PyAbel Documentation

Release 0.9.0

PyAbel team

Nov 10, 2023

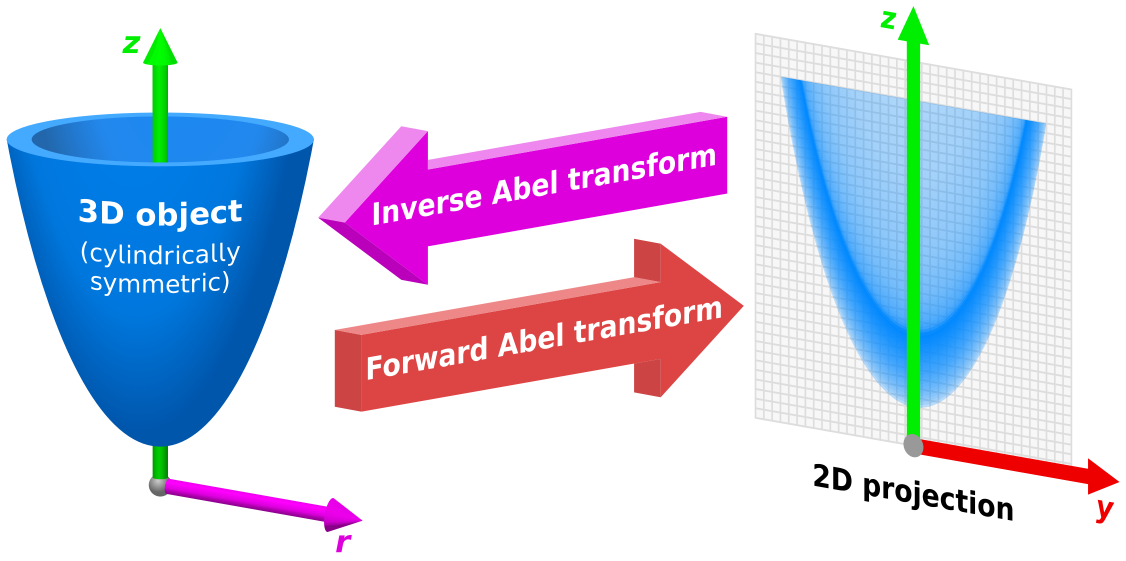
Contents

Chapter 1

PyAbel README

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.*
9. `rbasex` – a pBasex-like method formulated in terms of radial distributions.
10. `daun` – the regularized deconvolution method by Daun and co-workers, with additional capabilities.

1.3 Installation

PyAbel requires Python 3.7–3.12. (Note: PyAbel is also currently tested to work with Python 2.7, but Python 2 support will be removed soon.) 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.

The latest release can be installed from PyPI with

```
pip install PyAbel
```

If you prefer the development version from GitHub, download it here (clicking the [Code ▼] button), `cd` to the PyAbel directory, and use

```
pip install .
```

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

```
pip install -e .
```

1.3.1 Before uninstalling

Some transform methods can save generated basis sets to disk. If you want to uninstall PyAbel completely, these files need to be removed as well. To do so, please *first* run the following script:

```
import abel
import shutil
shutil.rmtree(abel.transform.get_basis_dir())
```

and *then* proceed with the usual module uninstallation process (for example, `pip uninstall PyAbel` if it was installed using pip).

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 an inverse transform using the Three Point method:

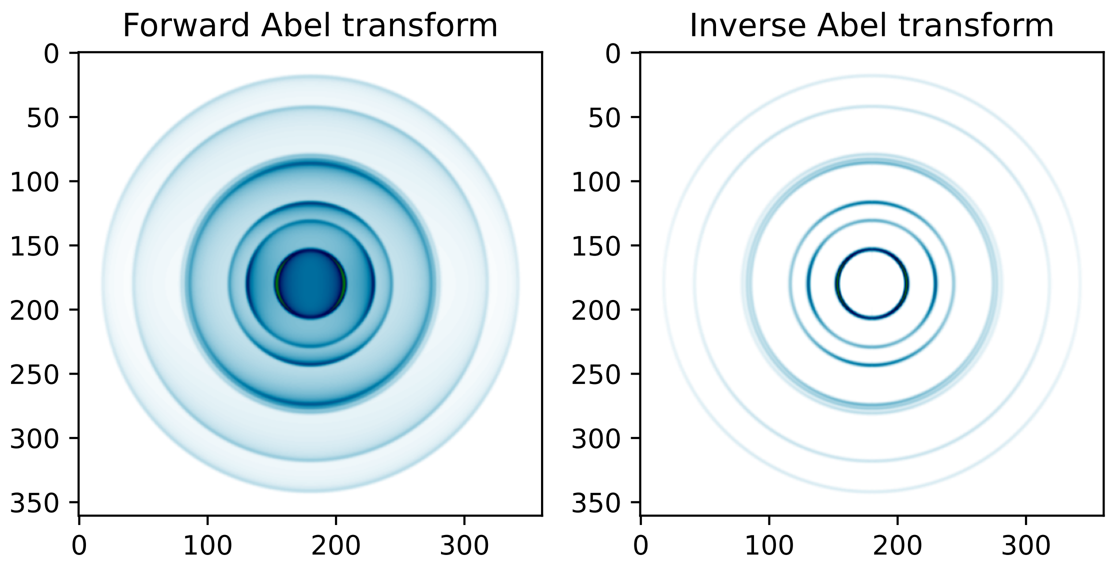
```
import abel
original = abel.tools.analytical.SampleImage(name='Gerber').func
forward_abel = abel.Transform(original, direction='forward',
                              method='hansenlaw').transform
inverse_abel = abel.Transform(forward_abel, direction='inverse',
                              method='three_point').transform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

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, 3))

axs[0].imshow(forward_abel, clim=(0, None), cmap='ocean_r')
axs[1].imshow(inverse_abel, clim=(0, None), cmap='ocean_r')

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 in *PyAbel examples*, 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.) Image coordinates are in the (row, column) format, consistent with NumPy array indexing, and negative values are interpreted as relative to the end of the corresponding axis. For example, `(-1, 0)` refers to the lower left corner (last row, 0th column). Cartesian coordinates can also be generated if needed. For example, the x, y grid for a centered 5×5 image:

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

  The `abel.tools.polar.index_coords` function does this for images of any shape with any origin.

- **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 centered 5×5 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)
```

  The `abel.tools.polar.cart2polar` and `abel.tools.polar.polar2cart` functions are available for conversion between these Cartesian and polar grids.

- **Image origin:** Fundamentally, the forward and inverse Abel transforms in PyAbel consider the origin 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 central pixel is effectively "shared" between both halves of the image.) In most situations, the image origin is specified using the `origin` keyword in `abel.Transform` (or directly using `abel.tools.center.center_image` to find the origin (the center of symmetry) of your image). This processing step takes care of shifting the origin of the image to 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. Some methods also provide low-level functions for transforming only the right half of the image (with the origin located in the middle of a 0th-column pixel).

