **r_max** (*float*) – range of the *r* interval
- **symmetric** (*boolean*) – if `True`, the *r* interval is [**r_max**, +**r_max**] (and **n** should be odd), otherwise the *r* interval is [0, **r_max**]
- **ranges** (*iterable of unpackable*) –

  (list of tuples of) polynomial parameters for each piece:

```
[(r_1_1st, r_2_1st, c_1st),
 (r_1_2nd, r_2_2nd, c_2nd),
 ...
 (r_1_nth, r_2_nth, c_nth)]
```

according to `Polynomial` conventions. All ranges are independent (may overlap and have gaps, may define polynomials of any degrees) and may include optional `Polynomial` parameters

**class** `abel.tools.analytical.`**`GaussianAnalytical`**(*n*, *r_max*, *sigma=1.0*, *A0=1.0*, *ratio_valid_sigma=2.0*, *symmetric=True*)

Bases: *`abel.tools.analytical.BaseAnalytical`*

Define a gaussian function and calculate its analytical Abel transform. See examples/example_gaussian.py

**Parameters**

- **n** (*int*) – number of points along the r axis
- **r_max** (*float*) – range of the r interval
- **symmetric** (*boolean*) – if True, the r interval is [-r_max, r_max] (and n should be odd), otherwise, the r interval is [0, r_max]
- **sigma** (*floats*) – sigma parameter for the gaussian
- **A0** (*float*) – amplitude of the gaussian
- **ratio_valid_sigma** (*float*) – in the benchmark take only the range 0 < r < ration_valid_sigma * sigma (exclude possible artefacts on the axis and the possibly clipped tail)

**class** `abel.tools.analytical.`**`TransformPair`**(*n*, *profile=5*)

Bases: *`abel.tools.analytical.BaseAnalytical`*

**Abel-transform pair analytical functions**.

**profiles 1–7**: Table 1 of Chan and Hieftje Spectrochimica Acta B 61, 31–41 (2006).

See *`abel.tools.transform_pairs`*.

**Returns**

- **r** (*numpy array*) – vector of positions along the r axis: *linspace(0, 1, n)*
- **dr** (*float*) – radial interval
- **func** (*numpy array*) – values of the original function (same shape as r)
- **abel** (*numpy array*) – values of the Abel transform (same shape as func)
- **label** (*str*) – name of the curve
- **mask_valid** (*boolean array*) – set all True. Used for unit tests

**class** `abel.tools.analytical.`**`SampleImage`**(*n=361*, *name='dribinski'*, *sigma=2*, *temperature=200*)

Bases: *`abel.tools.analytical.BaseAnalytical`*

Sample images, made up of Gaussian functions

**Parameters**

- **n** (*integer*) – image size n rows x n cols
- **name** (*str*) – one of “dribinski” or “Ominus”
- **sigma** (*float*) – Gaussian 1/e width (pixels)

- **temperature** (*float*) – for 'Ominus' only anion levels have Boltzmann population weight (2J+1) exp(-177.1 h c 100/k/temperature)

**`image`**
image

**Type** 2D numpy array

**`name`**
sample image name

**Type** str

## 3.2 abel.tools.center module

`abel.tools.center.`**`find_center`**(*IM*, *center=u'image_center'*, *square=False*, *verbose=False*, ***kwargs*)

**Find the coordinates of image center, using the method** specified by the *center* parameter.

**Parameters**

- **IM** (*2D np.array*) – image data
- **center** (*str*) – this determines how the center should be found. The options are:

  **`image_center`** the center of the image is used as the center. The trivial result.

  **`com`** the center is found as the center of mass.

  **`convolution`** center by convolution of two projections along each axis.

  **`gaussian`** the center is extracted by a fit to a Gaussian function. This is probably only appropriate if the data resembles a gaussian.

  **`slice`** the image is broken into slices, and these slices compared for symmetry.

- **square** (*bool*) – if 'True' returned image will have a square shape

**Returns** **out** – coordinate of the center of the image in the (y,x) format (row, column)

**Return type** (float, float)

`abel.tools.center.`**`center_image`**(*IM*, *center=u'com'*, *odd_size=True*, *square=False*, *axes=(0, 1)*, *crop=u'maintain_size'*, *verbose=False*, ***kwargs*)

**Center image with the custom value or by several methods provided in** *find_center* function

**Parameters**

- **IM** (*2D np.array*) – The image data.
- **center** (*tuple or str*) – center can either be (float, float), the coordinate of the center of the image in the (y,x) format (row, column)

  Or, it can be a string, to specify an automatic centering method. The options are:

  **`image_center`** the center of the image is used as the center. The trivial result.

  **`com`** the center is found as the center of mass.

  **`convolution`** center by convolution of two projections along each axis.

`gaussian` the center is extracted by a fit to a Gaussian function. This is probably only appropriate if the data resembles a gaussian.

`slice` the image is broken into slices, and these slices compared for symmetry.

- **odd_size** (*boolean*) – if True, an image will be returned containing an odd number of columns. Most of the transform methods require this, so it's best to set this to True if the image will subsequently be Abel transformed.
- **square** (*bool*) – if 'True' returned image will have a square shape
- **crop** (*str*) – This determines how the image should be cropped. The options are:

  `maintain_size` return image of the same size. Some of the original image may be lost and some regions may be filled with zeros.

  `valid_region` return the largest image that can be created without padding. All of the returned image will correspond to the original image. However, portions of the original image will be lost. If you can tolerate clipping the edges of the image, this is probably the method to choose.

  `maintain_data` the image will be padded with zeros such that none of the original image will be cropped.
- **axes** (*int or tuple*) – center image with respect to axis `0` (vertical), `1` (horizontal), or both axes `(0, 1)` (default).

**Returns** **out** – Centered image

**Return type** 2D np.array

`abel.tools.center.`**`set_center`**(*data*, *center*, *crop=u'maintain_size'*, *axes=(0, 1)*, *verbose=False*)

Move image center to mid-point of image

**Parameters**

- **data** (*2D np.array*) – The image data
- **center** (*tuple*) – image pixel coordinate center (row, col)
- **crop** (*str*) – This determines how the image should be cropped. The options are:

  `maintain_size` return image of the same size. Some of the original image may be lost and some regions may be filled with zeros.

  `valid_region` return the largest image that can be created without padding. All of the returned image will correspond to the original image. However, portions of the original image will be lost. If you can tolerate clipping the edges of the image, this is probably the method to choose.

  `maintain_data` the image will be padded with zeros such that none of the original image will be cropped.
- **axes** (*int or tuple*) – center image with respect to axis `0` (vertical), `1` (horizontal), or both axes `(0, 1)` (default).
- **verbose** (*boolean*) – True: print diagnostics

`abel.tools.center.`**`find_center_by_center_of_mass`**(*IM*, *verbose=False*, *round_output=False*, ***kwargs*)

Find image center by calculating its center of mass

`abel.tools.center.`**`find_center_by_convolution`**(*IM*, ***kwargs*)

**Center the image by convolution of two projections along each axis.** Code from the `linbasex` juptyer notebook

**Parameters** **IM** (*numpy 2D array*) – image data

**Returns** **center** – (row-center, col-center)

**Return type** tuple

`abel.tools.center.`**`find_center_by_center_of_image`**(*data*, *verbose=False*, ***kwargs*)

Find image center simply from its dimensions.

`abel.tools.center.`**`find_center_by_gaussian_fit`**(*IM*, *verbose=False*, *round_output=False*, ***kwargs*)

Find image center by fitting the summation along x and y axis of the data to two 1D Gaussian function.

`abel.tools.center.`**`axis_slices`**(*IM*, *radial_range=(0, -1)*, *slice_width=10*)

returns vertical and horizontal slice profiles, summed across slice_width.

**Parameters**

- **IM** (*2D np.array*) – image data
- **radial_range** (*tuple floats*) – (rmin, rmax) range to limit data
- **slice_width** (*integer*) – width of the image slice, default 10 pixels

**Returns** **top, bottom, left, right** – image slices oriented in the same direction

**Return type** 1D np.arrays shape (rmin:rmax, 1)

`abel.tools.center.`**`find_image_center_by_slice`**(*IM*, *slice_width=10*, *radial_range=(0, -1)*, *axis=(0, 1)*, ***kwargs*)

Center image by comparing opposite side, vertical (`axis=0`) and/or horizontal slice (`axis=1`) profiles. To center along both axis, use `axis=(0,1)`.

**Parameters**

- **IM** (*2D np.array*) – The image data.
- **slice_width** (*integer*) – Sum together this number of rows (cols) to improve signal, default 10.
- **radial_range** (*tuple*) – (rmin,rmax): radial range [rmin:rmax] for slice profile comparison.
- **axis** (*integer or tuple*) – Center with along `axis=0` (vertical), or `axis=1` (horizontal), or `axis=(0,1)` (both vertical and horizontal).

**Returns** **(vertical_shift, horizontal_shift)** – (axis=0 shift, axis=1 shift)

**Return type** tuple of floats

## 3.3 abel.tools.circularize module

`abel.tools.circularize.`**`circularize_image`**(*IM*, *method='lsq'*, *center=None*, *radial_range=None*, *dr=0.5*, *dt=0.5*, *smooth=0*, *ref_angle=None*, *inverse=False*, *return_correction=False*)

Corrects image distortion on the basis that the structure should be circular.

This is a simplified radial scaling version of the algorithm described in J. R. Gascooke and S. T. Gibson and W. D. Lawrance: 'A "circularisation" method to repair deformations and determine the centre of velocity map images' J. Chem. Phys. 147, 013924 (2017).

This function is especially useful for correcting the image obtained with a velocity-map-imaging spectrometer, in the case where there is distortion of the Newton Sphere (ring) structure due to an imperfect electrostatic lens or stray electromagnetic fields. The correction allows the highest-resolution 1D photoelectron distribution to be extracted.

The algorithm splits the image into "slices" at many different angles (set by *dt*) and compares the radial intensity profile of adjacent slices. A scaling factor is found which aligns each slice profile with the previous slice. The image is then corrected using a spline function that smoothly connects the discrete scaling factors as a continuous function of angle.

This circularization algorithm should only be applied to a well-centered image, otherwise use the *center* keyword (described below) to center it.

**Parameters**

- **IM** (*numpy 2D array*) – Image to be circularized.
- **method** (*str*) – Method used to determine the radial correction factor to align slice profiles:

  **`argmax` - compare intensity-profile.argmax() of each radial slice.** This method is quick and reliable, but it assumes that the radial intensity profile has an obvious maximum. The positioning is limited to the nearest pixel.

  **`lsq` - minimize the difference between a slice intensity-profile** with its adjacent slice. This method is slower and may fail to converge, but it may be applied to images with any (circular) structure. It aligns the slices with sub-pixel precision.
- **center** (*str, float tuple, or None*) – Pre-center image using `abel.tools.center.center_image()`. *center* may be: *com*, *convolution*, *gaussian*, *image_center*, *slice*, or a float tuple center $(y, x)$.
- **radial_range** (*tuple, or None*) – Limit slice comparison to the radial range tuple (rmin, rmax), in pixels, from the image center. Use to determine the distortion correction associated with particular peaks. It is recommended to select a region of your image where the signal-to-noise is highest, with sharp persistant (in angle) features.
- **dr** (*float*) – Radial grid size for the polar coordinate image, default = 0.5 pixel. This is passed to `abel.tools.polar.reproject_image_into_polar()`.

  Small values may improve the distortion correction, which is often of sub-pixel dimensions, at the cost of reduced signal to noise for the slice intensity profile. As a general rule, *dr* should be significantly smaller than the radial "feature size" in the image.
- **dt** (*float*) – Angular grid size. This sets the number of radial slices, given by $2\pi/dt$. Default = 0.1, ~ 63 slices. More slices, using smaller *dt*, may provide a more detailed angular variation of the correction, at the cost of greater signal to noise in the correction function.

  Also passed to `abel.tools.polar.reproject_image_into_polar()`
- **smooth** (*float*) – This value is passed to the `scipy.interpolate.UnivariateSpline()` function and controls how smooth the spline interpolation is. A value of zero corresponds to a spline that runs through all of the points, and higher values correspond to a smoother spline function.

  It is important to examine the relative peak position (scaling factor) data and how well it is represented by the spline function. Use the option `return_correction=True` to examine this data. Typically, *smooth* may remain zero, noisy data may require some smoothing.

- **ref_angle** (*None* or float) – Reference angle for which radial coordinate is unchanged. Angle varies between $-\pi$ to $\pi$, with zero angle vertical.

  *None* uses `numpy.mean(radial scale factors)()`, which attempts to maintain the same average radial scaling. This approximation is likely valid, unless you know for certain that a specific angle of your image corresponds to an undistorted image.
- **inverse** (*bool*) – Apply an inverse Abel transform the **polar** coordinate image, to remove the background intensity. This may improve the signal to noise, allowing the weaker intensity featured to be followed in angle.

  Note that this step is only for the purposes of allowing the algorithm to better follow peaks in the image. It does not affect the final image that is returned, except for (hopefully) slightly improving the precision of the distortion correction.
- **return_correction** (*bool*) – Additional outputs, as describe below.

**Returns**

- **IMcirc** (*numpy 2D array, same size as input*) – Circularized version of the input image.

  The following values are returned if `return_correction=True`:
- **angles** (*numpy 1D array*) – Mid-point angle (radians) of each image slice.
- **radial_correction** (*numpy 1D array*) – Radial correction scale factor at each angular slice.
- **radial_correction_function** (*numpy function that accepts numpy.array*) – Function that may be used to evaluate the radial correction at any angle.

`abel.tools.circularize.`**`circularize`**(*IM*, *radial_correction_function*, *ref_angle=None*)
Remap image from its distorted grid to the true cartesian grid.

**Parameters**

- **IM** (*numpy 2D array*) – Original image
- **radial_correction_function** (*funct*) – A function returning the radial correction for a given angle. It should accept a numpy 1D array of angles.

`abel.tools.circularize.`**`correction`**(*polarIMTrans*, *angles*, *radial*, *method*)

**Determines a radial correction factors that align an angular slice** radial intensity profile with its adjacent (previous) slice profile.

**Parameters**

- **polarIMTrans** (*numpy 2D array*) – Polar coordinate image, transposed $(\theta, r)$ so that each row is a single angle.
- **angles** (*numpy 1D array*) – Angle coordinates for one row of *polarIMTrans*.
- **radial** (*numpy 1D array*) – Radial coordinates for one column of *polarIMTrans*.
- **method** (*str*) – "argmax": radial correction factor from position of maximum intensity.

  "lsq" : least-squares determine a radial correction factor that will align a radial intensity profile with the previous, adjacent slice.

## 3.4 abel.tools.math module

`abel.tools.math.`**`gradient`**(*f*, *x=None*, *dx=1*, *axis=-1*)
Return the gradient of 1 or 2-dimensional array. The gradient is computed using central differences in the

interior and first differences at the boundaries.

Irregular sampling is supported (it isn't supported by np.gradient)

**Parameters**

- **f** (*1d or 2d numpy array*) – Input array.
- **x** (*array_like, optional*) – Points where the function f is evaluated. It must be of the same length as `f.shape[axis]`. If None, regular sampling is assumed (see dx)
- **dx** (*float, optional*) – If *x* is None, spacing given by *dx* is assumed. Default is 1.
- **axis** (*int, optional*) – The axis along which the difference is taken.

**Returns** **out** – Returns the gradient along the given axis.

**Return type** array_like

## Notes

To-Do: implement smooth noise-robust differentiators for use on experimental data. http://www.holoborodko.com/pavel/numerical-methods/numerical-derivative/smooth-low-noise-differentiators/

`abel.tools.math.`**`gaussian`**(*x*, *a*, *mu*, *sigma*, *c*)

Gaussian function

$f(x) = ae^{-(x-\mu)^2/(2\sigma^2)} + c$

ref: https://en.wikipedia.org/wiki/Gaussian_function

**Parameters**

- **x** (*1D np.array*) – coordinate
- **a** (*float*) – the height of the curve's peak
- **mu** (*float*) – the position of the center of the peak
- **sigma** (*float*) – the standard deviation, sometimes called the Gaussian RMS width
- **c** (*float*) – non-zero background

**Returns** **out** – the Gaussian profile

**Return type** 1D np.array

`abel.tools.math.`**`guss_gaussian`**(*x*)

Find a set of better starting parameters for Gaussian function fitting

**Parameters** **x** (*1D np.array*) – 1D profile of your data

**Returns** **out** – estimated value of (a, mu, sigma, c)

**Return type** tuple of float

`abel.tools.math.`**`fit_gaussian`**(*x*)

Fit a Gaussian function to x and return its parameters

**Parameters** **x** (*1D np.array*) – 1D profile of your data

**Returns** **out** – (a, mu, sigma, c)

**Return type** tuple of float

## 3.5 abel.tools.polar module

`abel.tools.polar.`**`reproject_image_into_polar`**(*data*, *origin=None*, *Jacobian=False*, *dr=1*, *dt=None*)

Reprojects a 2D numpy array (`data`) into a polar coordinate system. “origin” is a tuple of (x0, y0) relative to the bottom-left image corner, and defaults to the center of the image.

**Parameters**

- **data** (*2D np.array*)
- **origin** (*tuple*) – The coordinate of the image center, relative to bottom-left
- **Jacobian** (*boolean*) – Include `r` intensity scaling in the coordinate transform. This should be included to account for the changing pixel size that occurs during the transform.
- **dr** (*float*) – Radial coordinate spacing for the grid interpolation tests show that there is not much point in going below 0.5
- **dt** (*float*) – Angular coordinate spacing (in radians) if `dt=None`, dt will be set such that the number of theta values is equal to the maximum value between the height or the width of the image.

**Returns**

- **output** (*2D np.array*) – The polar image (r, theta)
- **r_grid** (*2D np.array*) – meshgrid of radial coordinates
- **theta_grid** (*2D np.array*) – meshgrid of theta coordinates

### Notes

Adapted from: http://stackoverflow.com/questions/3798333/image-information-along-a-polar-coordinate-system

`abel.tools.polar.`**`index_coords`**(*data*, *origin=None*)

Creates x & y coords for the indicies in a numpy array

**Parameters**

- **data** (*numpy array*) – 2D data
- **origin** (*(x,y) tuple*) – defaults to the center of the image. Specify origin=(0,0) to set the origin to the *bottom-left* corner of the image.

**Returns** **x, y**

**Return type** arrays

`abel.tools.polar.`**`cart2polar`**(*x*, *y*)

Transform Cartesian coordinates to polar

**Parameters** **x, y** (*floats or arrays*) – Cartesian coordinates

**Returns** **r, theta** – Polar coordinates

**Return type** floats or arrays

`abel.tools.polar.`**`polar2cart`**(*r*, *theta*)

Transform polar coordinates to Cartesian

**Parameters** **r, theta** (*floats or arrays*) – Polar coordinates

**Returns** **x, y** – Cartesian coordinates

**Return type** floats or arrays

## 3.6 abel.tools.polynomial module

See *Polynomials* for details and examples.

### 3.6.1 Polynomials

Implemented in `abel.tools.polynomial`.

#### Abel transform

The Abel transform of a polynomial

$$\text{func}(r) = \sum_{k=0}^{K} c_k r^k$$

defined on a domain $[r_{\text{min}}, r_{\text{max}}]$ (and zero elsewhere) is calculated as

$$\text{abel}(x) = \sum_{k=0}^{K} c_k \int r^k \, dy,$$

where $r = \sqrt{x^2 + y^2}$, and the Abel integral is taken over the domain where $r_{\text{min}} \le r \le r_{\text{max}}$. Namely,

$$\int r^k \, dy = 2 \int_{y_{\text{min}}}^{y_{\text{max}}} r^k \, dy,$$

$$y_{\text{min,max}} = \begin{cases} \sqrt{r_{\text{min,max}}^2 - x^2}, & x < r_{\text{min,max}}, \\ 0 & \text{otherwise}, \end{cases}$$

These integrals for any power $k$ are easily obtained from the recursive relation

$$\int r^k \, dy = \frac{1}{k+1} \left( y r^k + k x^2 \int r^{k-2} \, dy \right).$$

For **even** $k$ this yields a polynomial in $y$ and powers of $x$ and $r$:

$$\int r^k \, dy = y \sum_{m=0}^{k} C_m r^m x^{k-m}, \qquad \text{(summing over even } m\text{)}$$

$$C_k = \frac{1}{k+1}, \quad C_{m-2} = \frac{m}{m-1} C_m.$$

For **odd** $k$, the recursion terminates at

$$\int r^{-1} \, dy = \ln(y + r),$$

so

$$\int r^k \, dy = y \sum_{m=1}^{k} C_m r^m x^{k-m} + C_1 x^{k+1} \ln(y + r), \qquad \text{(summing over odd } m\text{)}$$

with the same expressions for $C_m$.

These sums are computed using Horner's method in $x$, which requires only $x^2$, $y$ (see above), $\ln(y+r)$ (for polynomials with odd degrees), and powers of $r$ up to $K$.

The sum of the integrals, however, is computed by direct addition. In particular, this means that an attempt to use this method for high-degree polynomials (for example, approximating some function with a 100-degree Taylor polynomial) will most likely fail due to loss of significance in floating-point operations. Splines are a much better choice in this respect, although at sufficiently large $r$ and $x$ (10000) these numerical problems might become significant even for cubic polynomials.

### Affine transformation

It is sometimes convenient to define a polynomial in some canonical form and adapt it to the particular case by an affine transformation (translation and scaling) of the independent variable, like in the *example* below.

The scaling around $r = 0$ is

$$P'(r) = P(r/s) = \sum_{k=0}^{K} c_k (r/s)^k,$$

which applies an $s$-fold stretching to the function. The coefficients of the transformed polynomial are thus

$$c'_k = c_k / s^k.$$

The translation is

$$P'(r) = P(r - r_0) = \sum_{k=0}^{K} c_k (r - r_0)^k,$$

which shifts the origin to $r_0$. The coefficients of the transformed polynomial can be obtained by expanding all powers of the binomial $r - r_0$ and collecting the powers of $r$. This is implemented in a matrix form

$$\mathbf{c}' = \mathrm{M}\mathbf{c},$$

where the coefficients are represented by a column vector $\mathbf{c} = (c_0, c_1, \dots, c_K)^\mathrm{T}$, and the matrix $\mathrm{M}$ is the Hadamard product of the upper-triangular Pascal matrix and the Toeplitz matrix of $r_0^k$:

$$\mathrm{M} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & \cdots \\ 0 & 1 & 2 & 3 & 4 & \cdots \\ 0 & 0 & 1 & 3 & 6 & \cdots \\ 0 & 0 & 0 & 1 & 4 & \cdots \\ 0 & 0 & 0 & 0 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \circ \begin{pmatrix} r_0^0 & r_0^1 & r_0^2 & \ddots & r_0^K \\ 0 & r_0^0 & r_0^1 & \ddots & r_0^{K-1} \\ 0 & 0 & r_0^0 & \ddots & r_0^{K-2} \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \ddots & r_0^0 \end{pmatrix}.$$

### Example

Consider a two-sided step function with soft edges:

The edges can be represented by the cubic smoothstep function

$$S(r) = 3r^2 - 2r^3,$$

which smoothly rises from $0$ at $r = 0$ to $1$ at $r = 1$. The left edge requires stretching it by $2w$ and shifting the origin to $r_\text{min} - w$. The right edge is $S(r)$ stretched by $-2w$ (the negative sign mirrors it horizontally) and shifted to $r_\text{max} + w$. The shelf is just a constant (zeroth-degree polynomial). It can be set to $1$, and then the desired function with the amplitude $A$ is obtained by multiplying the resulting piecewise polynomial by $A$:

```
import matplotlib.pyplot as plt
import numpy as np

from abel.tools.polynomial import PiecewisePolynomial as PP

r = np.arange(51.0)

rmin = 10
rmax = 40
w = 5
A = 3

c = [0, 0, 3, -2]
smoothstep = A * PP(r, [(rmin - w, rmin + w, c, rmin - w, 2 * w),
                        (rmin + w, rmax - w, [1]),
                        (rmax - w, rmax + w, c, rmax + w, -2 * w)])

fig, axs = plt.subplots(2, 1)

axs[0].set_title('func')
axs[0].set_xlabel('$r$')
axs[0].plot(r, smoothstep.func)

axs[1].set_title('abel')
axs[1].set_xlabel('$x$')
axs[1].plot(r, smoothstep.abel)

plt.tight_layout()
plt.show()
```

`Polynomial` and `PiecewisePolynomial` are also accessible through the *`abel.tools.analytical`* module. Amplitude scaling by multiplying the "function" (a Python object actually) is not supported there, but it can be achieved simply by scaling all the coefficients:

```
from abel.tools.analytical import PiecewisePolynomial as PP
c = A * np.array([0, 0, 3, -2])
smoothstep = PP(..., [(rmin - w, rmin + w, c, rmin - w, 2 * w),
                      (rmin + w, rmax - w, [A]),
                      (rmax - w, rmax + w, c, rmax + w, -2 * w)], ...)
```

**class** `abel.tools.polynomial.`**`Polynomial`**(*r, r_min, r_max, c, r_0=0.0, s=1.0, reduced=False*)

Bases: `object`

Polynomial function and its Abel transform.

Supports multiplication and division by numbers.

**Parameters**

- **r** (*numpy array*) – *r* values at which the function is generated (and *x* values for its Abel transform); must be non-negative and in ascending order
- **r_min, r_max** (*float*) – *r* domain: the function is defined as the polynomial on [**r_min**, **r_max**] and zero outside it; 0 **r_min** < **r_max** **max r** (**r_max** might exceed maximal **r**, but usually by < 1 pixel)
- **c** (*numpy array*) – polynomial coefficients in order of increasing degree: [$c_0$, $c_1$, $c_2$] means $c_0 + c_1\, r + c_2\, r^2$

- **r_0** (*float, optional*) – origin shift: the polynomial is defined as $c_0 + c_1 (r - \mathbf{r\_0}) + c_2 (r - \mathbf{r\_0})^2 + \ldots$
- **s** (*float, optional*) – *r* stretching factor (around **r_0**): the polynomial is defined as $c_0 + c_1 (r/\mathbf{s}) + c_2 (r/\mathbf{s})^2 + \ldots$
- **reduced** (*boolean, optional*) – internally rescale the *r* range to [0, 1]; useful to avoid floating-point overflows for high degrees at large r (and might improve numeric accuracy)

**class** `abel.tools.polynomial.`**`PiecewisePolynomial`**(*r*, *ranges*)

Bases: *`abel.tools.polynomial.Polynomial`*

Piecewise polynomial function (sum of `Polynomials`) and its Abel transform.

Supports multiplication and division by numbers.

**Parameters**

- **r** (*numpy array*) – *r* values at which the function is generated (and *x* values for its Abel transform)
- **ranges** (*iterable of unpackable*) –

  (list of tuples of) polynomial parameters for each piece:

```
[(r_min_1st, r_max_1st, c_1st),
 (r_min_2nd, r_max_2nd, c_2nd),
 ...
 (r_min_nth, r_max_nth, c_nth)]
```

  according to `Polynomial` conventions. All ranges are independent (may overlap and have gaps, may define polynomials of any degrees) and may include optional `Polynomial` parameters

## 3.7 abel.tools.transform_pairs module

Analytical function Abel-transform pairs

**profiles 1–7, table 1 of:** G. C.-Y Chan and G. M. Hieftje Spectrochimica Acta B 61, 31–41 (2006)

**Note:** the transform pair functions are more conveniently accessed through *`abel.tools.analytical.TransformPair`*:

```
func = abel.tools.analytical.TransformPair(n, profile=nprofile)
```

which sets the radial range *r* and provides attributes `.func` (source), `.abel` (projection), `.r` (radial range), `.dr` (step), `.label` (the profile name)

**Parameters** **r** (*floats or numpy 1D array of floats*) – value or grid to evaluate the function pair: `0 < r < 1`

**returns** **source, projection** – source function profile (inverse Abel transform of projection), projection functon profile (forward Abel transform of source)

**rtype** tuple of 1D numpy arrays of shape *r* `abel.tools.transform_pairs.`**`a`**`(n, r)`

coefficient

$$a_n = \sqrt{n^2 - r^2}$$

`abel.tools.transform_pairs.`**`profile1`**`(r)`

**profile1**: Cremers and Birkebak App. Opt. 5, 1057–1064 (1966) Eq(13)

$$\begin{aligned}
\epsilon(r) &= 0.75 + 12r^2 - 32r^3 && 0 \le r \le 0.25\\
\epsilon(r) &= \frac{16}{27}(1 + 6r - 15r^2 + 8r^3) && 0.25r \le 1\\
I(r) &= \frac{1}{108}(128a_1 + a_{0.25}) + \frac{2}{27}r^2(283a_{0.25} - 112a_1) + \\
&\quad \frac{8}{9}r^2\left[4(1+r^2)\ln\frac{1+a_1}{r} - (4+31r^2)\ln\frac{0.25+a_{0.25}}{r}\right] && 0 \le r \le 0.25\\
I(r) &= \frac{32}{27}\left[a_1 - 7a_1 r + 3r^2(1+r^2)\ln\frac{1+a_1}{r}\right] && 0.25r \le 1
\end{aligned}$$

`abel.tools.transform_pairs.`**`profile2`**`(r)`

**profile2**: Cremers and Birkebak App. Opt. 5, 1057–1064 (1966) Eq(13)

$$\begin{aligned}
\epsilon(r) &= 1 - 3r^2 + 2r^3 && 0 \le r \le 1\\
I(r) &= a_1\left(1 - \frac{5}{2}r^2\right) + \frac{3}{2}r^4\ln\frac{1+a_1}{r} && 0 \le r \le 1
\end{aligned}$$

`abel.tools.transform_pairs.`**`profile3`**`(r)`

**profile3**: Cremers and Birkebak App. Opt. 5, 1057–1064 (1966) Eq(13)

$$\begin{aligned}
\epsilon(r) &= 1 - 2r^2 && 0 \le r \le 0.5\\
\epsilon(r) &= 2(1 - r^2)^2 && 0.5r \le 1\\
I(r) &= \frac{4a_1}{3}(1 + 2r^2) - \frac{2a_{0.5}}{3}(1 + 8r^2) - 4r^2\ln\frac{1-a_1}{0.5+a_{0.5}} && 0 \le r \le 0.5\\
I(r) &= \frac{4a_1}{3}(1 + 2r^2) - 4r^2\ln\frac{1-a_1}{r} && 0.5r \le 1
\end{aligned}$$

`abel.tools.transform_pairs.`**`profile4`**`(r)`

**profile4**: Alvarez, Rodero, Quintero Spectochim. Acta B 57, 1665–1680 (2002)

**Note:** Published projection has misprints ("19**3**.30083" instead of "19**6**.30083" in both cases).

$$\begin{aligned}
\epsilon(r) &= 0.1 + 5.51r^2 - 5.25r^3 && 0 \le r \le 0.7\\
\epsilon(r) &= -40.74 + 155.56r - 188.89r^2 + 74.07r^3 && 0.7r \le 1\\
I(r) &= 22.68862a_{0.7} - 14.811667a_1 + (217.557a_{0.7} - 196.30083a_1)r^2 + \\
&\quad 155.56r^2\ln\frac{1+a_1}{0.7+a_{0.7}} + r^4\left(55.5525\ln\frac{1+a_1}{r} - 59.49\ln\frac{0.7+a_{0.7}}{r}\right) && 0 \le r \le 0.7\\
I(r) &= -14.811667a_1 - 196.30083a_1r^2 + r^2(155.56 + 55.5525r^2)\ln\frac{1+a_1}{r} && 0.7r \le 1
\end{aligned}$$

`abel.tools.transform_pairs.`**`profile5`**`(r)`

**profile5**: Buie et al. J. Quant. Spectrosc. Radiat. Transfer 55, 231–243 (1996)

$$\begin{aligned}\epsilon(r) &= 1 & 0 \le r \le 1\\ I(r) &= 2a_1 & 0 \le r \le 1\end{aligned}$$

`abel.tools.transform_pairs.`**`profile6`**`(r)`

**profile6**: Buie et al. J. Quant. Spectrosc. Radiat. Transfer 55, 231–243 (1996)

$$\begin{aligned}\epsilon(r) &= (1-r^2)^{-\frac{3}{2}} \exp\left[1.1^2\left(1-\frac{1}{1-r^2}\right)\right] & 0 \le r \le 1\\ I(r) &= \frac{\sqrt{\pi}}{1.1a_1}\exp\left[1.1^2\left(1-\frac{1}{1-r^2}\right)\right] & 0 \le r \le 1\end{aligned}$$

`abel.tools.transform_pairs.`**`profile7`**`(r)`

**profile7**: Buie et al. J. Quant. Spectrosc. Radiat. Transfer 55, 231–243 (1996)

$$\begin{aligned}\epsilon(r) &= \frac{1}{2}(1+10r^2-23r^4+12r^6) & 0 \le r \le 1\\ I(r) &= \frac{8}{105}a_1(19+34r^2-125r^4+72r^6) & 0 \le r \le 1\end{aligned}$$

## 3.8 abel.tools.symmetry module

`abel.tools.symmetry.`**`get_image_quadrants`**`(`*IM*, *reorient=True*, *symmetry_axis=None*, *use_quadrants=(True, True, True, True)*, *symmetrize_method=u'average'*`)`

Given an image (m,n) return its 4 quadrants Q0, Q1, Q2, Q3 as defined below.

**Parameters**

- **IM** (*2D np.array*) – Image data shape (rows, cols)
- **reorient** (*boolean*) – Reorient quadrants to match the orientation of Q0 (top-right)
- **symmetry_axis** (*int or tuple*) – can have values of `None`, `0`, `1`, or `(0,1)` and specifies no symmetry, vertical symmetry axis, horizontal symmetry axis, and both vertical and horizontal symmetry axes. Quadrants are added. See Note.
- **use_quadrants** (*boolean tuple*) – Include quadrant (Q0, Q1, Q2, Q3) in the symmetry combination(s) and final image
- **symmetrize_method** (*str*) – Method used for symmetrizing the image.

  **`average`** Simply average the quadrants.

  **`fourier`** Axial symmetry implies that the Fourier components of the 2-D projection should be real. Removing the imaginary components in reciprocal space leaves a symmetric projection.

  (ref: Overstreet, K., et al. "Multiple scattering and the density distribution of a Cs MOT." Optics express 13.24 (2005): 9672-9682. http://dx.doi.org/10.1364/OPEX.13.009672)

**Returns** **Q0, Q1, Q2, Q3** – shape: `(rows//2+rows%2, cols//2+cols%2)` all oriented in the same direction as Q0 if `reorient=True`

**Return type** tuple of 2D np.arrays

### Notes

The symmetry_axis keyword averages quadrants like this:

```
 +--------+--------+
 | Q1   * | *   Q0 |
 |   *    |    *   |
 |  *     |     *  |               cQ1 | cQ0
 +--------o--------+ --(output) -> ----o----
 |  *     |     *  |               cQ2 | cQ3
 |   *    |    *   |
 | Q2  *  | *   Q3 |          cQi == combined quadrants
 +--------+--------+

symmetry_axis = None - individual quadrants
symmetry_axis = 0 (vertical) - average Q0+Q1, and Q2+Q3
symmetry_axis = 1 (horizontal) - average Q1+Q2, and Q0+Q3
symmetry_axis = (0, 1) (both) - combine and average all 4 quadrants
```

The end results look like this:

```
(0) symmetry_axis = None

    returned image   Q1 | Q0
                    ----o----
                     Q2 | Q3

(1) symmetry_axis = 0

    Combine:  Q01 = Q0 + Q1, Q23 = Q2 + Q3
    returned image    Q01 | Q01
                     -----o-----
                      Q23 | Q23

(2) symmetry_axis = 1

    Combine: Q12 = Q1 + Q2, Q03 = Q0 + Q3
    returned image   Q12 | Q03
                    -----o-----
                     Q12 | Q03

(3) symmetry_axis = (0, 1)

    Combine all quadrants: Q = Q0 + Q1 + Q2 + Q3
    returned image   Q | Q
                    ---o---  all quadrants equivalent
                     Q | Q
```

`abel.tools.symmetry.`**`put_image_quadrants`**(*Q*, *original_image_shape*, *symmetry_axis=None*)

Reassemble image from 4 quadrants Q = (Q0, Q1, Q2, Q3) The reverse process to get_image_quadrants(reorient=True)

Note: the quadrants should all be oriented as Q0, the upper right quadrant

**Parameters**

- **Q** (*tuple of np.array (Q0, Q1, Q2, Q3)*) – Image quadrants all oriented as Q0 shape (`rows//2+rows%2, cols//2+cols%2`)

```
+--------+--------+
| Q1   * | *   Q0 |
|   *    |    *   |
|  *     |     *  |
+--------o--------+
|  *     |     *  |
|   *    |    *   |
| Q2  *  | *   Q3 |
+--------+--------+
```

- **original_image_shape** (*tuple*) – (rows, cols)

  reverses the padding added by *get_image_quadrants()* for odd-axis sizes

  odd row trims 1 row from Q1, Q0

  odd column trims 1 column from Q1, Q2

- **symmetry_axis** (*int or tuple*) –

  impose image symmetry

  ```
  symmetry_axis = 0 (vertical) - Q0 == Q1 and Q3 == Q2
  symmetry_axis = 1 (horizontal) - Q2 == Q1 and Q3 == Q0
  ```

**Returns**

**IM** –

Reassembled image of shape (rows, cols):

```
 symmetry_axis =

  None              0               1            (0,1)

 Q1 | Q0         Q1 | Q1         Q1 | Q0        Q1 | Q1
----o----   or  ----o----   or  ----o----   or ----o----
 Q2 | Q3         Q2 | Q2         Q1 | Q0        Q1 | Q1
```

**Return type** np.array

## 3.9 abel.tools.vmi module

`abel.tools.vmi.`**`angular_integration`**(*IM*, *origin=None*, *Jacobian=True*, *dr=1*, *dt=None*)

Angular integration of the image.

Returns the one-dimensional intensity profile as a function of the radial coordinate.

Note: the use of Jacobian=True applies the correct Jacobian for the integration of a 3D object in spherical coordinates.

**Parameters**

- **IM** (*2D numpy.array*) – The data image.
- **origin** (*tuple*) – Image center coordinate relative to *bottom-left* corner defaults to `rows//2, cols//2`.

- **Jacobian** (*boolean*) – Include $r \sin\theta$ in the angular sum (integration). Also, `Jacobian=True` is passed to `abel.tools.polar.reproject_image_into_polar()`, which includes another value of `r`, thus providing the appropriate total Jacobian of $r^2 \sin\theta$.
- **dr** (*float*) – Radial coordinate grid spacing, in pixels (default 1). *dr=0.5* may reduce pixel granularity of the speed profile.
- **dt** (*float*) – Theta coordinate grid spacing in radians. if `dt=None`, dt will be set such that the number of theta values is equal to the height of the image (which should typically ensure good sampling.)

**Returns**

- **r** (*1D numpy.array*) – radial coordinates
- **speeds** (*1D numpy.array*) – Integrated intensity array (vs radius).

`abel.tools.vmi.`**`average_radial_intensity`**(*IM*, ***kwargs*)

Calculate the average radial intensity of the image, averaged over all angles. This differs form `abel.tools.vmi.angular_integration()` only in that it returns the average intensity, and not the integrated intensity of a 3D image. It is equivalent to calling `abel.tools.vmi.angular_integration()` with *Jacobian=True* and then dividing the result by 2*pi.

**Parameters**

- **IM** (*2D numpy.array*) – The data image.
- **kwargs** – additional keyword arguments to be passed to `abel.tools.vmi.angular_integration()`

**Returns**

- **r** (*1D numpy.array*) – radial coordinates
- **intensity** (*1D numpy.array*) – one-dimensional intensity profile as a function of the radial coordinate.

`abel.tools.vmi.`**`radial_integration`**(*IM*, *radial_ranges=None*)

Intensity variation in the angular coordinate.

This function is the $\theta$-coordinate complement to `abel.tools.vmi.angular_integration()`

Evaluates intensity vs angle for defined radial ranges. Determines the anisotropy parameter for each radial range.

See *examples/example_PAD.py*

**Parameters**

- **IM** (*2D numpy.array*) – Image data
- **radial_ranges** (*list of tuple ranges or int step*) –

  **tuple** integration ranges `[(r0, r1), (r2, r3), ...]` evaluates the intensity vs angle for the radial ranges `r0_r1`, `r2_r3`, etc.