- **Intensity:** The pixel intensities can have any value (within the floating-point range). However, the intensity scale must be linear. Keep in mind that cameras and common image formats often use gamma correction and thus provide data with nonlinear intensity encoding. Thus, if possible, it is recommended to disable the gamma correction on cameras used to record images that will be inverse Abel-transformed. If this is not possible, then it is necessary to apply the appropriate intensity transformations before the analysis. Most PyAbel methods also assume intensities to be floating-point numbers, and when applied to integer types, can return inappropriately rounded results. The `abel.Transform` class recasts the input image to `float64` by default, but if you wish to call the transform methods directly or use other tools, you might need to perform the conversion yourself (as `IM.astype(float)`, for example).

1.7 Support

If you have a question about using PyAbel, the best way to contact the PyAbel Developers Team is through GitHub discussions. To report a bug or make a suggestion, please 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 to PyAbel has more information on how to contribute, such as how to run the unit tests and how to build the documentation.

1.9 License

PyAbel 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 at Zenodo.

Additionally, we have written a scientific paper comparing various Abel transform methods. You can find the manuscript at the Review of Scientific Instruments (DOI: 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='inverse'*, *method='three_point'*, *origin='none'*, *symmetry_axis=None*, *use_quadrants=(True*, *True*, *True*, *True)*, *symmetrize_method='average'*, *angular_integration=False*, *transform_options={}*, *center_options={}*, *angular_integration_options={}*, *recast_as_float64=True*, *verbose=False*, *center=<deprecated>*)

Bases: `object`

Abel transform image class. Also accessible as `abel.Transform`.

This class provides whole-image forward and inverse Abel transforms, together with preprocessing (centering, symmetrizing) and postprocessing (integration) functions.

Parameters

- **IM** (*a N×M 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` (default)
  An inverse Abel transform takes a 2D projection and reconstructs a 2D slice of the 3D image.

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

  `basex`
  the Gaussian "basis set expansion" method of Dribinski et al. (2002).

  `daun`
  the deconvolution method with Tikhonov regularization of Daun et al. and its extensions.

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

  `hansenlaw`
  the recursive algorithm described by Hansen and Law (1985).

`linbasex`
the 1D projections of velocity-mapping images in terms of 1D spherical functions by Gerber et al. (2013).

`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).

`rbasex`
a method similar to pBasex by Garcia et al. (2004) for velocity-mapping images, but with analytical basis functions developed by Ryazanov (2012).

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

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

- **origin** (*tuple or str*) – Before applying Abel transform, the image is centered around this point.

  If a tuple (float, float) is provided, this specifies the image origin in the (row, column) format. If a string is provided, an automatic centering algorithm is used:

  `image_center`
  The origin is assumed to be the center of the image.

  `convolution`
  The origin is found from autoconvolution of image projections along each axis.

  `slice`
  The origin is found by comparing slices in the horizontal and vertical directions.

  `com`
  The origin is calculated as the center of mass.

  `gaussian`
  The origin 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). Note that the Abel transform is always performed around the vertical axis. This parameter only affects how the image is modified before (and after) applying the Abel transform. For more information, see the "Quadrant combining" note below.

- **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 2D projection should be real. Removing the imaginary components in reciprocal space leaves a symmetric projection.

K. R. Overstreet, P. Zabawa, J. Tallant, A. Schwettmann, J. P. Shaffer, "Multiple scattering and the density distribution of a Cs MOT", Optics Express 13, 9672–9682 (2005).

- **angular_integration** (*bool*) – Integrate the image over angle to give the radial (speed) intensity distribution.

  Note: in PyAbel ≤0.8.4 the intensity distribution was off by a factor of π, please keep this in mind when comparing absolute intensities.

- **transform_options** (*dict*) – Additional arguments passed to the individual transform functions. See the documentation for the individual transform method for options: *basex*, *daun*, *direct*, *hansenlaw*, *linbasex*, *onion_bordas*, *onion_peeling*, *rbasex*, *three_point*, *two_point*.
- **center_options** (*dict*) – Additional arguments to be passed to the centering function, see *abel.tools.center.center_image()*.
- **angular_integration_options** (*dict*) – Additional arguments passed to the angular integration functions, see *abel.tools.vmi.angular_integration_3D()*.
- **recast_as_float64** (*bool*) – determines whether 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).
- **verbose** (*bool*) – determines whether 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 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.

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 be reloaded, meaning that future transforms are performed much more quickly.

`basex` *
: The "basis set exapansion" algorithm describes the data in terms of gaussian-like functions, which themselves can be Abel-transformed analytically. With the default functions, centered at 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.

  V. Dribinski, A. Ossadtchi, V. A. Mandelshtam, H. Reisler, "Reconstruction of Abel-transformable images: The Gaussian basis-set expansion Abel transform method", Rev. Sci. Instrum. 73, 2634–2642 (2002).

`daun` *
: Methods based on onion-peeling deconvolution using Tikhonov regularization described in

  K. J. Daun, K. A. Thomson, F. Liu, G. J. Smallwood, "Deconvolution of axisymmetric flame properties using Tikhonov regularization", Appl. Opt. 45, 4638–4646 (2006).

  In addition to the original implicit step-functions basis ("onion peeling") and the derivative regularization, linear and quadratic basis functions are implemented, as well as the L_2-norm Tikhonov regularization (like in `basex`) and non-negative least-squares solution.

`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.

`hansenlaw`
: This "recursive algorithm" produces reliable results and is quite fast (~0.1 s for a 1001×1001 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, P.-L. Law, "Recursive methods for computing the Abel transform and its inverse", J. Opt. Soc. Am. A 2, 510–520 (1985).

`linbasex` *

Velocity-mapping 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. Lin-BASEX evaluates 1D projections of VM images in terms of 1D projections of spherical functions, instead.

Th. Gerber, Yu. Liu, G. Knopp, P. Hemberger, A. Bodi, P. Radi, Ya. Sych, “Charged particle velocity map image reconstruction with one-dimensional projections of spherical functions”, Rev. Sci. Instrum. 84, 033101 (2013).

`onion_bordas`

The onion peeling method, also known as “back projection”, originates from C. Bordas, F. Paulig, “Photoelectron imaging spectrometry: Principle and inversion method”, Rev. Sci. Instrum. 67, 2257–2268 (1996).

The algorithm was subsequently coded in MatLab by C. E. Rallis, T. G. Burwitz, P. R. Andrews, M. Zohrabi, R. Averin, S. De, B. Bergues, B. Jochim, A. V. Voznyuk, N. Gregerson, B. Gaire, I. Znakovskaya, J. McKenna, K. D. Carnes, M. F. Kling, I. Ben-Itzhak, E. Wells, “Incorporating real time velocity map image reconstruction into closed-loop coherent control”, 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 transform achieved in one Python code-line, PR #155. See also `three_point` is the onion peeling algorithm as described by Dasch (1992), reference below.