  **int** the whole radial range `(0, step), (step, 2*step), ..`

**Returns**

- **Beta** (*array of tuples*) – (beta0, error_beta_fit0), (beta1, error_beta_fit1), … corresponding to the radial ranges
- **Amplitude** (*array of tuples*) – (amp0, error_amp_fit0), (amp1, error_amp_fit1), … corresponding to the radial ranges

- **Rmidpt** (*numpy float 1d array*) – radial-mid point of each radial range
- **Intensity_vs_theta** (*2D numpy.array*) – Intensity vs angle distribution for each selected radial range.
- **theta** (*1D numpy.array*) – Angle coordinates, referenced to vertical direction.

`abel.tools.vmi.`**`anisotropy_parameter`**(*theta*, *intensity*, *theta_ranges=None*)

Evaluate anisotropy parameter $\beta$, for $I$ vs $\theta$ data.

$$I = \frac{\sigma_{\text{total}}}{4\pi}[1 + \beta P_2(\cos\theta)]$$

where $P_2(x) = \frac{3x^2-1}{2}$ is a 2nd order Legendre polynomial.

Cooper and Zare "Angular distribution of photoelectrons" J Chem Phys 48, 942-943 (1968)

**Parameters**

- **theta** (*1D numpy array*) – Angle coordinates, referenced to the vertical direction.
- **intensity** (*1D numpy array*) – Intensity variation with angle
- **theta_ranges** (*list of tuples*) – Angular ranges over which to fit `[(theta1, theta2), (theta3, theta4)]`. Allows data to be excluded from fit, default include all data

**Returns**

- **beta** (*tuple of floats*) – (anisotropy parameter, fit error)
- **amplitude** (*tuple of floats*) – (amplitude of signal, fit error)

`abel.tools.vmi.`**`toPES`**(*radial*, *intensity*, *energy_cal_factor*, *per_energy_scaling=True*, *photon_energy=None*, *Vrep=None*, *zoom=1*)

Convert speed radial coordinate into electron kinetic or electron binding energy. Return the photoelectron spectrum (PES).

This calculation uses a single scaling factor `energy_cal_factor` to convert the radial pixel coordinate into electron kinetic energy.

Additional experimental parameters: `photon_energy` will give the energy scale as electron binding energy, in the same energy units, while `Vrep`, the VMI lens repeller voltage (volts), provides for a voltage independent scaling factor. i.e. `energy_cal_factor` should remain approximately constant.

The `energy_cal_factor` is readily determined by comparing the generated energy scale with published spectra. e.g. for O photodetachment, the strongest fine-structure transition occurs at the electron affinity $EA = 11\,784.676(7)$ cm$^{-1}$. Values for the ANU experiment are given below, see also *examples/example_hansenlaw.py*.

**Parameters**

- **radial** (*numpy 1D array*) – radial coordinates.
- **intensity** (*numpy 1D array*) – intensity values, at the radial array.
- **energy_cal_factor** (*float*) – energy calibration factor that will convert radius squared into energy. The units affect the units of the output. e.g. inputs in eV/pixel$^2$, will give output energy units in eV. A value of $1.148427 \times 10^{-5}$ cm$^{-1}$/pixel$^2$ applies for "examples/data/O-ANU1024.txt" (with Vrep = -98 volts).
- **per_energy_scaling (*bool*) – sets the intensity Jacobian.** If *True*, the returned intensities correspond to an "intensity per eV" or "intensity per cm$^{-1}$ ". If *False*, the returned intensities correspond to an "intensity per pixel".

  Optional:

- **photon_energy** (*None or float*) – measurement photon energy. The output energy scale is then set to electron-binding-energy in units of *energy_cal_factor*. The conversion from wavelength (nm) to *photon_energy* (cm$^1$) is $10^7/\lambda$ (nm) e.g. *1.0e7/812.51* for "examples/data/O-ANU1024.txt".
- **Vrep** (*None or float*) – repeller voltage. Convenience parameter to allow the *energy_cal_factor* to remain constant, for different VMI lens repeller voltages. Defaults to *None*, in which case no extra scaling is applied. e.g. *-98 volts*, for "examples/data/O-ANU1024.txt".
- **zoom** (*float*) – additional scaling factor if the input experimental image has been zoomed. Default 1.

**Returns**

- **eKBE** (*numpy 1d-array of floats*) – energy scale for the photoelectron spectrum in units of *energy_cal_factor*. Note that the data is no-longer on a uniform grid.
- **PES** (*numpy 1d-array of floats*) – the photoelectron spectrum, scaled according to the *per_energy_scaling* input parameter.

**class** abel.tools.vmi.**Distributions**(*origin=u'center'*, *rmax=u'MIN'*, *order=2*, *use_sin=True*, *weights=None*, *method=u'linear'*)

Bases: `object`

Class for calculating various radial distributions.

Objects of this class hold the analysis parameters and cache some intermediate computations that do not depend on the image data. Multiple images can be analyzed (using the same parameters) by feeding them to the object:

```
distr = Distributions(parameters)
results1 = distr(image1)
results2 = distr(image2)
```

If analyses with different parameters are required, multiple objects can be used. For example, to analyze 4 quadrants independently:

```
distr0 = Distributions('ll', ...)
distr1 = Distributions('lr', ...)
distr2 = Distributions('ur', ...)
distr3 = Distributions('ul', ...)

for image in images:
    Q0, Q1, Q2, Q3 = ...
    res0 = distr0(Q0)
    res1 = distr1(Q1)
    res2 = distr2(Q2)
    res3 = distr3(Q3)
```

However, if all the quadrants have the same dimensions, it is more memory-efficient to flip them all to the same orientation and use a single object:

```
distr = Distributions('ll', ...)

for image in images:
    Q0, Q1, Q2, Q3 = ...
    res0 = distr(Q0)
    res1 = distr(Q1[:, ::-1])  # or np.fliplr
    res2 = distr(Q2[::-1, ::-1])  # or np.flip(Q2, (0, 1))
    res3 = distr(Q3[::-1, :])  # or np.flipud
```

More concise function to calculate distributions for single images (without caching) are also available, see `harmonics()`, `Ibeta()` below.

**Parameters**

- **origin** (*tuple of int or str*) – origin of the radial distributions (the pole of polar coordinates) within the image.

  **`(int, int)`:** explicit row and column indices

  **`str`:** location string specifying the vertical and horizontal positions (in this order!) using the words from the following diagram:

  ```
                 left             center              right

     top/upper   [0, 0]---------[0, n//2]--------[0, n-1]
                 |                                       |
                 |                                       |
        center   [m//2, 0]     [m//2, n//2]     [m//2, n-1]
                 |                                       |
                 |                                       |
  bottom/lower   [m-1, 0]------[m-1, n//2]-----[m-1, n-1]
  ```

  The words can be abbreviated to their first letter each (such as `'top left'` → `'tl'`, the space is then not required).

  `'center center'`/`'cc'` can also be shortened to `'center'`/`'c'`.

  Examples:

  `'center'` or `'cc'` (default) for the full centered image

  `'center left'`/`'cl'` for the right image half, vertically centered

  `'bottom left'`/`'bl'` or `'lower left'`/`'ll'` for the upper-right image quadrant

- **rmax** (*int or str*) – largest radius to include in the distributions

  **`int`:** explicit value

  **`'hor'`:** fitting inside horizontally

  **`'ver'`:** fitting inside vertically

  **`'HOR'`:** touching horizontally

  **`'VER'`:** touching vertically

  **`'min'`:** minimum of `'hor'` and `'ver'`, the largest area with 4 full quadrants

  **`'max'`:** maximum of `'hor'` and `'ver'`, the largest area with 2 full quadrants

  **`'MIN'` (default):** minimum of `'HOR'` and `'VER'`, the largest area with 1 full quadrant (thus the largest with the full 90° angular range)

  **`'MAX'`:** maximum of `'HOR'` and `'VER'`

  **`'all'`:** covering all pixels (might have huge errors at large *r*, since the angular dependences must be inferred from very small available angular ranges)

- **order** (*int*) – highest order in the angular distributions. Even number  0.

- **use_sin** (*bool*) – use $|\sin\theta|$ weighting. This is the weight implied in spherical integration (for the total intensity, for example) and with respect to which the Legendre polynomials are orthogonal, so using it in the fitting procedure gives the most reasonable results even if the data deviates form the assumed angular behavior. It also reduces contributions from the centerline noise.

- **weights** (*m* × *n numpy array, optional*) – in addition to the optional $|\sin\theta|$ weighting (see **use_sin** above), use given weights for each pixel. The array shape must match the image shape.

  Parts of the image can be excluded from the fitting by assigning zero weights to their pixels.

  (Note: if `use_sin=False`, a reference to this array is cached instead of its content, so if you modify the array between creating the object and using it, the results will be surprising. However, if needed, you can pass a copy as `weights=weights.copy()`.)

- **method** (*str*) – numerical integration method used in the fitting procedure

  **`'nearest'`:** each pixel of the image is assigned to the nearest radial bin. The fastest, but noisier (especially for high orders).

  **`'linear'` (default):** each pixel of the image is linearly distributed over the two adjacent radial bins. About twice slower than `'nearest'`, but smoother.

  **`'remap'`:** the image is resampled to a uniform polar grid, then polar pixels are summed over all angles for each radius. The smoothest, but significantly slower and might have problems with **rmax** > `'MIN'` and discontinuous weights.

**class Results**(*r*, *cn*)

Bases: `object`

Class for holding the results of image analysis.

*`Distributions.image()`* returns an object of this class, from which various distributions can be retrieved using the methods described below, for example:

```
distr = Distributions(...)
res = distr(IM)
harmonics = res.harmonics()
```

All distributions are returned as 2D arrays with the rows (1st index) corresponding to particular terms of the expansion and the columns (2nd index) corresponding to the radii. The terms can be easily separated like `I, beta2, beta4 = res.Ibeta()`. Python 3 users can also collect all $\beta$ parameters as `I, *beta = res.Ibeta()` for any **order**. Alternatively, transposing the results as `Ibeta = res.Ibeta().T` allows accessing all terms $\big(I(r), \beta_2(r), \beta_4(r), \dots\big)$ at particular radius *r* as `Ibeta[r]`.

**r**

radii from 0 to **rmax**

**Type** numpy array

**cos**()

Radial distributions of $\cos^n\theta$ terms (0 $n$ **order**).

(You probably do not need them.)

**Returns cosn** – radial dependences of the $\cos^n\theta$ terms, ordered from the lowest to the highest power

**Return type** (# terms) × (rmax + 1) numpy array

**rcos**()

Same as *`cos()`*, but prepended with the radii row.

**cossin**()

Radial distributions of $\cos^n\theta \cdot \sin^m\theta$ terms ($n + m$ = **order**).

For **order** = 0:
  $\cos^0\theta$ is the total intensity.
For **order** = 2
  $\cos^2\theta$ corresponds to "parallel" () transitions,

  $\sin^2\theta$ corresponds to "perpendicular" () transitions.
For **order** = 4
  $\cos^4\theta$ corresponds to ,,

  $\cos^2\theta\cdot\sin^2\theta$ corresponds to , and ,.

  $\sin^4\theta$ corresponds to ,.
And so on.

Notice that higher orders can represent lower orders as well:
  $\cos^2\theta+\sin^2\theta=\cos^0\theta$ ( + = 1),

  $\cos^4\theta+\cos^2\theta\cdot\sin^2\theta=\cos^2\theta$ (, + , = , + , = ),

  $\cos^2\theta\cdot\sin^2\theta+\sin^4\theta=\sin^2\theta$ (, + , = , + , = ),

  and so forth.

  **Returns** **cosnsinm** – radial dependences of the $\cos^n\theta{\cdot}\sin^m\theta$ terms, ordered from the highest $\cos\theta$ power to the highest $\sin\theta$ power
  **Return type** (# terms) × (rmax + 1) numpy array

**rcossin**()
  Same as `cossin()`, but prepended with the radii row.

**harmonics**()
  Radial distributions of spherical harmonics (Legendre polynomials $P_n(\cos\theta)$).

  Spherical harmonics are orthogonal with respect to integration over the full sphere:

$$\iint P_n P_m\,d\Omega=\int_0^{2\pi}\int_0^{\pi}P_n(\cos\theta)P_m(\cos\theta)\,\sin\theta d\theta\,d\varphi=0$$

  for *n m*; and $P_0(\cos\theta)$ is the spherically averaged intensity.
  **Returns** **Pn** – radial dependences of the $P_n(\cos\theta)$ terms
  **Return type** (# terms) × (rmax + 1) numpy array

**rharmonics**()
  Same as `harmonics()`, but prepended with the radii row.

**Ibeta**(*window=1*)
  Radial intensity and anisotropy distributions.

  A cylindrically symmetric 3D intensity distribution can be expanded over spherical harmonics (Legendre polynomials $P_n(\cos\theta)$) as

$$I(r,\theta,\varphi)\,d\Omega=\frac{1}{4\pi}I(r)\big[1+\beta_2(r)P_2(\cos\theta)+\beta_4(r)P_4(\cos\theta)+\ldots\big],$$

  where $I(r)$ is the "radial intensity distribution" integrated over the full sphere:

$$I(r)=\int_0^{2\pi}\int_0^{\pi}I(r,\theta,\varphi)\,r^2\sin\theta d\theta\,d\varphi,$$

  and $\beta_n(r)$ are the dimensionless "anisotropy parameters" describing relative contributions of each harmonic order ($\beta_0(r)=1$ by definition). In particular:

$\beta_2 = 2$ for the $\cos^2\theta$ () angular distribution,

$\beta_2 = 0$ for the isotropic distribution,

$\beta_2 = -1$ for the $\sin^2\theta$ () angular distribution.

The radial intensity distribution alone for data with arbitrary angular variations can be obtained by using `weight='sin'` and `order=0`.

**Parameters** **window** (*int*) – window size in pixels for radial averaging of $\beta$. Since anisotropy parameters are non-linear, the central moving average is applied to the harmonics (which are linear), and then $\beta$ is calculated from them. In case of well separated peaks, setting **window** to the peak width will result in $\beta$ values at peak centers equal to total peak anisotropies (beware of the background, however).

**Returns** **Ibeta** – radial intensity distribution (0-th term) and radial dependences of anisotropy parameters (other terms)

**Return type** (# terms) × (rmax + 1) numpy array

**`rIbeta`**(*window=1*)

Same as `Ibeta()`, but prepended with the radii row.

**`image`**(*IM*)

Analyze an image.

This method can be also conveniently accessed by “calling” the object itself:

```
distr = Distributions(...)
Ibeta = distr(IM).Ibeta()
```

**Parameters** **IM** (*m × n numpy array*) – the image to analyze

**Returns** **results** – the object with analysis results, from which various distributions can be retrieved, see `Results`

**Return type** Distributions.Results object

`abel.tools.vmi.`**`harmonics`**(*IM, origin=u'cc', rmax=u'MIN', order=2, **kwargs*)

Convenience function to calculate harmonic distributions for a single image. Equivalent to `Distributions(...).image(IM).harmonics()`.

Notice that this function does not cache intermediate calculations, so using it to process multiple images is several times slower than through a `Distributions` object.

`abel.tools.vmi.`**`rharmonics`**(*IM, origin=u'cc', rmax=u'MIN', order=2, **kwargs*)

Same as `harmonics()`, but prepended with the radii row.

`abel.tools.vmi.`**`Ibeta`**(*IM, origin=u'cc', rmax=u'MIN', order=2, window=1, **kwargs*)

Convenience function to calculate radial intensity and anisotropy distributions for a single image. Equivalent to `Distributions(...).image(IM).Ibeta(window)`.

Notice that this function does not cache intermediate calculations, so using it to process multiple images is several times slower than through a `Distributions` object.

`abel.tools.vmi.`**`rIbeta`**(*IM, origin=u'cc', rmax=u'MIN', order=2, window=1, **kwargs*)

Same as `Ibeta()`, but prepended with the radii row.

## 3.10 abel.benchmark module

**class** `abel.benchmark.`**`Timent`**(*skip=0, repeat=1, duration=0.0*)

Bases: `object`

Helper class for measuring execution times.

The constructor only initializes the timing-procedure parameters. Use the `time()` method to run it for particular functions.

**Parameters**

- **skip** (*int*) – number of "warm-up" iterations to perform before the measurements. Can be specified as a negative number, then `abs(skip)` "warm-up" iterations are performed, but if this took more than **duration** seconds, they are accounted towards the measured iterations.
- **repeat** (*int*) – minimal number of measured iterations to perform. Must be positive.
- **duration** (*float*) – minimal duration (in seconds) of the measurements.

**time**(*func*, **args*, ***kwargs*)

Repeatedly executes a function at least **repeat** times and for at least **duration** seconds (see above), then returns the average time per iteration. The actual number of measured iterations can be retrieved from `Timent.count`.

**Parameters**

- **func** (*callable*) – function to execute
- ***args, **kwargs** (*any, optional*) – parameters to pass to **func**

**Returns** average function execution time

**Return type** float

## Notes

The measurements overhead can be estimated by executing

```
Timent(...).time(lambda: None)
```

with a sufficiently large number of iterations (to avoid rounding errors due to the finite timer precision). In 2018, this overhead was on the order of 100 ns per iteration.

**class** abel.benchmark.**AbelTiming**(*n=[301, 501], select=u'all', repeat=1, t_min=0.1, t_max=inf, verbose=True*)

Bases: `object`

Benchmark performance of different Abel implementations (basis generation, forward and inverse transforms, as applicable).

**Parameters**

- **n** (*int or sequence of int*) – array size(s) for the benchmark (assuming 2D square arrays (*n*, *n*))
- **select** (*str or sequence of str*) – methods to benchmark. Use `'all'` (default) for all available or choose any combination of individual methods:

```
select=['basex', 'direct_C', 'direct_Python', 'hansenlaw',
        'linbasex', 'onion_bordas, 'onion_peeling', 'two_point',
        'three_point']
```

- **repeat** (*int*) – repeat each benchmark at least this number of times to get the average values
- **t_min** (*float*) – repeat each benchmark for at least this number of seconds to get the average values

- **t_max** (*float*) – do not benchmark methods at array sizes when this is expected to take longer than this number of seconds. Notice that benchmarks for the smallest size from **n** are always run and that the estimations can be off by a factor of 2 or so.
- **verbose** (*boolean*) – determines whether benchmark progress should be reported (to stderr)

**`n`**

array sizes from the parameter **n**, sorted in ascending order

**Type** list of int

**`bs, fabel, iabel`**

benchmark results — dictionaries for

**bs** basis-set generation

**fabel** forward Abel transform

**iabel** inverse Abel transform

with methods as keys and lists of timings in milliseconds as entries. Timings correspond to array sizes in *`AbelTiming.n`*; for skipped benchmarks (see **t_max**) they are `np.nan`.

**Type** dict of list of float

### Notes

The results can be output in a nice format by simply `print(AbelTiming(...))`.

Keep in mind that most methods have $O(n^2)$ memory and $O(n^3)$ time complexity, so going from $n = 501$ to $n = 5001$ would require about 100 times more memory and take about 1000 times longer.

**class** `abel.benchmark.`**`DistributionsTiming`**(*n=[301, 501], shape=u'half', rmax=u'MIN', order=2, weight=[u'none', u'sin', u'sin+array'], method=u'all', repeat=1, t_min=0.1*)

Bases: `object`

Benchmark performance of different VMI distributions implementations.

**Parameters**

- **n** (*int or sequence of int*) – array size(s) for the benchmark (assuming full images to be 2D square arrays ($n$, $n$))
- **shape** (*str*) – image shape:

  **`'Q'`:** one quadrant (($n$ + 1)/2, ($n$ + 1)/2)

  **`'half'` (default):** half image ($n$, ($n$ + 1)/2), vertically centered

  **`'full'`:** full image ($n$, $n$), centered
- **rmax** (*str or sequence of str*) – `'MIN'` (default) and/or `'all'`, see **rmax** in *`abel.tools.vmi.Distributions`*
- **order** (*int*) – highest order in the angular distributions. Even number 0.
- **weight** (*str or sequence of str*) – weighting to test. Use `'all'` for all available or choose any combination of individual types:

```
weight=['none', 'sin', 'array', 'sin+array']
```

- **method** (*str or sequence of str*) – methods to benchmark. Use `'all'` (default) for all available or choose any combination of individual methods:

```
method=['nearest', 'linear', 'remap']
```

- **repeat** (*int*) – repeat each benchmark at least this number of times to get the average values
- **t_min** (*float*) – repeat each benchmark for at least this number of seconds to get the average values

**n**

array sizes from the parameter **n**

**Type** list of int

**results**

benchmark results — multi-level dictionary, in which `results[method][rmax][weight]` is the list of timings in milliseconds corresponding to array sizes in `DistributionsTiming.n`. Each timing is a tuple $(t_1, t_\infty)$ with $t_1$ corresponding to single-image (non-cached) performance, and $t_\infty$ corresponding to batch (cached) performance.

**Type** dict of dict of dict of list of tuple of float

### Notes

The results can be output in a nice format by simply `print(DistributionsTiming(...))`.

`abel.benchmark.`**`is_symmetric`**(*arr*, *i_sym=True*, *j_sym=True*)

Takes in an array of shape (n, m) and check if it is symmetric

**Parameters**

- **arr** (*1D or 2D array*)
- **i_sym** (*array*) – symmetric with respect to the 1st axis
- **j_sym** (*array*) – symmetric with respect to the 2nd axis

**Returns**

- *a binary array with the symmetry condition for the corresponding quadrants.*
- *The globa*

### Notes

If both **i_sym** = `True` and **j_sym** = `True`, the input array is checked for polar symmetry.

See https://github.com/PyAbel/PyAbel/issues/34#issuecomment-160344809 for the defintion of a center of the image.

`abel.benchmark.`**`absolute_ratio_benchmark`**(*analytical*, *recon*, *kind=u'inverse'*)

Check the absolute ratio between an analytical function and the result of a inverse Abel reconstruction.

**Parameters**

- **analytical** (*one of the classes from analytical, initialized*)
- **recon** (*1D ndarray*) – a reconstruction (i.e. inverse abel) given by some PyAbel implementation

CHAPTER 4

# Transform Methods

The numerical Abel transform is computationally intensive, and a basic numerical integration of the analytical equations does not reliably converge. Consequently, numerous algorithms have been developed in order to approximate the Abel transform in a reliable and efficient manner. So far, PyAbel includes the following transform methods:

1. ⋆ The *basex* method of Dribinski and co-workers, which uses a Gaussian basis set to provide a quick, robust transform. This is one of the de facto standard methods in photoelectron/photoion spectroscopy.
2. The *hansenlaw* recursive method of Hansen and Law, which provides an extremely fast transform with low centerline noise.
3. The *direct* numerical integration of the analytical Abel transform equations, which is implemented in Cython for efficiency. In general, while the forward Abel transform is useful, the inverse Abel transform requires very fine sampling of features (lots of pixels in the image) for good convergence to the analytical result, and is included mainly for completeness and for comparison purposes. For the inverse Abel transform, other methods are generally more reliable.
4. ⋆ The *three_point* method of Dasch and co-workers, which provides a fast and robust transform by exploiting the observation that underlying radial distribution is primarily determined from changes in the line-of-sight projection data in the neighborhood of each radial data point. This technique works very well in cases where the real difference between adjacent projections is much greater than the noise in the projections (i.e. where the raw data is not oversampled).
5. ⋆ The *two_point* method is also well described by Dasch. It is a simpler approximation to the *three point* transform. Computationally, very efficient in Python.
6. ⋆ The *onion_peeling* onion-peeling deconvolution method described by Dash is one of the simpler, and faster inversion methods. The article states the onion-peeling deconvolution is similar to the two point Abel. Both methods have less smoothing than the other methods examined by Dasch.
7. The *onion_bordas* onion-peeling method of Bordas et al. is based on the MatLab code of Rallis and Wells *et al.* The article claims "the method works properly only in the limit of large electrostatic energy to initial kinetic energy ratio and gives qualitatively the same results as a standard inversion method".

8. ⋆ The *linbasex* 1D-spherical basis method of Gerber et al. evaluates 1D projections of velocity-map images in terms of 1D projections of spherical functions. The results produce directly the coefficients of the involved spherical functions, making the reconstruction of sliced Newton spheres obsolete.
9. (Planned implementation) The *Fourier–Hankel* method, which is computationally efficient, but contains significant centerline noise and is known to introduce artifacts.
10. (Planned implementation) The *POP* (polar onion peeling) method. POP projects the image onto a basis set of Legendre polynomial-based functions, which can greatly reduce the noise in the reconstruction. However, this method only applies to images that contain features at constant radii. I.e., it works for the spherical shells seen in photoelectron/ion spectra, but not for flames.

⋆ Methods marked with an asterisk require the generation of basis sets. The first time each method is run for a specific image size, a basis set must be generated, which can take several seconds or minutes. However, this basis set is saved to disk (generally to the current directory) and can be reused, making subsequent transforms very efficient. Users who are transforming numerous images using these methods will want to keep this in mind and specify the directory containing the basis sets.

Contents:

## 4.1 Comparison of Abel Transform Methods

### 4.1.1 Introduction

Each new Abel transform method claims to be the best, providing a lower-noise, more accurate transform. The point of PyAbel is to provide an easy platform to try several abel transform methods and determine which provides the best results for a specific dataset.

So far, we have found that for a high-quality dataset, all of the transform methods produce good results.

### 4.1.2 Speed benchmarks

The `abel.benchmark.AbelTiming` class provides the ability to benchmark the relative speed of the Abel transform algorithms.

### 4.1.3 Transform quality

. . . coming soon! . . .

## 4.2 BASEX

### 4.2.1 Introduction

The BASEX ("basis set expansion") Abel-transform method utilizes well-behaved functions (i.e., functions that have a known analytic Abel transform) to transform images. In the current iteration of PyAbel, these functions (called basis functions) are Gaussian-like functions, following the original description of the method, developed in 2002 at USC and UC Irvine by Dribinski, Ossadtchi, Mandelshtam, and Reisler [Dribinski2002].

### 4.2.2 How it works

This method is based on expressing line-of-sight projection images (`raw_data`) as sums of functions that have known analytic Abel inverses. The provided raw images are expanded in a basis set composed of these basis functions, with the expansion coefficients determined through a least-squares fitting process. These coefficients are then applied to the (known) analytic inverse of these basis functions, which directly provides the Abel inverse of the raw images. Thus, the transform can be completed using simple linear algebra.

In the current iteration of PyAbel, these basis functions are Gaussian-like (see equations (14) and (15) in [Dribinski2002]). The process of evaluating these functions is computationally intensive, and the basis-set generation process can take several seconds to minutes for larger images (larger than ~1000×1000 pixels). However, once calculated, these basis sets can be reused, and are therefore stored on disk and loaded quickly for future use. The transform then proceeds very quickly, since each raw-image Abel inversion is a simple matrix multiplication.

### 4.2.3 When to use it

According to Dribinski et al., BASEX has several advantages:

1. For synthetic noise-free projections, BASEX reconstructs an essentially exact and artifact-free image, eschewing the need for interpolation procedures, which may introduce additional errors or assumptions.
2. BASEX is computationally cheap and only requires matrix multiplication, once the basis sets have been generated and saved to disk.
3. The current basis set is composed of the Gaussian-like functions, which are highly localized, uniform in coverage, and sufficiently narrow. This allows resolution of very sharp features in the raw data. Moreover, the reconstruction procedure does not contribute to noise in the reconstructed image; noise appears in the image only when it exists in the projection.
4. Resolution of images reconstructed with BASEX is superior to those obtained with the Fourier–Hankel method, particularly for noisy projections. However, to obtain maximal resolution, it is important to properly center the projections prior to transforming with BASEX.
5. BASEX-reconstructed images have an exact analytical expression, which allows an analytical high-resolution calculation of the speed distribution, without increasing computation time. (This is not yet implemented in PyAbel.)

### 4.2.4 How to use it

The recommended way to complete the inverse Abel transform using the BASEX algorithm for a full image is to use the *`abel.transform.Transform`* class:

```
abel.transform.Transform(raw_image, method='basex', direction='inverse').transform
```

The additional BASEX parameters are described in *`abel.basex.basex_transform()`* an can be passed to `Transform()` using the `transform_options` argument.

If you would like to access the BASEX algorithm directly (to transform a right-side half-image), you can use *`abel.basex.basex_transform()`*.

The behavior of the original *BASEX.exe* program by Karpichev with top–bottom symmetry and the "narrow" basis set can be reproduced as follows:

```
rescale = math.sqrt(math.pi) / 2

raw_image = <centered raw image>
```

(continues on next page)

(continued from previous page)

```
reg = <regularization parameter>
reconst = abel.Transform(raw_image, direction='inverse', symmetry_axis=(0, 1),
                         method='basex', transform_options=dict(
                             reg=reg*(rescale**2), correction=False
                        )).transform.clip(min=0) * rescale
```

(The `rescale` factor accounts for the wrong factor used in the *BASEX.exe* program for the basis projections, see *BASEX: computational details*.)

### 4.2.5 PyAbel improvements

- As noted above, the BASEX method implementation in PyAbel uses correct expressions for the basis projections, so unlike *BASEX.exe*, it is consistent with the original method description in [Dribinski2002] and with other methods implemented in PyAbel.
- Basis sets for any image size are generated automatically.
- Basis functions with any width parameter $\sigma$ (specified by the `sigma` parameter) can be used. They are $\rho_k(r) \approx \exp[-2(r/\sigma - k)^2]$, so their $1/e^2$ width is $2\sigma$, and the full width at half-maximum (FWHM) is $\sqrt{2\ln 2}\,\sigma \approx 1.18\,\sigma$. The spacing between the maxima of the adjacent basis functions is $\sigma$, which automatically determines the number of basis functions.
- An automatic intensity correction is available (enabled by default) for reducing the artifacts caused by the basis-functions shape and the sampling of their projections, as well as the intensity drop (especially near the axis) introduced by Tikhonov regularization.
- The forward Abel transform is also implemented, using the same method but swapping the basis functions and their projections.

Some additional information on the implementation is given in *BASEX: computational details*.

#### BASEX: computational details

To complement the general description given in the BASEX article, here we provide the full derivation of the basis projections and the details needed for their efficient computation. The differences in the PyAbel implementation of the method are also discussed below.

#### Basis projections

The basis functions are

$$\rho_k(r) = (e/k^2)^{k^2} (r/\sigma)^{2k^2} e^{-(r/\sigma)^2},$$

or in a reduced form,

$$\rho_k(u) = A_k\, u^{2k^2} e^{-u^2},$$

$$A_k = (e/k^2)^{k^2}, \quad u = r/\sigma.$$

Their Abel transform is most easily obtained by considering the projection in rectangular coordinates:

$$\chi_k(x) = \int_{-\infty}^{\infty} \rho_k(r)\,dy = 2\int_0^{\infty} \rho_k(r)\,dy,$$

$$r = \sqrt{x^2 + y^2}.$$

Then

$$\int_0^\infty \left(\sqrt{x^2+y^2}\right)^{2k^2} e^{-\left(x^2+y^2\right)}\,dy = \int_0^\infty \left(x^2+y^2\right)^{k^2} e^{-x^2} e^{-y^2}\,dy.$$

After expanding the binomial $\left(x^2+y^2\right)^{k^2}$, this integral becomes

$$e^{-x^2} \sum_{l=0}^{k^2} \binom{k^2}{l} x^{2l} \int_0^\infty y^{2\left(k^2-l\right)} e^{-y^2}\,dy,$$

where the binomial coefficients

$$\binom{k^2}{l} = \frac{k^2!}{l!\,(k^2-l)!} = \frac{\Gamma(k^2+1)}{\Gamma(l+1)\,\Gamma(k^2-l+1)},$$

and the integrals are also expressed through the gamma function:

$$\int_0^\infty y^{2\left(k^2-l\right)} e^{-y^2}\,dy \overset{t=y^2}{=} \int_0^\infty t^{k^2-l} e^{-t} \frac{1}{2\sqrt{t}}\,dt = \frac{1}{2}\Gamma\left(k^2 - l + \frac{1}{2}\right).$$

The complete expression for the projections (in a reduced form, $u = x/\sigma$) is thus

$$\chi_k(u) = A_k \sigma e^{-u^2} \sum_{l=0}^{k^2} \frac{\Gamma(k^2+1)\,\Gamma\left(k^2-l+\frac{1}{2}\right)}{\Gamma(l+1)\,\Gamma(k^2-l+1)} u^{2l}.$$

The case $k = 0$ is special, since formally $A_0 = (e/0)^0$, which is undefined. However, taking the limit $k \to 0$, we obtain

$$\rho_0(u) = e^{-u^2},$$

the Abel transform of which is simply

$$\chi_0(u) = \sqrt{\pi}\,\sigma e^{-u^2}.$$

**Note:** The original MATLAB implementation by Dribinski used an incorrect prefactor "2" instead if "$\sqrt{\pi}$" in calculations of the basis projections $\chi_k$ (in the above expression the $\sqrt{\pi}$ factor for $k > 0$ is invisibly present in the $\Gamma(\ldots + 1/2)$ terms). The *BASEX.exe* program by Karpichev also uses these MATLAB-generated basis sets and has the same problem, producing intensities off by a factor of $\sqrt{\pi}/2$ and applying regularization with a strength off by a square of that factor.

We use the correct expressions for all calculations.

## Computations

The above expressions for $\rho_k(u)$ and $\chi_k(u)$ involve very small ($e^{-u^2}$) and very large ($u^{2k^2}$) numbers and thus will cause floating-point underflow/overflow if computed directly. However, they can be recast as

$$\rho_k(u) = \exp\left[\left(1 - \ln k^2\right) k^2 + \ln u \cdot 2k^2 - u^2\right],$$

$$\begin{aligned}\chi_k(u) = \sigma \sum_{l=0}^{k^2} \exp\Big[ &\left(1 - \ln k^2\right) k^2 - u^2 + \\ &+ \ln\Gamma(k^2+1) + \ln\Gamma\left(k^2 - l + \frac{1}{2}\right) - \\ &- \ln\Gamma(l+1) - \ln\Gamma(k^2 - l + 1) + \\ &+ \ln u \cdot 2l \Big],\end{aligned}$$

in which all terms are comparable to $k^2$ and $u^2$. In particular, $\ln\Gamma(z) \sim (\ln z - 1)z$ and is available directly as `scipy.special.gammaln()`.

The $\ln\Gamma(z)$ functions are relatively computationally expensive, but as can be seen, computing the projections $\chi_k(u)$ for all $k$ up to $K$ requires only the values of $\ln\Gamma(n)$ and $\Delta\ln\Gamma(n) = \ln\Gamma(n) - \ln\Gamma(n - 1/2)$ for integers $n = 1, \dots, K^2 + 1$. They are precomputed and cached before the basis generation. This requires $O(K^2)$ extra memory (comparable to $O(NK)$ for the basis matrices themselves), but saves $O(NK^2)$ evaluations (see below) of these special functions.

The BASEX article mentions that actually "only a few terms contribute to the sum", but does not give any quantitative estimations. In order to obtain the practical constraints on the summation index, consider how the exponential terms change with $l$ at fixed $k$ and $u$:

$$\exp[\dots] = \exp f_{k,u} \cdot \exp g_{k,u}(l),$$

where

$$f_{k,u} = \left(1 - \ln k^2\right) k^2 - u^2 + \ln\Gamma(k^2+1)$$

does not depend on $l$, and

$$\begin{aligned}g_{k,u}(l) &= -\underbrace{\ln\Gamma(l+1)}_{\approx(\ln l - 1)l} - \underbrace{\Delta\ln\Gamma(k^2 - l + 1)}_{\approx\ln(k^2 - l)/2} + \ln u \cdot 2l = \\ &= (1 + \ln u^2 - \ln l)l + o(l).\end{aligned}$$

The last expression ($g$ without sublinear terms) reaches its maximum at $l_{\text{max}} = u^2$ and behaves near it as

$$g_{k,u}(l_{\text{max}} + \delta) = u^2 - \frac{\delta^2}{2u^2} + o(\delta^2).$$

From the practical perspective, the terms

$$\exp g_{k,u}(l) < \varepsilon_{\text{FP}} \cdot \exp g_{k,u}(l_{\text{max}}),$$

where $\varepsilon_{\text{FP}} \sim 10^{-16}$ is the floating-point precision, will be lost in rounding errors and thus do not need to be computed. This inequality can be transformed into

$$g_{k,u}(l) - g_{k,u}(l_{\text{max}}) = -\frac{\delta^2}{2u^2} < \ln\varepsilon_{\text{FP}},$$

from which

$$\delta > \sqrt{-2\ln\varepsilon_{\text{FP}}}\, u \approx 8.6\, u.$$

That is, the projections $\chi_k(u)$ can be computed to within the floating-point precision by summing only the terms with $l \in [l_{\text{max}} - \delta, l_{\text{max}} + \delta]$, where $l_{\text{max}} = u^2$ and $\delta = 9\,u$.

Since $\max u = K$, the total time complexity of computing $K$ basis projections at $N$ points is $O(NK^2)$.

### Intensity correction

The Gaussian-like BASEX basis functions do not sum to unity:

so they cannot describe a flat distribution, and for $\sigma \neq 1$ these intensity oscillations are visible in the reconstructed distributions. In addition, the basis projections are sampled only at pixel centers, which does not satisfy the requirements of the sampling theorem for their adequate representation. In particular, this leads to a reconstructed-intensity bias in the most useful $\sigma = 1$ case.

Moreover, the $k = 0$ basis function is broader than the $k > 0$ functions, and $\rho_k(r = 0) = 0$ for all $k > 0$, whereas $\rho_k(r \neq 0) \neq 0$. In other words, the region near the symmetry axis is treated quite differently from the rest of the image, which leads to an artifact near $r = 0$ in the reconstructed distributions.

Another problem arises when Tikhonov regularization is applied. Since it includes the norm of the solution in its minimization criterion, this generally leads to some intensity drop in the reconstructed distributions, especially near the symmetry axis.

In order to reduce these problems, PyAbel can use an automatic "intensity correction". It is based on the linearity of the transform and uses a "calibration" distribution with a known analytical Abel transform.

Specifically, a flat distribution (with a soft edge, to avoid ringing artifacts near the image boundary) and its analytical Abel transform are generated. Then the BASEX transform with the desired parameters is applied to that Abel transform, what should reconstruct the initial flat distribution, but actually includes the artifacts described above. The ratio of the desired flat distribution to this BASEX result is then taken as the intensity correction profile and is applied to the BASEX transform of the actual data.

Although this correction procedure does not reproduce analytical results for *all* distributions (except the calibration distribution itself), it greatly reduces the method artifacts in most cases.

### Vertical transform

(See this discussion about notation and details of the original implementation.)

Besides the horizontal transform that realizes the inverse Abel transform, the BASEX article and the *BASEX.exe* program also apply a vertical transform to the data. It is performed by multiplying the data by $\mathbf{B}$ in equation (13) to obtain the expansion coefficients and then multiplying these coefficients by $\mathbf{Z}$ in equation (9) to obtain the reconstructed image.

However, regularization is never applied to the vertical transform ($q_2^2 = 0$), so when $\mathbf{Z}$ has full rank ($\sigma = 1$, the "narrow" basis set in *BASEX.exe*), the overall vertical transform is

$$\mathbf{B}\mathbf{Z} = \mathbf{Z}^\mathrm{T}\left(\mathbf{Z}\mathbf{Z}^\mathrm{T}\right)^{-1}\mathbf{Z} = \mathbf{I},$$

that is, an identity transform, having no effect on the final results.

When $\mathbf{Z}$ is not of full rank, for example, for the "broad" basis set ($\sigma = 2$), the transform is no longer an identity, but actually has some undesirable properties.

First, it is not strictly translationally invariant (see the plot of the basis functions above) and thus is in fact not applied by the *BASEX.exe* program when "Line-by-line reconstruction" is chosen.

Second, far from the edges this transform is close to a convolution with the following functions:

so, in addition to the possibly useful vertical smoothing, it also introduces noticeable ringing artifacts.

Therefore in the PyAbel BASEX implementation we never apply the vertical transform. If the vertical smoothing for $\sigma > 1$ is desirable, it can be achieved by applying a vertical Gaussian blur to the transformed image.