`rbasex` *

The pBasex method by G. A. Garcia, L. Nahon, I. Powis, “Two-dimensional charged particle image inversion using a polar basis function expansion”, Rev. Sci. Instrum. 75, 4989–2996 (2004) adapts the BASEX (“basis set expansion”) method to the specific case of velocity-mapping images by using a basis of 2D functions in polar coordinates, such that the reconstructed radial distributions are obtained directly from the expansion coefficients.

This method employs the same approach, but uses more convenient basis functions, which have analytical Abel transforms separable into radial and angular parts, developed in M. Ryazanov, “Development and implementation of methods for sliced velocity map imaging. Studies of overtone-induced dissociation and isomerization dynamics of hydroxymethyl radical (CH_2OH and CD_2OH)”, Ph.D. dissertation, University of Southern California, 2012 (ProQuest, USC).

`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.

C. J. Dasch, “One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods”, Appl. Opt. 31, 1146–1152 (1992).

`two_point` *

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

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

Returns

- **transform** (*numpy 2D array*) – the 2D forward/inverse Abel-transformed image.
- **angular_integration** (*tuple*) – (radial-grid, radial-intensity) radial coordinates and the radial intensity (speed) distribution, evaluated using *`abel.tools.vmi.angular_integration_3D()`*.
- **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.
- **radial** (*numpy 1D array*) – with `method='linbasex'`: radial grid for **Beta** array
- **Beta** (*numpy 2D array*) – with `method='linbasex'`: 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]
- **projection** (*numpy 2D array*) – with `method='linbasex'`: radial projection profiles at angles **proj_angles**
- **distr** (*Distributions.Results object*) – with `method='rbasex'`: the object from which various radial distributions can be retrieved

`abel.transform.`**`set_basis_dir`**(*basis_dir=''*, *make=True*)

Changes the path to the directory for saving/loading cached basis sets that transform methods use by default.

Parameters

- **basis_dir** (*str or None*) – absolute or relative path. Passing `''` (default) resets to the system-dependent default path, see `default_basis_dir()`. For the current working directory (as in PyAbel up to v0.8.4), use `'.'`. To disable basis-set caching on disk, use `None`.
- **make** (*bool*) – create the directory if it does not exist (default: yes)

Return type

None

`abel.transform.`**`get_basis_dir`**(*make=False*)

Gets the path to the directory for saving/loading cached basis sets that transform methods use by default. If not changed by `set_basis_dir()`, it depends on the operating system, see `default_basis_dir()`.

Parameters

make (*bool*) – create the directory if it does not exist (default: no)

Returns

path – absolute or relative path if disk caching is enabled, otherwise `None`

Return type

str or None

`abel.transform.`**`default_basis_dir`**()

Gets full path to the system-dependent default directory for saving/loading cached basis sets:

Linux (and other Unix-like):

`~/.cache/PyAbel` (or `$XDG_CACHE_HOME/PyAbel` if set)

macOS:

`/Users/<user>/Library/Caches/PyAbel`

Windows:

`<user profile>\AppData\Local\PyAbel\cache` (or `%LOCALAPPDATA%\PyAbel\cache` if set). See important notes below.

Parameters
None

Returns
path – full path to the system-dependent default basis-sets directory

Return type
str

Notes for MS Windows users

- Python installed from Microsoft Store redirects subdirectory creation in `AppData\Local` to a "private per-user, per-app location" `AppData\Local\Packages\Python...\LocalCache\Local` (see Using Python on Windows / Known Issues). However, if `AppData\Local\PyAbel\` already exists (for example, was manually created *not* from Python), apparently it should be usable.
- Old Windows versions (2000, XP, Server 2003) by default don't set the `LOCALAPPDATA` environment variable, so PyAbel will create and use the `AppData\Local` subtree in the user profile folder. This is probably fine, but not how it should be. To use the standard location, please do

  ```
  set LOCALAPPDATA=%USERPROFILE%\Local Settings\Application Data
  ```

  before starting Python. Or permanently set it in "Environment Variables" from Windows "System Properties".

`abel.transform.`**`basis_dir_cleanup`**(*basis_dir=''*, *method=None*)

Deletes saved basis sets.

Parameters

- **basis_dir** (*str or None*) – path to the directory with saved basis sets. Use `''` for the default directory, see *`get_basis_dir()`*. (For convenience, `None` can be passed to do nothing.)
- **method** (*str or list of str or None*) – transform methods for which basis sets should be deleted. Can be a single string (see the `method` parameter in *`Transform`*) or a list of strings. Use `'all'` to delete basis sets for all methods. `None` does nothing.

Return type
None

2.2 abel.basex module

`abel.basex.`**`basex_transform`**(*data*, *sigma=1.0*, *reg=0.0*, *correction=True*, *basis_dir=''*, *dr=1.0*, *verbose=True*, *direction='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 $\ll 1$.
- **correction** (*boolean*) – apply intensity correction in order to reduce method artifacts (intensity normalization and oscillations)
- **basis_dir** (*str or None*) – path to the directory for saving / loading the basis sets. Use `''` for the default directory. If `None`, the basis set will not be loaded from or 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=''*, *dr=1.0*, *verbose=False*, *direction='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 or None*) – path to the directory for saving / loading the basis sets. Use `''` for the default directory. If `None`, the basis sets will not be loaded from or 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='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.

Return type
None

`abel.basex.`**`basis_dir_cleanup`**(*basis_dir=''*)

Utility function.

Deletes basis sets saved on disk.

Parameters
basis_dir (*str or None*) – absolute or relative path to the directory with saved basis sets. Use `''` for the default directory. `None` does nothing.

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.dasch module

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

The *two-point deconvolution method*.

C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

Parameters

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str or None*) – 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. Use `''` for the default directory. If `None`, the operator array will not be loaded from or 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=''*, *dr=1*, *direction='inverse'*, *verbose=False*)

The *three-point deconvolution method*.

C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

Parameters

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str or None*) – 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. Use `''` for the default directory. If `None`, the operator array will not be loaded from or 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=''*, *dr=1*, *direction='inverse'*, *verbose=False*`)`

The *onion-peeling deconvolution method*.

C. J. Dasch, “One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods”, Appl. Opt. 31, 1146–1152 (1992).

Parameters

- **IM** (*1D or 2D numpy array*) – right-side half-image (or quadrant)
- **basis_dir** (*str or None*) – 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. Use `''` for the default directory. If `None`, the operator array will not be loaded from or 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=''*, *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. Use `''` for the default directory. For `None`, do not load or 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

Return type
None

`abel.dasch.`**`basis_dir_cleanup`**`(`*method*, *basis_dir=''*`)`

Utility function.

Deletes deconvolution operator arrays saved on disk.