The behavior of the original *BASEX.exe* program with top–bottom symmetry and the "broad" basis set can be reproduced by replacing the line

```
return rawdata.dot(A)
```

in *abel.basex.basex_core_transform()* with the following code:

```
Mc = (_bs[1])[::-1]  # PyAbel and BASEX.exe use different coordinates
V = Mc.dot(inv((Mc.T).dot(Mc))).dot(Mc.T)
return V.dot(rawdata).dot(A)
```

and using the code example from BASEX/*How to use it* with a additional `sigma=2` parameter in `transform_options`.

### 4.2.6 Citation

## 4.3 Direct

### 4.3.1 Introduction

This method attempts a direct integration of the Abel transform integral. It makes no assumptions about the data (apart from cylindrical symmetry), but it typically requires fine sampling to converge. Such methods are typically inefficient, but thanks to this Cython implementation (by Roman Yurchuk), this 'direct' method is competitive with the other methods.

### 4.3.2 How it works

Information about the algorithm and the numerical optimizations is contained in PR #52

### 4.3.3 When to use it

When a robust forward transform is required, this method works quite well. It is not typically recommended for the inverse transform, but it can work well for smooth functions that are finely sampled.

### 4.3.4 How to use it

To complete the forward or inverse transform of a full image with the direct method, simply use the `abel.Transform` class:

```
abel.Transform(myImage, method='direct', direction='forward').transform
abel.Transform(myImage, method='direct', direction='inverse').transform
```

If you would like to access the Direct algorithm directly (to transform a right-side half-image), you can use *abel.direct.direct_transform()*.

## 4.4 Hansen–Law

### 4.4.1 Introduction

The Hansen and Law transform [1, 2] is a fast (linear time) Abel transform.

In their words, Hansen and Law [1] present:

*"... new family of algorithms, principally for Abel inversion, that are recursive and hence computationally efficient. The methods are based on a linear, space-variant, state-variable model of the Abel transform. The model is the basis for deterministic algorithms."*

and [2]:

*"... Abel transform, which maps an axisymmetric two-dimensional function into a line integral projection."*

The algorithm is efficient, one of the few methods to provide both the **forward** Abel and **inverse** Abel transform.

### 4.4.2 How it works

For an axis-symmetric source image the projection of a source image, $g(R)$, is given by the forward Abel transform:

$$g(R) = 2\int_R^\infty \frac{f(r)r}{\sqrt{r^2 - R^2}}dr$$

The corresponding inverse Abel transform is:

$$f(r) = -\frac{1}{\pi}\int_r^\infty \frac{g'(R)}{\sqrt{R^2 - r^2}}dR$$

The Hansen and Law method makes a coordinate transformation to model the Abel transform as a set of linear differential equation, with the driving function either the source image $f(r)$, for the forward transform, or the projection image gradient $g'(R)$, for the inverse transform. More detail is given in *themath* below.

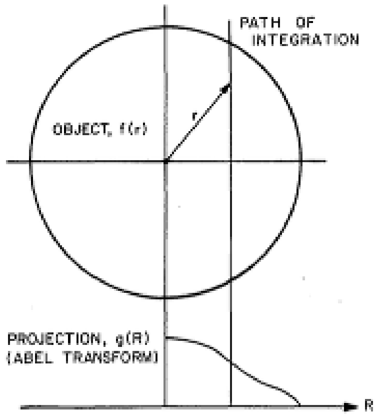


Fig. 1: Projection geometry (Fig. 1 [1])

Forward transform is:

$$\begin{aligned} x_{n-1} &= \Phi_n x_n + B_{0n} f_n + B_{1n} f_{n-1} \\ g_n &= \tilde{C} x_n, \end{aligned}$$

where $B_{1n} = 0$ for the zero-order hold approximation.

Inverse transform:

$$\begin{aligned} x_{n-1} &= \Phi_n x_n + B_{0n} g'_n + B_{1n} g'_{n-1} \\ f_n &= \tilde{C} x_n \end{aligned}$$

Note the only difference between the *forward* and *inverse* algorithms is the exchange of $f_n$ with $g'_n$ (or $g_n$).

Details on the evaluation of $\Phi$, $B_{0n}$, and $B_{1n}$ are given below, *themath*.

The algorithm iterates along each individual row of the image, starting at the out edge, ending at the center-line. Since all rows in an image can be processed simultaneously, the operation can be easily vectorized and is therefore numerically efficient.

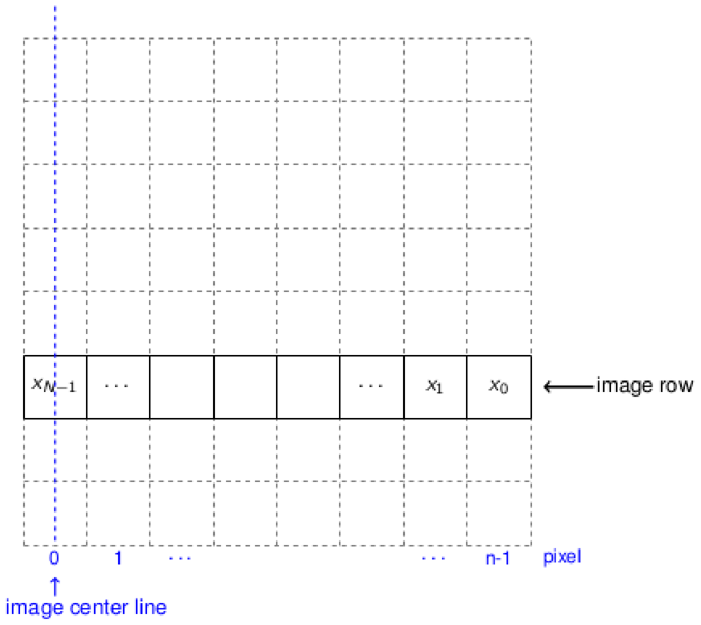


Fig. 2: Recursion: pixel value from adjacent outer-pixel

### 4.4.3 When to use it

The Hansen-Law algorithm offers one of the fastest, most robust methods for both the forward and inverse transforms. It requires reasonably fine sampling of the data to provide exact agreement with the analytical result, but otherwise this method is a hidden gem of the field.

### 4.4.4 How to use it

To complete the forward or inverse transform of
a full image with the `hansenlaw method`, simply use the `abel.Transform`: class

```
abel.Transform(myImage, method='hansenlaw', direction='forward').transform
abel.Transform(myImage, method='hansenlaw', direction='inverse').transform
```

If you would like to access the Hansen-Law algorithm directly (to transform a right-side half-image), you can use *`abel.hansenlaw.hansenlaw_transform()`*.

### 4.4.5 Tips

*hansenlaw* tends to perform better with images of large size $n1001$ pixel width. For smaller images the angular_integration (speed) profile may look better if sub-pixel sampling is used via:

```
angular_integration_options=dict(dr=0.5)
```

### 4.4.6 Example

*Source code*

### 4.4.7 Historical Note

The Hansen and Law algorithm was almost lost to the scientific community. It was rediscovered by Jason Gascooke (Flinders University, South Australia) for use in his velocity-map image analysis, and written up in his PhD thesis [3].

Eric Hansen provided guidence, algebra, and explanations, to aid the implementation of his first-order hold algorithm, described in Ref. [2] (April 2018).

### 4.4.8 The Math

The resulting state equations are, for the forward transform:

$$x'(r) = -\frac{1}{r}\tilde{A}x(r) + \frac{1}{\pi r}\tilde{B}f(R),$$

with inverse:

$$x'(R) = -\frac{1}{R}\tilde{A}x(R) - 2\tilde{B}f(R),$$

where $[\tilde{A}, \tilde{B}, \tilde{C}]$ realize the impulse response: $\tilde{h}(t) = \tilde{C}\exp(\tilde{A}t)\tilde{B} = \left[1 - e^{-2t}\right]^{-\frac{1}{2}}$, with:

$$\begin{aligned}\tilde{A} &= \mathrm{diag}[\lambda_1, \lambda_2, ..., \lambda_\mathrm{K}] \\ \tilde{B} &= [h_1, h_2, ..., h_K]^T \\ \tilde{C} &= [1, 1, ..., 1]\end{aligned}$$

The differential equations have the transform solutions, forward:

$$x(r) = \Phi(r, r_0)x(r_0) + 2\int_{r_0}^{r} \Phi(r, \epsilon)\tilde{B}f(\epsilon)d\epsilon.$$

and, inverse:

$$x(r) = \Phi(r, r_0)x(r_0) - \frac{1}{\pi}\int_{r_0}^{r} \frac{\Phi(r, \epsilon)}{r}\tilde{B}g'(\epsilon)d\epsilon,$$

with $\Phi(r, r_0) = \mathrm{diag}[(\frac{\mathrm{r_0}}{\mathrm{r}})^{\lambda_1}, ..., (\frac{\mathrm{r_0}}{\mathrm{r}})^{\lambda_\mathrm{K}}] \equiv \mathrm{diag}[(\frac{\mathrm{n}}{\mathrm{n-1}})^{\lambda_1}, ..., (\frac{\mathrm{n}}{\mathrm{n-1}})^{\lambda_\mathrm{K}}]$, where the integration limits $(r, r_0)$ extend across one grid interval or a pixel, so $r_0 = n\Delta$, $r = (n-1)\Delta$.

To evaluate the (superposition) integral, the driven part of the solution, the driving function $f(\epsilon)$ or $g'(\epsilon)$ is assumed to either be constant across each grid interval, the **zero-order hold** approximation, $f(\epsilon) \sim f(r_0)$, or linear, a **first-order hold** approximation, $f(\epsilon) \sim p + q\epsilon = (r_0 f(r) - r f(r_0))/\Delta + (f(r_0) - f(r))\epsilon/\Delta$. The integrand then separates into a sum over terms multiplied by $h_k$,

$$\sum_k h_k f(r_0)\int_{r_0}^{r} \Phi_k(r, \epsilon)d\epsilon$$

with each integral:

$$\int_{r_0}^{r} \left(\frac{\epsilon}{r}\right)_k^{\lambda} d\epsilon = \frac{r}{r_0}\left[1 - \left(\frac{r}{r_0}\right)^{\lambda_k+1}\right] = \frac{(n-1)^a}{\lambda_k + a}\left[1 - \left(\frac{n}{n-1}\right)^{\lambda_k+a}\right],$$

where, the right-most-side of the equation has an additional parameter, $a$ to generalize the power of $\lambda_k$. For the inverse transform, there is an additional factor $\frac{1}{\pi r}$ in the state equation, and hence the integrand has $\lambda_k$ power, reduced by -1. While, for the first-order hold approximation, the linear $\epsilon$ term increases $\lambda_k$ by +1.

### 4.4.9 Citation

[1] E. W. Hansen and P.-L. Law, "Recursive methods for computing the Abel transform and its inverse", J. Opt. Soc. A2, 510-520 (1985).

[2] E. W. Hansen, "Fast Hankel Transform", IEEE Trans. Acoust. Speech Signal Proc. 33, 666 (1985).

[3] J. R. Gascooke, PhD Thesis: "Energy Transfer in Polyatomic-Rare Gas Collisions and Van Der Waals Molecule Dissociation", Flinders University (2000).

## 4.5 Lin-Basex

### 4.5.1 Introduction

Inversion procedure based on 1-dimensional projections of VM-images as described in Gerber et al. [1].

[ *from the abstract* ]

*VM-images are composed of projected Newton spheres with a common centre. The 2D images are usually evaluated by a decomposition into base vectors each representing the 2D projection of a set of particles starting from a centre with a specific velocity distribution. We propose to evaluate 1D projections of VM-images in terms of 1D projections of spherical functions, instead. The proposed evaluation algorithm shows that all distribution information can be retrieved from an adequately chosen set of 1D projections, alleviating the numerical effort for the interpretation of VM-images considerably. The obtained results produce directly the coefficients of the involved spherical functions, making the reconstruction of sliced Newton spheres obsolete.*

### 4.5.2 How it works

A projection of 3D Newton spheres along the $x$ axis yields a compact 1D function:

$$L(z,u) = \sum_k \sum_\ell P_\ell(u) P_\ell\left(\frac{z}{r_k}\right) \frac{\prod_{r_k}(z)}{2r_k} p_{\ell k}$$

with $u = \cos(\theta)$. This function constitutes a system of equations expressing $L(z,u)$ as a linear combination of $P_\ell(z/r_k)$. There exists for a given base a unique set of coefficients $p_{\ell k}$ producing a least-squares fit to the function $L(z,u)$.

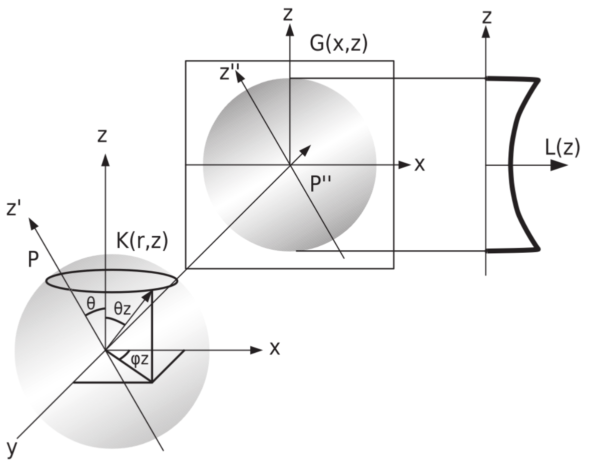


Fig. 3: projections (Fig. 2 of [1])

[ *extract of a comment made by Thomas Gerber (method author)* ]

*Imaging an PES experiment which produces electrons that are distributed on the surface of a sphere. This sphere can be described by spherical functions. If all electrons have the same energy we expect them on a (Newton) sphere with radius $i$. This radius is projected to the CCD. The distribution on the CCD has (if optics are approriate) the same radius $i$. Now let us assume that the distribution on the Newton sphere has some anisotropy. We can describe the distribution on this sphere by spherical functions $Y_{nm}$. Let's say $xY_{00} + yY_{20}$. The 1D projection of those spheres produces just $xP_{i0}(k) + yP_{i2}(k)$ where $P_i$ denotes Legendre Polynomials scaled to the interval $i$ and $k$ is the argument (pixel).*

*For one projection Lin-Basex now solves for the parameters $x$ and $y$. If we look at another projection turned by an angle, the Basis $P_{i0}$ and $P_{i2}$ has to be modified because the projection of e.g., $Y_{20}$ turned by an angle yields another function. It was shown that this function for e.g., $P_2$ is just $P_2(a)P_{i2}(k)$ where $a$ is the turning angle. Solving the equations for the 1D projection at angle $(a)$ with this modified basis yields the same $x$* and *$y$ parameters as before.*

*Lin-Basex aims at the determination of contributions in terms of spherical functions calculating the weight of each $Y_{l0}$. If we reconstruct the 3D object by adding all the $Y_{l0}$ contributions we get the inverse Laplace transform of the image on the CCD from which we can derive "Slices".*

### 4.5.3 When to use it

[ *another extract from comments by the method author Thomas Gerber* ]

*The advantage of* `linbasex` *is, that not so many projections are needed (typically* `len(an) ~ len(pol)()`*). So,* `linbasex` *evaluation using a mathematically appropriate and correct basis set should eventually be much faster than* `basex`*.*

*If our 3D object is "sparse" (i.e., contains a sparse set of Newton spheres) a sparse basis may be used. In this case one must have primary information about what "sparsity" is appropriate.*

*That means that an Abel transform may be simplified if primary information about the object is available. That is not the case with the other methods.*

*Absolute noise increases in each sphere with sqrt(counts) but relative noise decreases with* $1/\sqrt{\text{counts}}$.

### 4.5.4 How to use it

To complete the inverse Abel transform of a full image with the `linbasex method`, simply use the `abel.Transform`: class

```
abel.Transform(myImage, method='linbasex').transform
```

Note, the parameter `transform_options=dict(return_Beta=True)`, provides additional attributes, direct from the transform procedure:

- `.Beta[0]` - the speed distribution
- `.Beta[1]` - the anisotropy parameter vs radius
- `.radial` - the radial array
- `.projection` - the radial projections at angles *an*.

A more complete global call, that centers the image, ensures that the size is odd, and returns the attributes above, would be e.g.

```
abel.Transform(myImage, method='linbasex', center='convolution',
               transform_options=dict(return_Beta=True))
```

Alternatively, the linbasex algorithm *`abel.linbasex.linbasex_transform_full()`* directly transforms the full image, with the attributes returned as a tuple in this case.

### 4.5.5 Tips

Including more projection angles may improve the transform:

```
an = [0, 45, 90, 135]
```

or

```
an = arange(0, 180, 10)
```

### 4.5.6 Example

*Source code*

### 4.5.7 Historical

PyAbel python code was extracted from this jupyter notebook supplied by Thomas Gerber.

### 4.5.8 Citation

[1] Gerber, Thomas, Yuzhu Liu, Gregor Knopp, Patrick Hemberger, Andras Bodi, Peter Radi, and Yaroslav Sych, "Charged Particle Velocity Map Image Reconstruction with One-Dimensional Projections of Spherical Functions." Rev. Sci. Instrum. 84, no. 3, 033101 (2013)

## 4.6 Two Point (Dasch)

### 4.6.1 Introduction

The "Dasch two-point" deconvolution algorithm is one of several described in the Dasch [1] paper. See also the `three_point` and `onion_peeling` descriptions.

### 4.6.2 How it works

The Abel integral is broken into intervals between the $r_j$ points, and $P'(r)$ is assumed constant between $r_j$ and $r_{j+1}$.

### 4.6.3 When to use it

This method is simple and computationally very efficient. The method incorporates no smoothing.

### 4.6.4 How to use it

To complete the inverse transform of a full image with the `two_point method`, simply use the `abel.Transform` class:

```
abel.Transform(myImage, method='two_point').transform
```

If you would like to access the `two_point` algorithm directly (to transform a right-side half-image), you can use *`abel.dasch.two_point_transform()`*.

### 4.6.5 Example

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

```

(continues on next page)

(continued from previous page)

```
"""example_dasch_methods.py.
"""

import numpy as np
import abel
import matplotlib.pyplot as plt

# Dribinski sample image size 501x501
n = 501
IM = abel.tools.analytical.SampleImage(n).image

# split into quadrants
origQ = abel.tools.symmetry.get_image_quadrants(IM)

# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration(origQ[0], origin=(0,0))
scale_factor = orig_speed[1].max()

plt.plot(orig_speed[0], orig_speed[1]/scale_factor, linestyle='dashed',
         label="Dribinski sample")


# forward Abel projection
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

# split projected image into quadrants
Q = abel.tools.symmetry.get_image_quadrants(fIM)

dasch_transform = {\
"two_point": abel.dasch.two_point_transform,
"three_point": abel.dasch.three_point_transform,
"onion_peeling": abel.dasch.onion_peeling_transform}

for method in dasch_transform.keys():
    Q0 = Q[0].copy()
# method inverse Abel transform
    AQ0 = dasch_transform[method](Q0, basis_dir='bases')
# speed distribution
    speed = abel.tools.vmi.angular_integration(AQ0, origin=(0,0))

    plt.plot(speed[0], speed[1]*orig_speed[1][14]/speed[1][14]/scale_factor,
             label=method)

plt.title("Dasch methods for Dribinski sample image $n={:d}$".format(n))
plt.axis(xmax=250, ymin=-0.1)
plt.legend(loc=0, frameon=False, labelspacing=0.1, fontsize='small')
plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```

or more information on the PyAbel implementation of the `two_point` algorithm, please see Pull Request #155.

### 4.6.6 Citation

[1] Dasch, Applied Optics, Vol 31, No 8, March 1992, Pg 1146-1152.

## 4.7 Three Point

### 4.7.1 Introduction

The "Three Point" Abel transform method exploits the observation that the value of the Abel inverted data at any radial position `r` is primarily determined from changes in the projection data in the neighborhood of `r`. This technique was developed by Dasch [1].

### 4.7.2 How it works

The projection data (raw data $\mathbf{P}$) is expanded as a quadratic function of $r - r_{j*}$ in the neighborhood of each data point in $\mathbf{P}$. In other words, $\mathbf{P}'(r) = dP/dr$ is estimated using a 3-point approximation (to the derivative), similar to central differencing. Doing so enables an analytical integration of the inverse Abel integral around each point $r_j$. The result of this integration is expressed as a linear operator $\mathbf{D}$, operating on the projection data $\mathbf{P}$ to give the underlying radial distribution $\mathbf{F}$.

### 4.7.3 When to use it

Dasch recommends this method based on its speed of implementation, robustness in the presence of sharp edges, and low noise. He also notes that this technique works best for cases where the real difference between adjacent projections is much greater than the noise in the projections. This is important, because if the projections are oversampled (raw data $\mathbf{P}$ taken with data points very close to each other), the spacing between adjacent projections is decreased, and the real difference between them becomes comparable with the noise in the projections. In such situations, the deconvolution is highly inaccurate, and the projection data $\mathbf{P}$ must be smoothed before this technique is used. (Consider smoothing with scipy.ndimage.filters.gaussian_filter.)

### 4.7.4 How to use it

To complete the inverse transform of a full image with the `three_point method`, simply use the `abel.Transform` class:

```
abel.Transform(myImage, method='three_point', direction='inverse').transform
```

Note that the forward Three point transform is not yet implemented in PyAbel.

If you would like to access the Three Point algorithm directly (to transform a right-side half-image), you can use *`abel.dasch.three_point_transform()`*.

### 4.7.5 Example

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

"""example_dasch_methods.py.
"""

import numpy as np
import abel
```

(continues on next page)

(continued from previous page)

```python
import matplotlib.pyplot as plt

# Dribinski sample image size 501x501
n = 501
IM = abel.tools.analytical.SampleImage(n).image

# split into quadrants
origQ = abel.tools.symmetry.get_image_quadrants(IM)

# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration(origQ[0], origin=(0,0))
scale_factor = orig_speed[1].max()

plt.plot(orig_speed[0], orig_speed[1]/scale_factor, linestyle='dashed',
         label="Dribinski sample")


# forward Abel projection
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

# split projected image into quadrants
Q = abel.tools.symmetry.get_image_quadrants(fIM)

dasch_transform = {\
"two_point": abel.dasch.two_point_transform,
"three_point": abel.dasch.three_point_transform,
"onion_peeling": abel.dasch.onion_peeling_transform}

for method in dasch_transform.keys():
    Q0 = Q[0].copy()
# method inverse Abel transform
    AQ0 = dasch_transform[method](Q0, basis_dir='bases')
# speed distribution
    speed = abel.tools.vmi.angular_integration(AQ0, origin=(0,0))

    plt.plot(speed[0], speed[1]*orig_speed[1][14]/speed[1][14]/scale_factor,
             label=method)

plt.title("Dasch methods for Dribinski sample image $n={:d}$".format(n))
plt.axis(xmax=250, ymin=-0.1)
plt.legend(loc=0, frameon=False, labelspacing=0.1, fontsize='small')
plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```

### 4.7.6 Notes

The algorithm contained two typos in Eq (7) in the original citation [1]. A corrected form of these equations is presented in Karl Martin's 2002 PhD thesis [2]. PyAbel uses the corrected version of the algorithm.

For more information on the PyAbel implementation of the three-point algorithm, please see Issue #61 and Pull Request #64.

### 4.7.7 Citation

[1] Dasch, Applied Optics, Vol 31, No 8, March 1992, Pg 1146-1152.

[2] Martin, Karl. PhD Thesis, University of Texas at Austin. Acoustic Modification of Sooting Combustion. 2002: https://www.lib.utexas.edu/etd/d/2002/martinkm07836/martinkm07836.pdf

## 4.8 Onion Peeling (Dasch)

### 4.8.1 Introduction

The "Dasch onion peeling" deconvolution algorithm is one of several described in the Dasch [1] paper. See also the `two_point` and `three_point` descriptions.

### 4.8.2 How it works

In the onion-peeling method the projection is approximated by rings of constant property between $r_j - \Delta r/2$ and $r_j + \Delta r/2$ for each data point $r_j$.

The projection data is given by $P(r_i) = \Delta r \sum_{j=i}^{\infty} W_{ij} F(r_j)$

where

$$
\begin{aligned}
W_{ij} &= 0 \ (j < i) \\
&\sqrt{(2j+1)^2 - 4i^2} \ (j = i) \\
&\sqrt{(2j+1)^2 - 4i^2} - \sqrt{(2j-1)^2 - 4i^2} \ (j > i)
\end{aligned}
$$

The onion-peeling deconvolution function is: $D_{ij} = (W^{-1})_{ij}$.

### 4.8.3 When to use it

This method is simple and computationally very efficient. The article states that it has less smoothing that other methods (discussed in Dasch).

### 4.8.4 How to use it

To complete the inverse transform of a full image with the `onion_dasch` method, simply use the `abel.Transform` class:

```
abel.Transform(myImage, method='onion_peeling').transform
```

If you would like to access the `onion_peeling` algorithm directly (to transform a right-side half-image), you can use *`abel.dasch.onion_peeling_transform()`*.

### 4.8.5 Example

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

"""example_dasch_methods.py.
"""
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import abel
import matplotlib.pyplot as plt

# Dribinski sample image size 501x501
n = 501
IM = abel.tools.analytical.SampleImage(n).image

# split into quadrants
origQ = abel.tools.symmetry.get_image_quadrants(IM)

# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration(origQ[0], origin=(0,0))
scale_factor = orig_speed[1].max()

plt.plot(orig_speed[0], orig_speed[1]/scale_factor, linestyle='dashed',
         label="Dribinski sample")


# forward Abel projection
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

# split projected image into quadrants
Q = abel.tools.symmetry.get_image_quadrants(fIM)

dasch_transform = {\
"two_point": abel.dasch.two_point_transform,
"three_point": abel.dasch.three_point_transform,
"onion_peeling": abel.dasch.onion_peeling_transform}

for method in dasch_transform.keys():
    Q0 = Q[0].copy()
# method inverse Abel transform
    AQ0 = dasch_transform[method](Q0, basis_dir='bases')
# speed distribution
    speed = abel.tools.vmi.angular_integration(AQ0, origin=(0,0))

    plt.plot(speed[0], speed[1]*orig_speed[1][14]/speed[1][14]/scale_factor,
             label=method)

plt.title("Dasch methods for Dribinski sample image $n={:d}$".format(n))
plt.axis(xmax=250, ymin=-0.1)
plt.legend(loc=0, frameon=False, labelspacing=0.1, fontsize='small')
plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```

or more information on the PyAbel implementation of the `onion_peeling` algorithm, please see Pull Request #155.

### 4.8.6 Citation

[1] Dasch, Applied Optics, Vol 31, No 8, March 1992, Pg 1146-1152.

## 4.9 Onion Peeling (Bordas)

### 4.9.1 Introduction

The onion peeling method, also known as "back projection" has been ported to Python from the original Matlab implementation, created by Chris Rallis and Eric Wells of Augustana University, and described in this paper [1]. The algorithm actually originates from this 1996 RSI paper by Bordas ~et al.[2]

See the discussion here: https://github.com/PyAbel/PyAbel/issues/56

### 4.9.2 How it works

This algorithm calculates the contributions of particles, at a given kinetic energy, to the signal in a given pixel (in a row). This signal is then subtracted from the projected (experimental) pixel and also added to the back-projected image pixel. The procedure is repeated until the center of the image is reached. The whole procedure is done for each pixel row of the image.

### 4.9.3 When to use it

This is a historical implementation of the onion-peeling method.

### 4.9.4 How to use it

To complete the inverse transform of a full image with the `onion_bordas` method, simply use the `abel.Transform`: class

```
abel.Transform(myImage, method='onion_bordas').transform
```

If you would like to access the onion-peeling algorithm directly (to transform a right-side half-image), you can use *`abel.onion_bordas.onion_bordas_transform()`*.

### 4.9.5 Example

*Source code*

### 4.9.6 Citation

[1] http://scitation.aip.org/content/aip/journal/rsi/85/11/10.1063/1.4899267

[2] http://scitation.aip.org/content/aip/journal/rsi/67/6/10.1063/1.1147044

## 4.10 Polar Onion Peeling (not implemented)

### 4.10.1 Introduction

The polar onion peeling (POP) method is still under development.

See the discussion here: https://github.com/PyAbel/PyAbel/issues/30

### 4.10.2 How it works

It doesn't exists in PyAbel!

### 4.10.3 When to use it

When you implement it! :)

### 4.10.4 How to use it

Code it!

### 4.10.5 Example

Put it here!

### 4.10.6 Citation

[1] http://dx.doi.org/10.1063/1.3126527

[2] http://www.mathworks.com/matlabcentral/fileexchange/41064-polar-onion-peeling

## 4.11 Fourier–Hankel

### 4.11.1 Introduction

The Fourier–Hankel method breaks the Abel transform in to a Fourier transform and a Hankel transform. It takes advantage of the fact that there are fast numerical implementations of the Fourier and Hankel transforms to provide a quick alorithm. It is known to produce artifacts in the transform [Dribinski2002]

This method is not yet implemented in PyAbel. See the discussion in Issue #26 for more information.

### 4.11.2 How it works

It doesn't work in PyAbel yet.

### 4.11.3 When to use it

To compare with other methods? Very large images?

### 4.11.4 How to use it

Implement it!

### 4.11.5 Example

### 4.11.6 Notes

### 4.11.7 Citation

# CHAPTER 5

# Anisotropy Parameter

For linearly polarized light the angular distribution of photodetached electrons from negative ions is given by

$$I(\epsilon, \theta) = \frac{\sigma_{\text{total}}(\epsilon)}{4\pi}[1 + \beta(\epsilon)P_2(\cos\theta)],$$

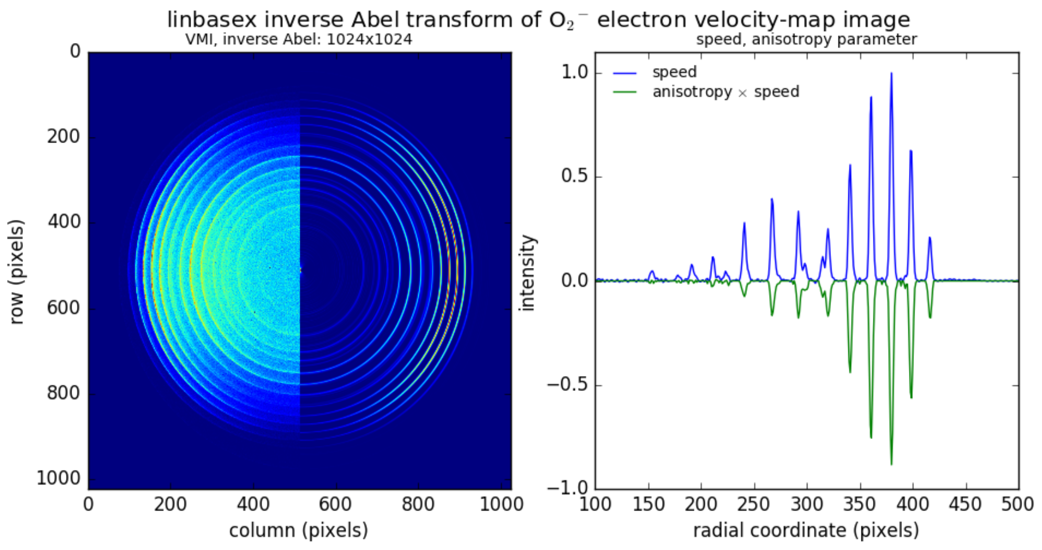
where $\beta(\epsilon)$ is the electron kinetic energy ($\epsilon$) dependent anisotropy parameter, which varies between 1 and +2, and $P_2(\cos\theta)$ is the 2nd-order Legendre polynomial in $\cos\theta$. $\sigma_{\text{total}}$ is the total photodetachment cross section. The anisotropy parameter provides phase information about the dynamics of the photon process [1].

## 5.1 Methods

`PyAbel` provides several methods to determine the anisotropy parameter $\beta$:

Method 1: *linbasex* evaluates $\beta$ directly, available as the class attribute *Beta[1]*.

> This method fits spherical harmonic functions to the velocity-map image to directly determine the anisotropy parameter as a function of the radial coordinate. This parameter has greater uncertainty in radial regions of low intensity, and so it is commonly plotted as the product $I \times \beta$. See `examples/example_linbasex.py`.



Method 2: using *abel.tools.vmi.radial_integration()*.

This method determines the anisotropy parameter from the inverse Abel-transformed image, by extracting intensity vs angle for each specified radial range and then fitting the intensity formula given above. This method is best applied to the radial ranges corresponding to strong spectral intensity in the image. It has the advantage of providing the least-squares fit error estimate for the parameter(s).

Method 3: using *abel.tools.vmi.Distributions*.

This method, like the previous one, works on the inverse Abel-transformed image, but fits the angular intensity dependence at each radius, providing radially dependent anisotropy parameters, like in the first method. If the anisotropy parameters are known to be smooth radial functions, a moving-window averaging can be employed for noise reduction.

## 5.2 Example

See *Example: Anisotropy parameter*. In this case the anisotropy parameter is determined using each method. Note:

- In method 1, the filter parameter `threshold=0.2` is set to a larger value so as to exclude evaluation in regions of weak intensity.
- Method 2 evaluates the anisotropy parameter for particular radial regions of strong intensity.
- In method 3, the anisotropy parameter is calculated with 9-pixel radial averaging and plotted only in the regions with > 1% of the maximal intensity.

## 5.3 Reference

[1] J. Cooper and R. N. Zare, "Angular distribution of photoelectrons", J. Chem. Phys. 48, 942 (1968)

CHAPTER 6

# Circularization of Images

## 6.1 Background

While the Abel transform only assumes cylindrical symmetry, often the objects to be transformed also have some degree of spherical symmetry, (i.e., features that appear at a constant radius for all angles) and thus the 2D projection should be perfectly circular. Experimental images may have distortions in the circular charged particle energy structure, due to, for example, stray magnetic fields, or optical distortion of the camera lens that images the particle detector. The effect of distortion is to degrade the radial (or velocity or kinetic energy) resolution, since a particular energy peak will "walk" in radial position, depending on the particular angular position on the detector. Imposing a physical circular distribution of particles, may substantially improve the kinetic energy resolution, at the expense of uncertainly in the absolution kinetic-energy position of the transition.

## 6.2 Approach

The algorithm is implemented in `abel.tools.circularize.circularize_image()` compares the radial positions of strong features in angular slice intensity profiles. i.e. follow the radial position of a peak as a function of angle. A linear correction is applied to the radial grid to align the peak at each angle.

```
 before     after
   ^          ^    slice0
     ^        ^    slice1
   ^          ^    slice2
  ^           ^    slice3
   :          :
    ^         ^    slice#
radial peak position
```

Peak alignment is achieved through a radial scaling factor $R_i(actual) = R_i \times scalefactor_i$. The scalefactor is determined by a choice of methods, `argmax`, where $scalefactor_i = R_0/R_i$, with $R_0$ a reference peak. Or `lsq`, which directly determines the radial scaling factor that best aligns adjacent slice intensity profiles.

This is a simplified radial scaling version of the algorithm described in J. R. Gascooke and S. T. Gibson and W. D. Lawrance: 'A "circularisation" method to repair deformations and determine the centre of velocity map images' J. Chem. Phys. 147, 013924 (2017).

## 6.3 Implementation

Cartesian $(y, x)$ image is converted to a polar coordinate image $(r, \theta)$ for easy slicing into angular blocks. Each radial intensity profile is compared with its adjacent slice, providing a radial scaling factor that best aligns the two intensity profiles.

The set of radial scaling factors, for each angular slice, is then spline interpolated to correct the $(y, x)$ grid, and the image remapped to an unperturbed grid.

## 6.4 How to use it

The `circularize_image()` function is called directly

```
IMcirc, angle, radial_correction, radial_correction_function =\
    abel.tools.circularize.circularize_image(IM, method='lsq',\
    center='slice', dr=0.5, dt=0.1, return_correction=True)
```

The main input parameters are the image *IM*, and the number of angular slices, to use, which is set by $2\pi/dt$. The default *dt* = *0.1* uses ~63 slices. This parameter determines the angular resolution of the distortion correction function, but is limited by the signal to noise loss with smaller *dt*. Other parameters may help better define the radial correction function.

## 6.5 Warning

Ensure the returned radial_correction vs angle data is a well behaved function. See the example, below, bottom left figure. If necessary limit the `radial_range=(Rmin, Rmax)`, or change the value of the spline smoothing parameter.

## 6.6 Example

```
import numpy as np
import matplotlib.pyplot as plt
import abel
import scipy.interpolate

#######################################################################
#
# example_circularize_image.py
#
# O- sample image -> forward Abel + distortion = measured VMI
#  measured VMI   -> inverse Abel transform -> speed distribution
# Compare disorted and circularized speed profiles
#
#######################################################################
```

(continues on next page)

(continued from previous page)

```
# sample image -----------
IM = abel.tools.analytical.SampleImage(n=511, name='Ominus', sigma=2).image

# forward transform == what is measured
IMf = abel.Transform(IM, method='hansenlaw', direction="forward").transform

# flower image distortion
def flower_scaling(theta, freq=2, amp=0.1):
    return 1 + amp*np.sin(freq*theta)**4

# distort the image
IMdist = abel.tools.circularize.circularize(IMf,
                                radial_correction_function=flower_scaling)

# circularize ------------
IMcirc, sla, sc, scspl = abel.tools.circularize.circularize_image(IMdist,
               method='lsq', dr=0.5, dt=0.1, smooth=0, return_correction=True)

# inverse Abel transform for distored and circularized images ---------
AIMdist = abel.Transform(IMdist, method="three_point",
                         transform_options=dict(basis_dir='bases')).transform
AIMcirc = abel.Transform(IMcirc, method="three_point",
                         transform_options=dict(basis_dir='bases')).transform

# respective speed distributions
rdist, speeddist = abel.tools.vmi.angular_integration(AIMdist, dr=0.5)
rcirc, speedcirc = abel.tools.vmi.angular_integration(AIMcirc, dr=0.5)

# note the small image size is responsible for the slight over correction
# of the background near peaks

row, col = IMcirc.shape

# plot ---------------------

fig, axs = plt.subplots(2, 2, figsize=(8, 8))
fig.subplots_adjust(wspace=0.5, hspace=0.5)

extent = (np.min(-col//2), np.max(col//2), np.min(-row//2), np.max(row//2))
axs[0, 0].imshow(IMdist, aspect='auto', origin='lower', extent=extent)
axs[0, 0].set_title("Ominus distorted sample image")

axs[0, 1].imshow(AIMcirc, vmin=0, aspect='auto', origin='lower',
                 extent=extent)
axs[0, 1].set_title("circ. + inv. Abel")

axs[1, 0].plot(sla, sc, 'o')
ang = np.arange(-np.pi, np.pi, 0.1)
axs[1, 0].plot(ang, scspl(ang))
axs[1, 0].set_xticks([-np.pi, 0, np.pi])
axs[1, 0].set_xticklabels([r"$-\pi$", "0", r"$\pi$"])
axs[1, 0].set_xlabel("angle (radians)")
axs[1, 0].set_ylabel("radial correction factor")
axs[1, 0].set_title("radial correction")
```

(continues on next page)

(continued from previous page)

```
axs[1, 1].plot(rdist, speeddist, label='dist.')
axs[1, 1].plot(rcirc, speedcirc, label='circ.')
axs[1, 1].axis(xmin=100, xmax=240)
axs[1, 1].set_title("speed distribution")
axs[1, 1].legend(frameon=False)
axs[1, 1].set_xlabel('radius (pixels)')
axs[1, 1].set_ylabel('intensity')

plt.savefig("plot_example_circularize_image.png", dpi=75)
plt.show()
```

# CHAPTER 7

# Examples

Contents:

## 7.1 Example: Direct Gaussian

```
# -*- coding: utf-8 -*-

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import matplotlib.pyplot as plt
from time import time
import sys

from abel.direct import direct_transform
from abel.tools.analytical import GaussianAnalytical


n = 101
r_max = 30
sigma = 10

ref = GaussianAnalytical(n, r_max, sigma, symmetric=False)

fig, ax = plt.subplots(1,2)

# forward Abel transform
reconC = direct_transform(ref.func, dr=ref.dr, direction="forward",
                          correction=True)
reconP = direct_transform(ref.func, dr=ref.dr, direction="forward",
                          correction=False)
```