Parameters

- **method** (*str*) – Abel transform method `'onion_peeling'`, `'three_point'`, or `'two_point'`
- **basis_dir** (*str or None*) – absolute or relative path to the directory with saved deconvolution operator arrays. Use `''` for the default directory. `None` does nothing.

Return type
None

2.4 abel.daun module

`abel.daun.`**`daun_transform`**`(`*data*, *reg=0.0*, *degree=0*, *dr=1.0*, *direction='inverse'*, *basis_dir=None*, *verbose=True*`)`

Forward and inverse Abel transforms based on onion-peeling deconvolution using Tikhonov regularization described in

K. J. Daun, K. A. Thomson, F. Liu, G. J. Smallwood, "Deconvolution of axisymmetric flame properties using Tikhonov regularization", Appl. Opt. 45, 4638–4646 (2006).

with additional basis-function types and regularization methods (see *description*).

This function operates on the "right side" of an image, that it, just one half of a cylindrically symmetric image, with the axial pixels located in the 0-th column.

Parameters

- **data** (*m × n numpy array*) – the image to be transformed. `data[:, 0]` should correspond to the central column of the image.

- **reg** (*float or tuple or str*) – regularization for the inverse transform:

  `strength`:
  same as `('diff', strength)`

  `('diff', strength)`:
  Tikhonov regularization using the first-order difference operator (first-derivative approximation), as described in the original article

  `('L2', strength)`:
  Tikhonov regularization using the L_2 norm, like in *BASEX*

  `('L2c', strength)`:
  same as `('L2', strength)`, but with an intensity correction applied to compensate the drop near the symmetry axis

  `'nonneg'`:
  non-negative least-squares solution.

  Warning: this regularization method is very slow, typically taking up to a minute for a megapixel image.

- **degree** (*int*) – degree of basis-function polynomials:

  0:
  rectangular functions (step-function approximation), corresponding to “onion peeling” from the original article

  1:
  triangular functions (piecewise linear approximation)

  2:
  piecewise quadratic functions (smooth approximation)

  3:
  piecewise cubic functions (cubic-spline approximation)

- **dr** (*float*) – pixel size in the radial direction. This only affects the absolute scaling of the transformed image.
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed
- **basis_dir** (*str, optional*) – path to the directory for saving / loading the basis set (potentially useful only for **degree** = 3 and transform without regularization; time savings in other cases are small and might be negated by the disk-access overhead). Use `''` for the default directory. If `None` (default), the basis set will not be loaded from or saved to disk.
- **verbose** (*bool*) – determines whether progress report should be printed

Returns
recon – the transformed (half) image

Return type
m × n numpy array

`abel.daun.`**`get_bs_cached`**(*n*, *degree=0*, *reg_type='diff'*, *strength=0*, *direction='inverse'*, *basis_dir=None*, *verbose=False*)

Internal function.

Gets the basis set and calculates the necessary transform matrix (notice that inverse direction with `'nonneg'` regularization, as well as with **strength** = 0 for **degree** $\neq$ 3, gives the forward (triangular) matrix, to be used in solvers).

Parameters

- **n** (*int*) – half-width of the image in pixels, must include the axial pixel
- **degree** (*int*) – polynomial degree for basis functions (0–3)
- **reg_type** (*None or str*) – regularization type (`None`, `'diff'`, `'L2'`, `'L2c'`, `'nonneg'`)
- **strength** (*float*) – Tikhonov regularization parameter (for **reg_type** = `'diff'` and `'L2'`/`'L2c'`, ignored otherwise)
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed
- **basis_dir** (*str or None*) – path to the directory for saving / loading the basis set. Use `''` for the default directory. If `None`, the basis sets will not be loaded from or saved to disk.
- **verbose** (*bool*) – print some debug information

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

Return type
n × n numpy array

`abel.daun.`**`cache_cleanup`**`(`*select='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 set

`'inverse'`
only inverse transform

Return type
None

`abel.daun.`**`basis_dir_cleanup`**`(`*basis_dir=''*`)`

Utility function.

Deletes basis sets saved on disk.

Parameters
basis_dir (*str or None*) – absolute or relative path to the directory with saved basis sets. Use `''` for the default directory. `None` does nothing.

Return type
None

2.5 abel.direct module

`abel.direct.`**`direct_transform`**`(`*fr*, *dr=None*, *r=None*, *direction='inverse'*, *derivative=<function gradient>*, *int_func=<function trapz>*, *correction=True*, *backend='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 a compiled C version using Cython, which is much faster. The implementation can be selected using the backend argument. If the C-backend is not available, you must re-install PyAbel with Numpy, Cython, and a C-compiler already installed.

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.

2.6 abel.hansenlaw module

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

Forward/Inverse Abel transformation using the algorithm from

E. W. Hansen, "Fast Hankel transform algorithm", IEEE Trans. Acoust. Speech Signal Proc. 33, 666–671 (1985)

and

E. W. Hansen, 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 origin `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

2.7 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='inverse'*, *verbose=False*, *dr=None*)

Wrapper function for linbasex to process a single image quadrant in the upper right orientation (Q0). *Is not applicable to images with odd Legendre orders.*

Parameters not described below are passed directly to `linbasex_transform_full()`.

Parameters

- **IM** (*numpy 2D array*) – upper right quadrant of the image data, must be square in shape
- **return_Beta** (*bool*) – in addition to the transformed image, return the **radial**, **Beta** and **projections** arrays
- **dr** (*any*) – dummy variable for call compatibility with the other methods

Returns

- **inv_IM** (*numpy 2D array*) – upper right quadrant of the inverse Abel transformed image
- **radial** (*numpy 1D array*) – (only if **return_Beta** = `True`) radii of each Newton sphere
- **Beta** (*numpy 2D array*) – (only 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 **radial** is the speed distribution

  Beta[1] vs **radial** is the anisotropy of each Newton sphere

- **projections** (*numpy 2D array*) – (only if **return_Beta** = `True`) 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=<deprecated>, norm_range=(0, -1), direction='inverse', verbose=False*)

Inverse Abel transform using 1D projections of images.

Th. Gerber, Yu. Liu, G. Knopp, P. Hemberger, A. Bodi, P. Radi, Ya. Sych, "Charged particle velocity map image reconstruction with one-dimensional projections of spherical functions", Rev. Sci. Instrum. 84, 033101 (2013).

Lin-Basex 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.

This function operates on the whole image.