(continues on next page)

(continued from previous page)

```
ax[0].set_title('Forward transform of a Gaussian', fontsize='smaller')
ax[0].plot(ref.r, ref.abel, label='Analytical transform')
ax[0].plot(ref.r, reconC , '--', label='correction=True')
ax[0].plot(ref.r, reconP , ':', label='correction=False')
ax[0].set_ylabel('intensity (arb. units)')
ax[0].set_xlabel('radius')


# inverse Abel transform
reconc = direct_transform(ref.abel, dr=ref.dr, direction="inverse",
                          correction=True)

reconnoc = direct_transform(ref.abel, dr=ref.dr, direction="inverse",
                         correction=False)

ax[1].set_title('Inverse transform of a Gaussian', fontsize='smaller')
ax[1].plot(ref.r, ref.func, 'C0', label='Original function')
ax[1].plot(ref.r, reconc , 'C1--', label='correction=True')
ax[1].plot(ref.r, reconnoc , 'C2:', label='correction=False')
ax[1].set_xlabel('radius')

for axi in ax:
    axi.set_xlim(0, 20)
    axi.legend(labelspacing=0.1, fontsize='smaller')

plt.savefig("plot_example_direct_gaussian.png", dpi=100)
plt.show()
```

## 7.2 Example: Hansen–Law

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel
import matplotlib.pylab as plt
import bz2

# Hansen and Law inverse Abel transform of velocity-map imaged electrons
# from O2- photodetachement at 454 nm. The spectrum was recorded in 2010
# at the Australian National University (ANU)
# J. Chem. Phys. 133, 174311 (2010) DOI: 10.1063/1.3493349

# load image as a numpy array
# use scipy.misc.imread(filename) to load image formats (.png, .jpg, etc)
print('HL: loading "data/O2-ANU1024.txt.bz2"')
imagefile = bz2.BZ2File('data/O2-ANU1024.txt.bz2')
IM = np.loadtxt(imagefile)

rows, cols = IM.shape    # image size
```

(continues on next page)

(continued from previous page)

```
# center image returning odd size
IMc = abel.tools.center.center_image(IM, center='com')

# dr=0.5 may help reduce pixel grid coarseness
# NB remember to also pass as an option to angular_integration
AIM = abel.Transform(IMc, method='hansenlaw',
                     use_quadrants=(True, True, True, True),
                     symmetry_axis=None,
                     transform_options=dict(dr=0.5, align_grid=False),
                     angular_integration=True,
                     angular_integration_options=dict(dr=0.5),
                     verbose=True)

# convert to photoelectron spectrum vs binding energy
# conversion factors depend on measurement parameters
eBE, PES = abel.tools.vmi.toPES(*AIM.angular_integration,
                                energy_cal_factor=1.204e-5,
                                photon_energy=1.0e7/454.5, Vrep=-2200,
                                zoom=IM.shape[-1]/2048)

# Set up some axes
fig = plt.figure(figsize=(15, 4))
ax1 = plt.subplot2grid((1, 3), (0, 0))
ax2 = plt.subplot2grid((1, 3), (0, 1))
ax3 = plt.subplot2grid((1, 3), (0, 2))

# raw image
im1 = ax1.imshow(IM, aspect='auto', extent=[-512, 512, -512, 512])
fig.colorbar(im1, ax=ax1, fraction=.1, shrink=0.9, pad=0.03)
ax1.set_xlabel('x (pixels)')
ax1.set_ylabel('y (pixels)')
ax1.set_title('velocity map image: size {:d}x{:d}'.format(rows, cols))

# 2D transform
c2 = cols//2   # half-image width
im2 = ax2.imshow(AIM.transform, aspect='auto', vmin=0,
                 vmax=AIM.transform[:c2-50, :c2-50].max(),
                 extent=[-512, 512, -512, 512])
fig.colorbar(im2, ax=ax2, fraction=.1, shrink=0.9, pad=0.03)
ax2.set_xlabel('x (pixels)')
ax2.set_ylabel('y (pixels)')
ax2.set_title('Hansen Law inverse Abel')

# 1D speed distribution
#ax3.plot(radial, speeds/speeds[200:].max())
#ax3.axis(xmax=500, ymin=-0.05, ymax=1.1)
#ax3.set_xlabel('speed (pixel)')
#ax3.set_ylabel('intensity')
#ax3.set_title('speed distribution')

# PES
ax3.plot(eBE, PES/PES[eBE < 5000].max())
ax3.axis(xmin=0)
ax3.set_xlabel(r'elecron binding energy (cm$^{-1}$)')
ax3.set_ylabel('intensity')
ax3.set_title(r'O${_2}{^-}$ 454 nm photoelectron spectrum')
```

(continues on next page)

(continued from previous page)

```
# Prettify the plot a little bit:
plt.subplots_adjust(left=0.06, bottom=0.17, right=0.95, top=0.89, wspace=0.35,
                    hspace=0.37)

# save copy of the plot
plt.savefig('plot_example_hansenlaw.png', dpi=100)

plt.show()
```

## 7.3 Example: $O_2$ PES PAD

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel
import bz2

import matplotlib.pylab as plt

# This example demonstrates Hansen and Law inverse Abel transform
# of an image obtained using a velocity map imaging (VMI) photoelecton
# spectrometer to record the photoelectron angular distribution resulting
# from photodetachement of O2- at 454 nm.
# Measured at  The Australian National University
# J. Chem. Phys. 133, 174311 (2010) DOI: 10.1063/1.3493349

# Load image as a numpy array - numpy handles .gz, .bz2
imagefile = bz2.BZ2File('data/O2-ANU1024.txt.bz2')
IM = np.loadtxt(imagefile)
# use scipy.misc.imread(filename) to load image formats (.png, .jpg, etc)

rows, cols = IM.shape    # image size

# Image center should be mid-pixel, i.e. odd number of colums
if cols % 2 != 1:
    print ("even pixel width image, make it odd and re-adjust image center")
    IM = abel.tools.center.center_image(IM, center="slice")
    rows, cols = IM.shape   # new image size

r2 = rows//2   # half-height image size
c2 = cols//2   # half-width image size

# Hansen & Law inverse Abel transform
AIM = abel.Transform(IM, method="hansenlaw", direction="inverse",
                     symmetry_axis=None).transform

# PES - photoelectron speed distribution  -------------
print('Calculating speed distribution:')

r, speed  = abel.tools.vmi.angular_integration(AIM)
```

(continues on next page)

(continued from previous page)

```
# normalize to max intensity peak
speed /= speed[200:].max()  # exclude transform noise near centerline of image

# PAD - photoelectron angular distribution  ------------
print('Calculating angular distribution:')
# radial ranges (of spectral features) to follow intensity vs angle
# view the speed distribution to determine radial ranges
r_range = [(93, 111), (145, 162), (255, 280), (330, 350), (350, 370),
           (370, 390), (390, 410), (410, 430)]

# map to intensity vs theta for each radial range
Beta, Amp, rad,intensities, theta = abel.tools.vmi.radial_integration(AIM, radial_
↪ranges=r_range)

print("radial-range      anisotropy parameter (beta)")
for beta, rr in zip(Beta, r_range):
    result = "    {:3d}-{:3d}        {:+.2f}+-{:.2f}"\
             .format(rr[0], rr[1], beta[0], beta[1])
    print(result)

# plots of the analysis
fig = plt.figure(figsize=(15, 4))
ax1 = plt.subplot(131)
ax2 = plt.subplot(132)
ax3 = plt.subplot(133)

# join 1/2 raw data : 1/2 inversion image
vmax = IM[:, :c2-100].max()
AIM *= vmax/AIM[:, c2+100:].max()
JIM = np.concatenate((IM[:, :c2], AIM[:, c2:]), axis=1)
rr = r_range[-3]
intensity = intensities[-3]
beta, amp = Beta[-3], Amp[-3]

# Prettify the plot a little bit:
# Plot the raw data
im1 = ax1.imshow(JIM, origin='lower', aspect='auto', vmin=0, vmax=vmax)
fig.colorbar(im1, ax=ax1, fraction=.1, shrink=0.9, pad=0.03)
ax1.set_xlabel('x (pixels)')
ax1.set_ylabel('y (pixels)')
ax1.set_title('VMI, inverse Abel: {:d}x{:d}'\
              .format(rows, cols))

# Plot the 1D speed distribution
ax2.plot(speed)
ax2.plot((rr[0], rr[0], rr[1], rr[1]), (1, 1.1, 1.1, 1), 'r-')  # red highlight
ax2.axis(xmax=450, ymin=-0.05, ymax=1.2)
ax2.set_xlabel('radial pixel')
ax2.set_ylabel('intensity')
ax2.set_title('speed distribution')

# Plot anisotropy variation
ax3.plot(theta, intensity, 'r',
         label="expt. data r=[{:d}:{:d}]".format(*rr))


def P2(x):   # 2nd order Legendre polynomial
```

(continues on next page)

(continued from previous page)

```
    return (3*x*x-1)/2


def PAD(theta, beta, amp):
    return amp*(1 + beta*P2(np.cos(theta)))


ax3.plot(theta, PAD(theta, beta[0], amp[0]), 'b', lw=2, label="fit")
ax3.annotate("$\\beta = ${:+.2f}+-{:.2f}".format(*beta), (-2, -1.1))
ax3.legend(loc=1, labelspacing=0.1, fontsize='small')

ax3.axis(ymin=-2, ymax=12)
ax3.set_xlabel("angle $\\theta$ (radians)")
ax3.set_ylabel("intensity")
ax3.set_title("anisotropy parameter")


# Plot the angular distribution
plt.subplots_adjust(left=0.06, bottom=0.17, right=0.95, top=0.89,
                    wspace=0.35, hspace=0.37)

# Save a image of the plot
plt.savefig("plot_example_O2_PES_PAD.png", dpi=100)

# Show the plots
plt.show()
```

## 7.4 Example: Hansen–Law xenon

```
# -*- coding: utf-8 -*-

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import matplotlib.pyplot as plt

import abel
import scipy.misc

# This example demonstrates Hansen and Law inverse Abel transform
# of an image obtained using a velocity map imaging (VMI) photoelecton
# spectrometer to record the photoelectron angular distribution resulting
# from photodetachement of O2- at 454 nm.
# This spectrum was recorded in 2010
# ANU / The Australian National University
# J. Chem. Phys. 133, 174311 (2010) DOI: 10.1063/1.3493349

filename = 'data/Xenon_ATI_VMI_800_nm_649x519.tif'

# Name the output files
name = filename.split('.')[0].split('/')[1]
```

(continues on next page)

(continued from previous page)

```
output_image = name + '_inverse_Abel_transform_HansenLaw.png'
output_text  = name + '_speeds_HansenLaw.dat'
output_plot  = 'plot_' + name + '_comparison_HansenLaw.png'

print('Loading ' + filename)
#im = np.loadtxt(filename)
im = plt.imread(filename)
(rows,cols) = np.shape(im)
print ('image size {:d}x{:d}'.format(rows,cols))


# Step 2: perform the Hansen & Law transform!
print('Performing Hansen and Law inverse Abel transform:')

recon = abel.Transform(im, method="hansenlaw", direction="inverse",
                       symmetry_axis=None, verbose=True,
                       center=(240,340)).transform

r, speeds = abel.tools.vmi.angular_integration(recon)

# Set up some axes
fig = plt.figure(figsize=(15,4))
ax1 = plt.subplot(131)
ax2 = plt.subplot(132)
ax3 = plt.subplot(133)

# raw data
im1 = ax1.imshow(im, origin='lower', aspect='auto')
fig.colorbar(im1, ax=ax1, fraction=.1, shrink=0.9, pad=0.03)
ax1.set_xlabel('x (pixels)')
ax1.set_ylabel('y (pixels)')
ax1.set_title('velocity map image')

# 2D transform
im2 = ax2.imshow(recon, origin='lower', aspect='auto')
fig.colorbar(im2, ax=ax2, fraction=.1, shrink=0.9, pad=0.03)
ax2.set_xlabel('x (pixels)')
ax2.set_ylabel('y (pixels)')
ax2.set_title('Hansen Law inverse Abel')

# 1D speed distribution
ax3.plot(speeds)
ax3.set_xlabel('Speed (pixel)')
ax3.set_ylabel('Yield (log)')
ax3.set_title('Speed distribution')
#ax3.set_yscale('log')

# Prettify the plot a little bit:
plt.subplots_adjust(left=0.06, bottom=0.17, right=0.95, top=0.89, wspace=0.35,
                    hspace=0.37)

# Save a image of the plot
plt.savefig(output_plot, dpi=100)

# Show the plots
plt.show()
```

## 7.5 Example: Basex Gaussian

```
import numpy as np
import matplotlib.pyplot as plt
import abel

# This example performs a BASEX transform of a simple 1D Gaussian function and␣
↪compares
# this to the analytical inverse Abel transform

fig, ax= plt.subplots(1,1)
plt.title('Abel tranforms of a gaussian function')

# Analytical inverse Abel:
n = 101
r_max = 20
sigma = 10

ref = abel.tools.analytical.GaussianAnalytical(n, r_max, sigma,symmetric=False)

ax.plot(ref.r, ref.func, 'b', label='Original signal')
ax.plot(ref.r, ref.abel, 'r', label='Direct Abel transform x0.05 [analytical]')

center = n//2

# BASEX Transform:
# Calculate the inverse abel transform for the centered data
recon = abel.basex.basex_transform(ref.abel, verbose=True, basis_dir=None,
        dr=ref.dr, direction='inverse')

ax.plot(ref.r, recon , 'o',color='red', label='Inverse transform [BASEX]', ms=5, mec=
↪'none',alpha=0.5)

ax.legend()

ax.set_xlim(0,20)
ax.set_xlabel('x')
ax.set_ylabel('f(x)')

plt.legend()
plt.show()
```

## 7.6 Example: Basex photoelectron

```
# -*- coding: utf-8 -*-

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import os.path
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import abel

# This example demonstrates a BASEX transform of an image obtained using a
# velocity map imaging (VMI) photoelecton spectrometer to record the
# photoelectron angualar distribution resulting from above threshold ionization (ATI)
# in xenon gas using a ~40 femtosecond, 800 nm laser pulse.
# This spectrum was recorded in 2012 in the Kapteyn-Murnane research group at
# JILA / The University of Colorado at Boulder
# by Dan Hickstein and co-workers (contact DanHickstein@gmail.com)
# http://journals.aps.org/prl/abstract/10.1103/PhysRevLett.109.073004
#
# Before you start your own transform, identify the central pixel of the image.
# It's nice to use a program like ImageJ for this.
# http://imagej.nih.gov/ij/

# Specify the path to the file
filename = os.path.join('data', 'Xenon_ATI_VMI_800_nm_649x519.tif')

# Name the output files
output_image = filename[:-4] + '_Abel_transform.png'
output_text  = filename[:-4] + '_speeds.txt'
output_plot  = filename[:-4] + '_comparison.pdf'

# Step 1: Load an image file as a numpy array
print('Loading ' + filename)
raw_data = plt.imread(filename).astype('float64')

# Step 2: Specify the center in y,x (vert,horiz) format
center = (245,340)
# or, use automatic centering
# center = 'com'
# center = 'gaussian'

# Step 3: perform the BASEX transform!
print('Performing the inverse Abel transform:')

recon = abel.Transform(raw_data, direction='inverse', method='basex',
                       center=center, transform_options=dict(basis_dir='bases'),
                       verbose=True).transform

speeds = abel.tools.vmi.angular_integration(recon)

# Set up some axes
fig = plt.figure(figsize=(15,4))
ax1 = plt.subplot(131)
ax2 = plt.subplot(132)
ax3 = plt.subplot(133)

# Plot the raw data
im1 = ax1.imshow(raw_data,origin='lower',aspect='auto')
fig.colorbar(im1,ax=ax1,fraction=.1,shrink=0.9,pad=0.03)
ax1.set_xlabel('x (pixels)')
ax1.set_ylabel('y (pixels)')

# Plot the 2D transform
im2 = ax2.imshow(recon,origin='lower',aspect='auto',clim=(0,2000))
fig.colorbar(im2,ax=ax2,fraction=.1,shrink=0.9,pad=0.03)
```

(continues on next page)

(continued from previous page)

```
ax2.set_xlabel('x (pixels)')
ax2.set_ylabel('y (pixels)')

# Plot the 1D speed distribution

ax3.plot(*speeds)
ax3.set_xlabel('Speed (pixel)')
ax3.set_ylabel('Yield (log)')
ax3.set_yscale('log')
#ax3.set_ylim(1e2,1e5)

# Prettify the plot a little bit:
plt.subplots_adjust(left=0.06,bottom=0.17,right=0.95,top=0.89,wspace=0.35,hspace=0.37)

# Show the plots
plt.show()
```

## 7.7 Example: All Dribinski

```
# -*- coding: utf-8 -*-

# This example compares the available inverse Abel transform methods
# for the Ominus sample image
#
# Note it transforms only the Q0 (top-right) quadrant
# using the fundamental transform code

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel

import collections
import matplotlib.pylab as plt
from time import time

fig, (ax1,ax2) = plt.subplots(1, 2, figsize=(8,4))

# inverse Abel transform methods -----------------------------
#   dictionary of method: function()

transforms = {
  "direct": abel.direct.direct_transform,
  "hansenlaw": abel.hansenlaw.hansenlaw_transform,
  "onion": abel.dasch.onion_peeling_transform,
  "basex": abel.basex.basex_transform,
  "three_point": abel.dasch.three_point_transform,
  "two_point": abel.dasch.two_point_transform,
}

# sort dictionary:
```

(continues on next page)

(continued from previous page)

```
transforms = collections.OrderedDict(sorted(transforms.items()))
# number of transforms:
ntrans = np.size(transforms.keys())

IM = abel.tools.analytical.SampleImage(n=301, name="dribinski").image

h, w = IM.shape

# forward transform:
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

Q0, Q1, Q2, Q3 = abel.tools.symmetry.get_image_quadrants(fIM, reorient=True)

Q0fresh = Q0.copy()    # keep clean copy
print ("quadrant shape {}".format(Q0.shape))

# process Q0 quadrant using each method --------------------

iabelQ = []  # keep inverse Abel transformed image

for q, method in enumerate(transforms.keys()):

    Q0 = Q0fresh.copy()   # top-right quadrant of O2- image

    print ("\n------- {:s} inverse ...".format(method))
    t0 = time()

    # inverse Abel transform using 'method'
    IAQ0 = transforms[method](Q0, direction="inverse", basis_dir='bases')

    print ("                    {:.4f} sec".format(time()-t0))

    iabelQ.append(IAQ0)  # store for plot

    # polar projection and speed profile
    radial, speed = abel.tools.vmi.angular_integration(IAQ0, origin=(0, 0),␣
↪Jacobian=False)

    # normalize image intensity and speed distribution
    IAQ0 /= IAQ0.max()
    speed /= speed.max()

    # method label for each quadrant
    annot_angle = -(45+q*90)*np.pi/180  # -ve because numpy coords from top
    if q > 3:
        annot_angle += 50*np.pi/180    # shared quadrant - move the label
    annot_coord = (h/2+(h*0.9)*np.cos(annot_angle)/2 -50,
                   w/2+(w*0.9)*np.sin(annot_angle)/2)
    ax1.annotate(method, annot_coord, color="yellow")

    # plot speed distribution
    ax2.plot(radial, speed, label=method)

# reassemble image, each quadrant a different method

# for < 4 images pad using a blank quadrant
blank = np.zeros(IAQ0.shape)
```

(continues on next page)

(continued from previous page)

```
for q in range(ntrans, 4):
    iabelQ.append(blank)

# more than 4, split quadrant
if ntrans == 5:
    # split last quadrant into 2 = upper and lower triangles
    tmp_img = np.tril(np.flipud(iabelQ[-2])) +\
              np.triu(np.flipud(iabelQ[-1]))
    iabelQ[3] = np.flipud(tmp_img)

im = abel.tools.symmetry.put_image_quadrants((iabelQ[0], iabelQ[1],
                                              iabelQ[2], iabelQ[3]),
                                              original_image_shape=IM.shape)

ax1.imshow(im, vmin=0, vmax=0.15)
ax1.set_title('Inverse Abel comparison')


ax2.set_xlim(0, 200)
ax2.set_ylim(-0.5,2)
ax2.legend(loc=0, labelspacing=0.1, frameon=False)
ax2.set_title('Angular integration')
ax2.set_xlabel('Radial coordinate (pixel)')
ax2.set_ylabel('Integrated intensity')


plt.suptitle('Dribinski sample image')

plt.tight_layout()
plt.savefig('plot_example_all_dribinski.png', dpi=100)
plt.show()
```