Parameters

- **IM** (*numpy 2D array*) – image data must have square shape of odd size
- **basis_dir** (*str or None*) – path to the directory for saving / loading the basis sets. Use `''` for the default directory. If `None` (default), the basis set will not be loaded from or saved to disk.
- **proj_angles** (*list of float*) – 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 of int*) – 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 $1/e$ halfwidth equal to **smoothing**.
- **rcond** (*float*) – (default 0.0005) `scipy.linalg.lstsq()` fit conditioning value. Use 0 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.
- **threshold** (*float*) – threshold for normalization of higher-order Newton spheres (default 0.2): if **Beta[0] < threshold**, the associated **Beta[j]** for all j $\geqslant$ 1 are set to zero
- **clip** (*int*) – clip first vectors (smallest Newton spheres) to avoid singularities (default 0)
- **norm_range** (*tuple of int*) – (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.
- **direction** (*str*) – Abel transform direction. Only "inverse" is implemented.
- **verbose** (*bool*) – print information about processing (normally used for debugging)

Returns

- **inv_IM** (*numpy 2D array*) – inverse Abel transformed image
- **radial** (*numpy 1D array*) – radii of each Newton sphere
- **Beta** (*numpy 2D array*) – 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 **radial** is the speed distribution

  Beta[1] vs **radial** is the anisotropy of each Newton sphere
- **projections** (*numpy 2D array*) – 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.

Warning: This function is deprecated and will be remove in the future. See issue #356.

For integrating the speed distribution and averaging the anisotropy, please use `mean_beta()`.

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.`**`mean_beta`**(*radial*, *Beta*, *regions*)

Integrate normalized intensity (`Beta[0]`) and perform intensity-weighted averaging of anisotropy (`Beta[1:]`) over ranges of Newton spheres.

Parameters

- **radial** (*numpy 1D array*) – radii of Newton spheres
- **Beta** (*numpy 2D array*) – speed and anisotropy distribution from *`linbasex_transform_full()`*
- **regions** (*list of tuple of int*) – radial ranges [(min0, max0), (min1, max1), . . .]. Note that inclusion of regions where **Beta[0]** is below **threshold** set in *`linbasex_transform_full()`* will bias the mean anisotropies towards zero.

Returns
Beta_mean – overall intensity (`Beta_mean[0]`) and mean anisotropy values (`Beta_mean[1:]`) in each region

Return type
2D 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}_{clip}*.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 or None*) – path to the directory for saving / loading the basis. Use `''` for the default directory. If `None`, the basis set will not be loaded from or saved to disk.
- **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_angles}_{radial_step}_{clip}.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

Return type
None

`abel.linbasex.`**`basis_dir_cleanup`**`(`*basis_dir=''*`)`

Utility function.

Deletes basis sets saved on disk.

Parameters
basis_dir (*str or None*) – relative or absolute path to the directory with saved basis sets. Use `''` for the default directory. `None` does nothing.

Return type
None

2.8 abel.onion_bordas module

`abel.onion_bordas.`**`onion_bordas_transform`**`(`*IM*, *dr=1*, *direction='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

C. E. Rallis, T. G. Burwitz, P. R. Andrews, M. Zohrabi, R. Averin, S. De, B. Bergues, B. Jochim, A. V. Voznyuk, N. Gregerson, B. Gaire, I. Znakovskaya, J. McKenna, K. D. Carnes, M. F. Kling, I. Ben-Itzhak, E. Wells, "Incorporating real time velocity map image reconstruction into closed-loop coherent control", Rev. Sci. Instrum. 85, 113105 (2014).

The algorithm actually originates from

C. Bordas, F. Paulig, "Photoelectron imaging spectrometry: Principle and inversion method", Rev. Sci. Instrum. 67, 2257–2268 (1996).

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 image origin should be at the left edge, not mid-pixel. This corresponds to an even-width full image.

However, `shift_grid=True` (default) provides the typical behavior, where the image origin corresponds to the pixel center in the 0th column.

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 inverse transform is currently implemented.
- **shift_grid** (*bool*) – place the image origin on the grid (left edge) by shifting the image 1/2 pixel to the left.

Returns
AIM – the inverse Abel transformed half-image

Return type
1D or 2D numpy array

2.9 abel.rbasex module

`abel.rbasex.`**`rbasex_transform`**(*IM*, *origin='center'*, *rmax='MIN'*, *order=2*, *odd=False*, *weights=None*, *direction='inverse'*, *reg=None*, *out='same'*, *basis_dir=None*, *verbose=False*)

rBasex Abel transform for velocity-mapping images, operating in polar coordinates.

This function takes the input image and outputs its forward or inverse Abel transform as an image and its radial distributions.

The **origin**, **rmax**, **order**, **odd** and **weights** parameters are passed to *`abel.tools.vmi.Distributions`*, so see its documentation for their detailed descriptions.

Parameters

- **IM** (*m × n numpy array*) – the image to be transformed
- **origin** (*tuple of int or str*) – image origin, explicit in the (row, column) format, or as a location string (by default, the image center)
- **rmax** (*int or string*) – largest radius to include in the transform (by default, the largest radius with at least one full quadrant of data)
- **order** (*int*) – highest angular order present in the data, ≥ 0 (by default, 2). Working with very high orders ($\gtrsim 15$) can result in excessive noise, especially at small radii and for narrow peaks.
- **odd** (*bool*) – include odd angular orders (by default is *False*, but is enabled automatically if **order** is odd)
- **weights** (*m × n numpy array, optional*) – weighting factors for each pixel. The array shape must match the image shape. Parts of the image can be excluded from analysis by assigning zero weights to their pixels. By default is *None*, which applies equal weight to all pixels.
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed (by default, inverse)
- **reg** (*None or str or tuple (str, float), optional*) – regularization to use for inverse Abel transform. `None` (default) means no regularization, a string selects a non-parameterized regularization method, and parameterized methods are selected by a tuple (*method*, *strength*). Available methods are:

  `('L2', strength):`
  Tikhonov L_2 regularization with *strength* as the square of the Tikhonov factor. This is the same as “Tikhonov regularization” used in BASEX, with almost identical effects on the radial distributions.

  `('diff', strength):`
  Tikhonov regularization with the difference operator (approximation of the derivative)

multiplied by the square root of *strength* as the Tikhonov matrix. This tends to produce less blurring, but more negative overshoots than `'L2'`.

`('SVD', strength)`:
truncated SVD (singular value decomposition) with N = *strength* × **rmax** largest singular values removed for each angular order. This mimics the approach proposed (but in fact not used) in pBasex. *Not recommended* due to generally poor results.

`'pos'`:
non-parameterized method, finds the best (in the least-squares sense) solution with non-negative $\cos^n\theta\sin^m\theta$ terms (see `cossin()`). For **order** = 0, 1, and 2 (with **odd** = *False*) this is equivalent to $I(r,\theta) \geqslant 0$; for higher orders this assumption is stronger than $I \geqslant 0$ and corresponds to no interference between different multiphoton channels. Not implemented for odd orders > 1.

Notice that this method is nonlinear, which also means that it is considerably slower than the linear methods and the transform operator cannot be cached.

In all cases, *strength* = 0 provides no regularization. For the Tikhonov methods, *strength* ~ 100 is a reasonable value for megapixel images. For truncated SVD, *strength* must be < 1; *strength* ~ 0.1 is a reasonable value; *strength* ~ 0.5 can produce noticeable ringing artifacts. See the *full description* and examples there.

- **out** (*str or None*) – shape of the output image:

  `'same'` (default):
  same shape and origin as the input

  `'fold'` (fastest):
  Q0 (upper right) quadrant (for `odd=False`) or right half (for `odd=True`) up to **rmax**, but limited to the largest input-image quadrant (or half)

  `'unfold'`:
  like `'fold'`, but symmetrically “unfolded” to all 4 quadrants

  `'full'`:
  all pixels with radii up to **rmax**

  `'full-unique'`:
  the unique part of `'full'`: Q0 (upper right) quadrant for `odd=False`, right half for `odd=True`

  `None`:
  no image (**recon** will be `None`). Can be useful to avoid unnecessary calculations when only the transformed radial distributions (**distr**) are needed.

- **basis_dir** (*str, optional*) – path to the directory for saving / loading the basis set (useful only for the inverse transform without regularization; time savings in other cases are small and might be negated by the disk-access overhead). Use `''` for the default directory. If `None` (default), the basis set will not be loaded from or saved to disk.
- **verbose** (*bool*) – print information about processing (for debugging), disabled by default

Returns

- **recon** (*2D numpy array or None*) – the transformed image. Is centered and might have different dimensions than the input image.
- **distr** (*Distributions.Results object*) – the object from which various distributions for the transformed image can be retrieved, see `abel.tools.vmi.Distributions.Results`

`abel.rbasex.`**`get_bs_cached`**(*Rmax*, *order=2*, *odd=False*, *direction='inverse'*, *reg=None*, *valid=None*, *basis_dir=None*, *verbose=False*)

Internal function.

Gets the basis set (from cache or runs computations and caches them) and calculates the transform matrix. Loaded/calculated matrices are also cached in memory.

Parameters

- **Rmax** (*int*) – largest radius to be transformed
- **order** (*int*) – highest angular order
- **odd** (*bool*) – include odd angular orders
- **direction** (str: `'forward'` or `'inverse'`) – type of Abel transform to be performed
- **reg** (*None or str or tuple (str, float)*) – regularization type and strength for inverse transform
- **valid** (*None or bool array*) – flags to exclude invalid radii from transform
- **basis_dir** (*str, optional*) – path to the directory for saving / loading the basis set. Use `''` for the default directory. If `None`, the basis set will not be loaded from or saved to disk.
- **verbose** (*bool*) – print some debug information

Returns

A – (**Rmax** + 1) × (**Rmax** + 1) matrices of the Abel transform (forward or inverse) for each angular order

Return type

list of 2D numpy arrays

`abel.rbasex.`**`cache_cleanup`**(*select='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.

Return type

None

`abel.rbasex.`**`basis_dir_cleanup`**(*basis_dir=''*)

Utility function.

Deletes basis sets saved on disk.

Parameters

basis_dir (*str or None*) – absolute or relative path to the directory with saved basis sets. Use `''` for the default directory. `None` does nothing.

Return type

None

Chapter 3

Image processing tools

3.1 abel.tools.analytical module

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

Bases: `object`

Base class for functions that have a known Abel transform (see *`GaussianAnalytical`* for a concrete example). Every such class should expose the following public attributes:

`r`

vector of positions along the r axis

Type
numpy array

`func`

the values of the original function (same shape as *`r`* for 1D functions, or same row size as *`r`* for 2D images)

Type
numpy array

`abel`

the values of the Abel transform (same shape as *`func`*)

Type
numpy array

`mask_valid`

mask (same shape as *`func`*) where the function is well smoothed/well behaved (no known artefacts in the inverse Abel reconstuction), typically excluding the origin, the domain boundaries, and function discontinuities, that can be used for unit testing.

Type
numpy array

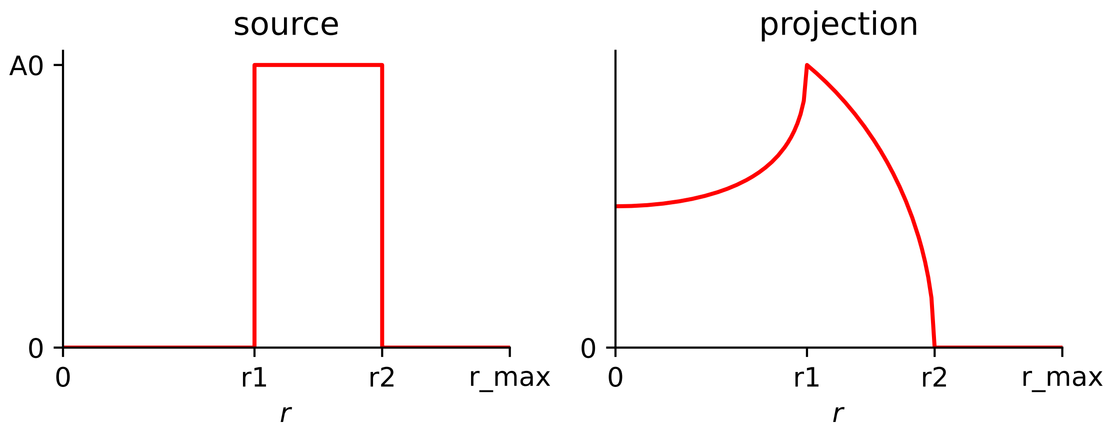
Parameters

- **n** (*int*) – number of points along the r axis (saved to attribute `n`)
- **r_max** (*float*) – maximum r value (saved to attribute `r_max`)
- **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**]

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

Bases: *`BaseAnalytical`*

Define a step function and calculate its analytical Abel transform:



See *examples/example_basex_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 for $r > 0$ (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.

Parameters

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

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: *`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
- **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)
- **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**]

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

Bases: *`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
- **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
- **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**]

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

Bases: *`BaseAnalytical`*

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

Parameters

- **n** (*int*) – number of points along the r axis
- **r_max** (*float*) – range of the r interval
- **sigma** (*float*) – 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)
- **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]

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

Bases: `BaseAnalytical`

Abel-transform pair analytical functions.

profiles 1–7: Table 1 of G. C.-Y. Chan, Gary M. Hieftje, "Estimation of confidence intervals for radial emissivity and optimization of data treatment techniques in Abel inversion", 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=None, temperature=200)`

Bases: `BaseAnalytical`

Sample images, made up of Gaussian functions (or cubic splines, for `'O2'`).

Parameters

- **n** (*integer*) – image size **n** rows × **n** cols (must be odd for most purposes; even **n** values would result in half-pixel centering)
- **name** (*str*) –

  `'Dribinski'`

  Sample test image used in the BASEX paper Rev. Sci. Instrum. 73, 2634 (2002), intensity function Eq. (16) (there are some missing negative exponents in the publication).