## 7.8 Example: Dasch methods

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

"""example_dasch_methods.py.
"""

import numpy as np
import abel
import matplotlib.pyplot as plt

# Dribinski sample image size 501x501
n = 501
IM = abel.tools.analytical.SampleImage(n).image

# split into quadrants
origQ = abel.tools.symmetry.get_image_quadrants(IM)

# speed distribution of original image
```

(continues on next page)

(continued from previous page)

```
orig_speed = abel.tools.vmi.angular_integration(origQ[0], origin=(0,0))
scale_factor = orig_speed[1].max()

plt.plot(orig_speed[0], orig_speed[1]/scale_factor, linestyle='dashed',
         label="Dribinski sample")


# forward Abel projection
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

# split projected image into quadrants
Q = abel.tools.symmetry.get_image_quadrants(fIM)

dasch_transform = {\
"two_point": abel.dasch.two_point_transform,
"three_point": abel.dasch.three_point_transform,
"onion_peeling": abel.dasch.onion_peeling_transform}

for method in dasch_transform.keys():
    Q0 = Q[0].copy()
# method inverse Abel transform
    AQ0 = dasch_transform[method](Q0, basis_dir='bases')
# speed distribution
    speed = abel.tools.vmi.angular_integration(AQ0, origin=(0,0))

    plt.plot(speed[0], speed[1]*orig_speed[1][14]/speed[1][14]/scale_factor,
             label=method)

plt.title("Dasch methods for Dribinski sample image $n={:d}$".format(n))
plt.axis(xmax=250, ymin=-0.1)
plt.legend(loc=0, frameon=False, labelspacing=0.1, fontsize='small')
plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```

## 7.9 Example: Onion Bordas

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel
import matplotlib.pyplot as plt

# Dribinski sample image
IM = abel.tools.analytical.SampleImage(n=501).image

# split into quadrants
origQ = abel.tools.symmetry.get_image_quadrants(IM)

# speed distribution
orig_speed = abel.tools.vmi.angular_integration(origQ[0], origin=(0,0))
```

(continues on next page)

(continued from previous page)

```
# forward Abel projection
fIM = abel.Transform(IM, direction="forward", method="hansenlaw").transform

# split projected image into quadrants
Q = abel.tools.symmetry.get_image_quadrants(fIM)
Q0 = Q[0].copy()

# onion_bordas inverse Abel transform
borQ0 = abel.onion_bordas.onion_bordas_transform(Q0)
# speed distribution
bor_speed = abel.tools.vmi.angular_integration(borQ0, origin=(0,0))

plt.plot(*orig_speed, linestyle='dashed', label="Dribinski sample")
plt.plot(bor_speed[0], bor_speed[1], label="onion_bordas")
plt.axis(ymin=-0.1)
plt.legend(loc=0)
plt.savefig("plot_example_onion_bordas.png",dpi=100)
plt.show()
```

## 7.10 Example: Linbasex

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel
import os
import bz2

import matplotlib.pylab as plt

# This example demonstrates ``linbasex`` inverse Abel transform
# of a velocity-map image of photoelectrons from O2- photodetachment at 454 nm.
# Measured at  The Australian National University
# J. Chem. Phys. 133, 174311 (2010) DOI: 10.1063/1.3493349

# Load image as a numpy array - numpy handles .gz, .bz2
imagefile = bz2.BZ2File('data/O2-ANU1024.txt.bz2')
IM = np.loadtxt(imagefile)

if os.environ.get('READTHEDOCS', None) == 'True':
    IM = IM[::2,::2]
# the [::2, ::2] reduces the image size x1/2, decreasing processing memory load
# for the online readthedocs.org

# Image center should be mid-pixel and the image square,
# `center=convolution` takes care of this

un = [0, 2]  # spherical harmonic orders
proj_angles = np.arange(0, 2*np.pi, np.pi/20) # projection angles
# adjust these parameter to 'improve' the look
smoothing = 0.9  # smoothing Gaussian 1/e width
```

(continues on next page)

(continued from previous page)

```
threshold = 0.01 # exclude small amplitude Newton spheres
# no need to change these
radial_step = 1
clip = 0

# linbasex inverse Abel transform
LIM = abel.Transform(IM, method="linbasex", center="convolution",
                     center_options=dict(square=True),
                     transform_options=dict(basis_dir=None, return_Beta=True,
                                            legendre_orders=un,
                                            proj_angles=proj_angles,
                                            smoothing=smoothing,
                                            radial_step=radial_step, clip=clip,
                                            threshold=threshold))

# angular, and radial integration - direct from `linbasex` transform
# as class attributes
radial = LIM.radial
speed  = LIM.Beta[0]
anisotropy = LIM.Beta[1]

# normalize to max intensity peak i.e. max peak height = 1
speed /= speed[200:].max()  # exclude transform noise near centerline of image

# plots of the analysis
fig = plt.figure(figsize=(11, 5))
ax1 = plt.subplot2grid((1, 2), (0, 0))
ax2 = plt.subplot2grid((1, 2), (0, 1))

# join 1/2 raw data : 1/2 inversion image
inv_IM = LIM.transform
cols = inv_IM.shape[1]
c2 = cols//2
vmax = IM[:, :c2-100].max()
inv_IM *= vmax/inv_IM[:, c2+100:].max()
JIM = np.concatenate((IM[:, :c2], inv_IM[:, c2:]), axis=1)

# raw data
im1 = ax1.imshow(JIM, origin='upper', aspect='auto', vmin=0, vmax=vmax)
ax1.set_xlabel('column (pixels)')
ax1.set_ylabel('row (pixels)')
ax1.set_title('VMI, inverse Abel: {:d}x{:d}'.format(*inv_IM.shape),
              fontsize='small')

# Plot the 1D speed distribution and anisotropy parameter ("looks" better
# if multiplied by the intensity)
ax2.plot(radial, speed, label='speed')
ax2.plot(radial, speed*anisotropy, label=r'anisotropy $\times$ speed')
ax2.set_xlabel('radial pixel')
row, cols = IM.shape
ax2.axis(xmin=100*cols/1024, xmax=500*cols/1024, ymin=-1.5, ymax=1.8)
ax2.set_title("speed, anisotropy parameter", fontsize='small')
ax2.set_ylabel('intensity')
ax2.set_xlabel('radial coordinate (pixels)')

plt.legend(loc='best', frameon=False, labelspacing=0.1, fontsize='small')
plt.suptitle(
```

(continues on next page)

(continued from previous page)

```
r'linbasex inverse Abel transform of O$_{2}{}^{-}$ electron velocity-map image',
             fontsize='larger')

# Save a image of the plot
plt.savefig("plot_example_linbasex.png", dpi=100)

# Show the plots
plt.show()
```

## 7.11 Example: Anisotropy parameter

```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import abel
import bz2

import matplotlib.pylab as plt

# Demonstration of two techniques to determine the anisotropy parameter
# (a) directly, using `linbasex`
# (b) from the inverse Abel transformed image

# Load image as a numpy array
imagefile = bz2.BZ2File('data/O2-ANU1024.txt.bz2')
IM = np.loadtxt(imagefile)
# use scipy.misc.imread(filename) to load image formats (.png, .jpg, etc)

# === linbasex transform ===================================
legendre_orders = [0, 2, 4]  # Legendre polynomial orders
proj_angles = range(0, 180, 10)  # projection angles in 10 degree steps
radial_step = 1  # pixel grid
smoothing = 0.9  # smoothing 1/e-width for Gaussian convolution smoothing
threshold = 0.2  # threshold for normalization of higher order Newton spheres
clip = 0  # clip first vectors (smallest Newton spheres) to avoid singularities

# linbasex method - center and center_options ensure image has odd square shape
LIM = abel.Transform(IM, method='linbasex', center='slice',
                     center_options=dict(square=True),
                     transform_options=dict(basis_dir=None,
                     proj_angles=proj_angles, radial_step=radial_step,
                     smoothing=smoothing, threshold=threshold, clip=clip,
                     return_Beta=True, verbose=True))


# === Hansen & Law inverse Abel transform ==================
HIM = abel.Transform(IM, center="slice", method="hansenlaw",
                     symmetry_axis=None, angular_integration=True)

# speed distribution
radial, speed = HIM.angular_integration
```

(continues on next page)

(continued from previous page)

```
# normalize to max intensity peak
speed /= speed[200:].max()  # exclude transform noise near centerline of image

# PAD - photoelectron angular distribution from image ======================
# Note: `linbasex` provides the anisotropy parameter directly LIM.Beta[1]
#       here we extract I vs theta for given radial ranges
#       and use fitting to determine the anisotropy parameter
#
# radial ranges (of spectral features) to follow intensity vs angle
# view the speed distribution to determine radial ranges
r_range = [(145, 162), (200, 218), (230, 250), (255, 280), (280, 310),
           (310, 330), (330, 350), (350, 370), (370, 390), (390, 410),
           (410, 430)]

# anisotropy parameter from image for each tuple r_range
Beta, Amp, Rmid, Ivstheta, theta =\
              abel.tools.vmi.radial_integration(HIM.transform, r_range)

# OR  anisotropy parameter for ranges (0, 20), (20, 40) ...
# Beta_whole_grid, Amp_whole_grid, Radial_midpoints =\
#                          abel.tools.vmi.anisotropy(AIM.transform, 20)

# Radial intensity and anisotropy distributions
I, beta2 = abel.tools.vmi.Ibeta(HIM.transform, window=9)
# normalize to max intensity peak
I /= I.max()
# remove (noisy) anisotropy values for low-intensity parts
beta2[I < 0.01] = np.nan

# plots of the analysis
fig = plt.figure(figsize=(8, 4))
ax1 = plt.subplot(121)
ax2 = plt.subplot(122)

# join 1/2 raw data : 1/2 inversion image
rows, cols = IM.shape
c2 = cols//2
vmax = IM[:, :c2-100].max()
AIM = HIM.transform
AIM *= vmax/AIM[:, c2+100:].max()
JIM = np.concatenate((IM[:, :c2], AIM[:, c2:]), axis=1)

# Plot the image data VMI | inverse Abel
im1 = ax1.imshow(JIM, origin='lower', aspect='auto', vmin=0, vmax=vmax)
fig.colorbar(im1, ax=ax1, fraction=.1, shrink=0.9, pad=0.03)
ax1.set_xlabel('x (pixels)')
ax1.set_ylabel('y (pixels)')
ax1.set_title('VMI, inverse Abel: {:d}×{:d}'.format(rows, cols))

# Plot the 1D speed distribution
line01, = ax2.plot(LIM.Beta[0], 'r-', label='linbasex-Beta[0]')
line02, = ax2.plot(speed, 'b-', label='speed')
line03, = ax2.plot(I, 'c--', label='$I(r)$')
legend0 = ax2.legend(handles=[line01, line02, line03],
                     frameon=False, labelspacing=0.1, numpoints=1, loc=2,
                     fontsize='small')
```

(continues on next page)

(continued from previous page)

```
plt.gca().add_artist(legend0)

# Plot anisotropy parameter, attribute Beta[1], x speed
line11, = ax2.plot(LIM.Beta[1], 'r-', label='linbasex-Beta[2]')
BetaT = np.transpose(Beta)
line12 = ax2.errorbar(Rmid, BetaT[0], BetaT[1], fmt='.', color='g',
                      label='specific radii')
# ax2.plot(Radial_midpoints, Beta_whole_grid[0], '-g', label='stepped')
line13, = ax2.plot(beta2, 'c', label=r'$\beta_2(r)$')
legend1 = ax2.legend(handles=[line11, line12, line13],
                     frameon=False, labelspacing=0.1, numpoints=1, loc=3,
                     fontsize='small')

ax2.axis(xmin=100, xmax=450, ymin=-1.2, ymax=1.2)
ax2.set_xlabel('radial pixel')
ax2.set_ylabel('speed/anisotropy')
ax2.set_title('speed/anisotropy distribution')

plt.subplots_adjust(left=0.06, bottom=0.17, right=0.95, top=0.89,
                    wspace=0.35, hspace=0.37)

# Save a image of the plot
plt.savefig("plot_example_PAD.png", dpi=100)

# Show the plots
plt.show()
```

## 7.12 Example: Circularize image

```
import numpy as np
import matplotlib.pyplot as plt
import abel
import scipy.interpolate

#######################################################################
#
# example_circularize_image.py
#
# O- sample image -> forward Abel + distortion = measured VMI
#  measured VMI   -> inverse Abel transform -> speed distribution
# Compare disorted and circularized speed profiles
#
#######################################################################


# sample image -----------
IM = abel.tools.analytical.SampleImage(n=511, name='Ominus', sigma=2).image

# forward transform == what is measured
IMf = abel.Transform(IM, method='hansenlaw', direction="forward").transform

# flower image distortion
def flower_scaling(theta, freq=2, amp=0.1):
    return 1 + amp*np.sin(freq*theta)**4
```

(continues on next page)

(continued from previous page)

```
# distort the image
IMdist = abel.tools.circularize.circularize(IMf,
                                radial_correction_function=flower_scaling)

# circularize ------------
IMcirc, sla, sc, scspl = abel.tools.circularize.circularize_image(IMdist,
               method='lsq', dr=0.5, dt=0.1, smooth=0, return_correction=True)

# inverse Abel transform for distored and circularized images ---------
AIMdist = abel.Transform(IMdist, method="three_point",
                         transform_options=dict(basis_dir='bases')).transform
AIMcirc = abel.Transform(IMcirc, method="three_point",
                         transform_options=dict(basis_dir='bases')).transform

# respective speed distributions
rdist, speeddist = abel.tools.vmi.angular_integration(AIMdist, dr=0.5)
rcirc, speedcirc = abel.tools.vmi.angular_integration(AIMcirc, dr=0.5)

# note the small image size is responsible for the slight over correction
# of the background near peaks

row, col = IMcirc.shape

# plot --------------------

fig, axs = plt.subplots(2, 2, figsize=(8, 8))
fig.subplots_adjust(wspace=0.5, hspace=0.5)

extent = (np.min(-col//2), np.max(col//2), np.min(-row//2), np.max(row//2))
axs[0, 0].imshow(IMdist, aspect='auto', origin='lower', extent=extent)
axs[0, 0].set_title("Ominus distorted sample image")

axs[0, 1].imshow(AIMcirc, vmin=0, aspect='auto', origin='lower',
                 extent=extent)
axs[0, 1].set_title("circ. + inv. Abel")

axs[1, 0].plot(sla, sc, 'o')
ang = np.arange(-np.pi, np.pi, 0.1)
axs[1, 0].plot(ang, scspl(ang))
axs[1, 0].set_xticks([-np.pi, 0, np.pi])
axs[1, 0].set_xticklabels([r"$-\pi$", "0", r"$\pi$"])
axs[1, 0].set_xlabel("angle (radians)")
axs[1, 0].set_ylabel("radial correction factor")
axs[1, 0].set_title("radial correction")

axs[1, 1].plot(rdist, speeddist, label='dist.')
axs[1, 1].plot(rcirc, speedcirc, label='circ.')
axs[1, 1].axis(xmin=100, xmax=240)
axs[1, 1].set_title("speed distribution")
axs[1, 1].legend(frameon=False)
axs[1, 1].set_xlabel('radius (pixels)')
axs[1, 1].set_ylabel('intensity')

plt.savefig("plot_example_circularize_image.png", dpi=75)
plt.show()
```

CHAPTER 8

# Contributing to PyAbel

PyAbel is an open source project and we welcome improvements! Please let us know about any issues with the software, even if's just a typo. The easiest way to get started is to open a new issue.

If you would like to make a Pull Request, the following information may be useful.

## 8.1 Unit tests

Before submitting at Pull Request, be sure to run the unit tests. The test suite can be run from within the PyAbel package with

```
pytest
```

For more detailed information, the following can be used:

```
pytest abel/  -v  --cov=abel
```

Note that this requires that you have pytest and (optionally) pytest-cov installed. You can install these with

```
pip install pytest pytest-cov
```

## 8.2 Documentation

PyAbel uses Sphinx and Napoleon to process Numpy style docstrings, and is synchronized to pyabel.readthedocs.io. To build the documentation locally, you will need Sphinx, the recommonmark package, and the sphinx_rtd_theme. You can install all this this using

```
pip install sphinx
pip install recommonmark
pip install sphinx_rtd_theme
```

Once you have that installed, then you can build the documentation using

```
cd PyAbel/doc/
 make html
```

Then you can open `doc/_build/hmtl/index.html` to look at the documentation. Sometimes you need to use

```
make clean
make html
```

to clear out the old documentation and get things to re-build properly.

When you get tired of typing `make html` every time you make a change to the documentation, it's nice to use use sphix-autobuild to automatically update the documentation in your browser for you. So, install sphinx-autobuild using

```
pip install sphinx-autobuild
```

Now you should be able to

```
cd PyAbel/doc/
make livehtml
```

which should launch a browser window displaying the docs. When you save a change to any of the docs, the re-build should happen automatically and the docs should update in a matter of a few seconds.

Alternatively, restview is a nice way to preview the `.rst` files.

## 8.3 Before merging

If possible, before merging your pull request please rebase your fork on the last master on PyAbel. This could be done as explained in this post

```
# Add the remote, call it "upstream" (only the fist time)
git remote add upstream https://github.com/PyAbel/PyAbel.git

# Fetch all the branches of that remote into remote-tracking branches,
# such as upstream/master:

git fetch upstream

# Make sure that you're on your master branch
# or any other branch your are working on

git checkout master  # or your other working branch

# Rewrite your master branch so that any commits of yours that
# aren't already in upstream/master are replayed on top of that
# other branch:

git rebase upstream/master

# push the changes to your fork

git push -f
```

See this wiki for more information.

## 8.4 Adding a new forward or inverse Abel implementation

We are always looking for new implementation of forward or inverse Abel transform, therefore if you have an implementation that you would want to contribute to PyAbel, don't hesitate to do so.

In order to allow a consistent user experience between different implementations, and insure an overall code quality, please consider the following points in your pull request.

### 8.4.1 Naming conventions

The implementation named *<implementation>*, located under *abel/<implementation>.py* should use the following naming system for top-level functions,

- `<implemenation>_transform` : core transform (when defined)
- `_bs_<implementation>` : function that generates the basis sets (if necessary)

### 8.4.2 Unit tests

To detect issues early, the submitted implementation should have the following properties and pass the corresponding unit tests,

1. The reconstruction has the same shape as the original image. Currently all transform methods operate with odd-width images and should raise an exception if provided with an even-width image.
2. Given an array of 0 elements, the reconstruction should also be a 0 array.
3. The implementation should be able to calculated the inverse (or forward) transform of a Gaussian function defined by a standard deviation `sigma`, with better than a `10 %` relative error with respect to the analytical solution for `0 > r > 2*sigma`.

Unit tests for a given implementation are located under `abel/tests/test_<implemenation>.py`, which should contain at least the following 3 functions `test_<implementation>_shape`, `test_<implementation>_zeros`, `test_<implementation>_gaussian`. See `abel/tests/test_basex.py` for a concrete example.

## 8.5 Dependencies

The current list of dependencies can be found in setup.py. Please refrain from adding new dependencies, unless it cannot be avoided.

## 8.6 Change Log

If the change is significant (more than just a typo-fix), please leave a short note about the change in CHANGELOG.rst

## 8.7 Releasing on PyPi

PyAbel should be automatically released on PyPi (see PR #161) whenever a new release is drafted on GitHub via the "Draft New Release" button on the Releases page. Just remember to increment the version number in abel/_version.py first.

## 8.8 Citations

Each version of PyAbel that is released triggers a new DOI on Zenodo, so that people can cite the project. If you would like you name added to the author list on Zenodo, please include it in `.zenodo.json`.

# CHAPTER 9

# Indices and tables

- genindex
- modindex
- search

# Bibliography

[Dribinski2002] Dribinski et al, 2002 (Rev. Sci. Instrum. 73, 2634), (pdf)

# Python Module Index

# Index