  9 Gaussian peaks with alternating anisotropies ($\beta = -1, 0, +2$), plus 1 wide background Gaussian. Peak amplitudes are designed to produce comparable heights in the *speed distribution*, thus the peaks at small radii appear very intense in the image and its Abel transform.

`'Gaussian'`

Isotropic 2D Gaussian $\exp(-r^2/\mathbf{sigma}^2)$.

Its Abel transform is also a Gaussian with the same width: $\sqrt{\pi}\,\mathbf{sigma}\exp(-r^2/\mathbf{sigma}^2)$.

`'Gerber'`

Artificial test image used in the lin-BASEX article Rev. Sci. Instrum. 84, 033101 (2013), Table I.

8 Gaussian peaks with various intensities and anisotropies up to 4th order (β_4).

`'O2'`

Synthetic image mimicking a velocity-map image of O$^+$ from multiphoton photodissociation/ionization $\mathrm{O}_2 \xrightarrow{4h\nu} \mathrm{O} + \mathrm{O}^+ + e^-$ at 44 444 cm^{-1} (225 nm)

Multiple peaks with various intensities and anisotropies; see, for example, J. Chem. Phys. 107, 2357 (1997).

`'Ominus'` or `'O-'`

Simulate the photoelectron spectrum of O$^-$ photodetachment $^3P_J \leftarrow {}^2P_{3/2,1/2}$.

6 transitions, triplet neutral, and doublet anion.

- **sigma** (*float*) – 1/*e* halfwidth of peaks in pixels, default values are: $2{\cdot}r_{\max}/180$ for `'Dribinski'`, $2{\cdot}r_{\max}/500$ for `'Ominus'`, $r_{\max}/3$ for `'Gaussian'`, $\sqrt{2}$ (std. dev. = 1) for `'Gerber'`.

  For `'O2'`: HWHM of narrow peaks in pixels, default is 1.5 for any $r_{\max}$.

- **temperature** (*float*) – anion temperature in kelvins (default: 200) for `'Ominus'`: anion levels have Boltzmann population weight $(2J+1)\exp[-hc\cdot 177.1\text{ cm}^{-1}/(k\cdot\mathbf{temperature})]$

`name`

sample-image name

Type
str

`transform`(*tol=0.0048*)

Compute forward Abel transform of the image as an analytical Abel transform of its piecewise polynomial approximation (except `'Gaussian'` and `'O2'`, which are computed exactly).

Parameters
tol (*float*) – relative tolerance of the approximation (max. deviation divided by max. amplitude, default: 4.8e-3 $\lesssim$ 0.5%); the resulting Abel transform is somewhat more accurate

Returns
abel – Abel-transformed image, also accessible as the `abel` attribute

Return type
2D numpy array

`property image`

Deprecated. Use `func` instead.

`property abel`

Abel transform of the image, computed (with default accuracy) only if necessary; see `transform()` for details.

3.2 abel.tools.center module

`abel.tools.center.`**`find_origin`**(*IM*, *method='image_center'*, *axes=(0, 1)*, *verbose=False*, ***kwargs*)

Find the coordinates of image origin, using the specified method.

Parameters

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

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

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

  `convolution`
  the origin is found as the maximum of autoconvolution of the image projections along each axis.

  `gaussian`
  the origin 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.

- **axes** (*int or tuple of int*) – find origin coordinates: `0` (vertical), or `1` (horizontal), or `(0, 1)` (both vertical and horizontal).

Returns
out – coordinates of the origin of the image in the (row, column) format. For coordinates not in **axes**, the center of the image is returned.

Return type
(float, float)

`abel.tools.center.`**`center_image`**(*IM*, *method='com'*, *odd_size=True*, *square=False*, *axes=(0, 1)*, *crop='maintain_size'*, *order=3*, *verbose=False*, *center=<deprecated>*, ***kwargs*)

Center image with the custom value or by several methods provided in `find_origin()` function.

Parameters

- **IM** (*2D np.array*) – The image data.
- **method** (*str or tuple of float*) – either a tuple (float, float), the coordinate of the origin of the image in the (row, column) format, or a string to specify an automatic centering method:

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

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

  `convolution`
  the origin is found as the maximum of autoconvolution of the image projections along each axis.

`gaussian`
: the origin is extracted from 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`, the returned image will contain 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`, the returned image will have a square shape.
- **crop** (*str*) – determines how the image should be cropped. The options are:

  `maintain_size`
  return image of the same size. Some regions 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.

  See `set_center()` for examples.
- **axes** (*int or tuple of int*) – center image with respect to axis `0` (vertical), `1` (horizontal), or both axes `(0, 1)` (default). When specifying an explicit origin in **method**, unused coordinates can also be passed as `None`, for example, `method=(row, None)` or `method=(None, col)`.
- **order** (*int*) – interpolation order, see `set_center()` for details.

Returns
out – centered image

Return type
2D np.array

`abel.tools.center.set_center`(*data*, *origin*, *crop='maintain_size'*, *axes=(0, 1)*, *order=3*, *verbose=False*, *center=<deprecated>*)

Move image origin to mid-point of image (`rows // 2, cols // 2`).

Parameters

- **data** (*2D np.array*) – the image data
- **origin** (*tuple of float*) – (row, column) coordinates of the image origin. Coordinates set to `None` are ignored.
- **crop** (*str*) – determines how the image should be cropped. The options are:

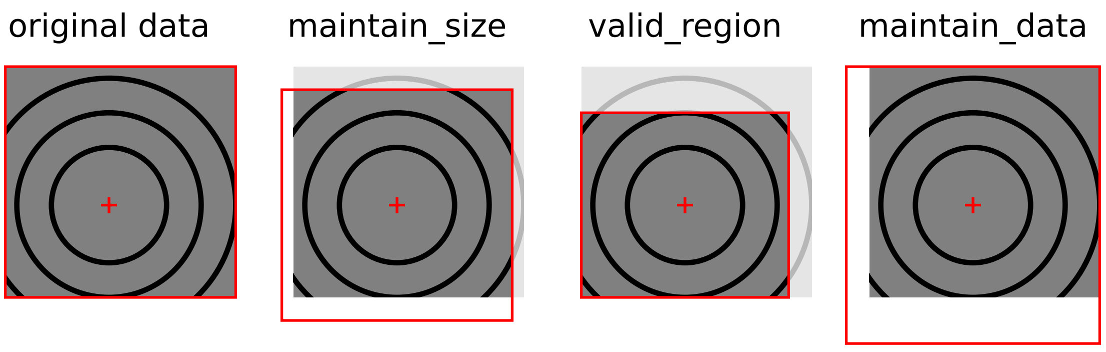
  `maintain_size` **(default)**
  return image of the same size. Some regions of the original image may be lost and some regions may be filled with zeros.

  `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.

Examples:



- **axes** (*int or tuple of int*) – center image with respect to axis `0` (vertical), `1` (horizontal), or both axes `(0, 1)` (default).
- **order** (*int*) – interpolation order (0–5, default is 3) for centering with fractional **origin**. Lower orders work faster; **order** = 0 (also implied for integer **origin**) means a whole-pixel shift without interpolation and is much faster.
- **verbose** (*bool*) – print some information for debugging

Returns
out – centered image

Return type
2D np.array

`abel.tools.center.`**`find_origin_by_center_of_mass`**(*IM*, *axes=(0, 1)*, *verbose=False*, *round_output=False*, ***kwargs*)

Find image origin by calculating its center of mass.

Parameters

- **IM** (*numpy 2D array*) – image data
- **axes** (*int or tuple*) – find origin coordinates: `0` (vertical), or `1` (horizontal), or `(0, 1)` (both vertical and horizontal).
- **round_output** (*bool*) – if `True`, the coordinates are rounded to integers; otherwise they are floats.

Returns
origin – (row, column)

Return type
(float, float)

`abel.tools.center.`**`find_origin_by_convolution`**(*IM*, *axes=(0, 1)*, *projections=False*, ***kwargs*)

Find the image origin as the maximum of autoconvolution of its projections along each axis.

Code from the `linbasex` juptyer notebook.

Parameters

- **IM** (*numpy 2D array*) – image data
- **projections** (*bool*) – if this parameter is `True`, the autoconvoluted projections along both axes will be returned after the origin.
- **axes** (*int or tuple*) – find origin coordinates: `0` (vertical), or `1` (horizontal), or `(0, 1)` (both vertical and horizontal).

Returns

origin – (row, column)

or (row, column), conv_0, conv_1

Return type
(float, float)

`abel.tools.center.`**`find_origin_by_center_of_image`**(*IM*, *axes=(0, 1)*, *verbose=False*, ***kwargs*)

Find image origin simply as its center, from its dimensions.

Parameters

- **IM** (*numpy 2D array*) – image data
- **axes** (*int or tuple*) – has no effect

Returns
origin – (row, column)

Return type
(int, int)

`abel.tools.center.`**`find_origin_by_gaussian_fit`**(*IM*, *axes=(0, 1)*, *verbose=False*, *round_output=False*, ***kwargs*)

Find image origin by fitting the summation along rows and columns of the data to two 1D Gaussian functions.

Parameters

- **IM** (*numpy 2D array*) – image data
- **axes** (*int or tuple*) – find origin coordinates: `0` (vertical), or `1` (horizontal), or `(0, 1)` (both vertical and horizontal).
- **round_output** (*bool*) – if `True`, the coordinates are rounded to integers; otherwise they are floats.

Returns
origin – (row, column)

Return type
(float, float)

`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 of float*) – (rmin, rmax) range to limit data
- **slice_width** (*int*) – 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_origin_by_slice`**(*IM*, *axes=(0, 1)*, *slice_width=10*, *radial_range=(0, -1)*, *axis=<deprecated>*, ***kwargs*)

Find the image origin by comparing opposite sides.

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.
- **axes** (*int or tuple*) – find origin coordinates: `0` (vertical), or `1` (horizontal), or `(0, 1)` (both vertical and horizontal).

Returns
origin – (row, column)

Return type
(float, float)

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

Deprecated function. Use *`find_origin()`* instead.

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

Deprecated function. Use *`find_origin_by_center_of_mass()`* instead.

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

Deprecated function. Use *`find_origin_by_convolution()`* instead.

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

Deprecated function. Use *`find_origin_by_center_of_image()`* instead.

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

Deprecated function. Use *`find_origin_by_gaussian_fit()`* instead.

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

Deprecated function. Use *`find_origin_by_slice()`* instead.

3.3 abel.tools.circularize module

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

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, S. T. Gibson, 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 **origin** 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.

- **origin** (*float tuple, str or None*) – Pre-center image using *`abel.tools.center.center_image()`*. May be an explicit (row, column) tuple or a method name: `'com'`, `'convolution'`, `'gaussian;`, `'image_center'`, `'slice'`. `None` (default) assumes that the image is already centered.
- **radial_range** (*tuple or None*) – Limit slice comparison to the radial range tuple (rmin, rmax), in pixels, from the image origin. 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 ratio is highest, with sharp persistent (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*) – Deprecated, use **tol** instead. The relationship is **smooth** = $N_{\text{angles}} \times$ **tol**^{2}, where N_{angles} is the number of slices (see **dt**).
- **ref_angle** (*None or float*) – Reference angle for which radial coordinate is unchanged. Angle varies between $-\pi$ and π, with zero angle vertical.

  `None` uses `numpy.mean()` of the radial correction function, 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 to the *polar*-coordinate image, to remove the background intensity. This may improve the signal-to-noise ratio, 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.

- **tol** (*float*) – Root-mean-square (RMS) fitting tolerance for the spline function. At the default zero value, the spline interpolates between the discrete scaling factors. At larger values, a smoother spline is found such that its RMS deviation from the discrete scaling factors does not exceed this number. For example, `tol=0.01` means 1% RMS tolerance for the radial scaling correction. At very large tolerances, the spline degenerates to a constant, the average of the discrete scaling factors.

  Typically, **tol** may remain zero (use interpolation), but noisy data may require some smoothing, since the found discrete scaling factors can have noticeable errors. To examine the relative scaling factors and how well they are represented by the spline function, use the option `return_correction=True`.

Returns

- **IMcirc** (*numpy 2D array*) – Circularized version of the input image, same size as input.
- 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** (*function(numpy 1D 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** (*function(numpy 1D array)*) – A function returning the radial correction for a given angle. It should accept a numpy 1D array of angles.
- **ref_angle** (*None or float*) – Reference angle at which the radial correction function is renormalized to unity. If `None`, the angular average is used for renormalization.

`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 (θ, 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.

Returns
radcorr – radial correction factors for angles

Return type
numpy 1D array

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

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

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.`**`guess_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

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

Deprecated function. Use `guess_gaussian()` instead.

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, with the pole placed at **origin** and the angle measured clockwise from the upward direction. The resulting array has rows corresponding to the radial grid, and columns corresponding to the angular grid.

Parameters

- **data** (*2D np.array*) – the image array
- **origin** (*tuple or None*) – (row, column) coordinates of the image origin. If `None`, the geometric center of the image is used.
- **Jacobian** (*bool*) – 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 or None*) – angular coordinate spacing (in radians). If `None`, the number of angular grid points will be set to the largest dimension (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 angular coordinates

Notes

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

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

Creates *x* and *y* coordinates for the indices in a numpy array, relative to the **origin**, with the *x* axis going to the right, and the *y* axis going *up*.

Parameters

- **data** (*numpy array*) – 2D data. Only the array shape is used.
- **origin** (*tuple or None*) – (row, column). Defaults to the geometric center of the image.

Returns
x, y

Return type
2D numpy 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.

`class` `abel.tools.polynomial.`**`BasePolynomial`**

Bases: `object`

Abstract base class for polynomials. Implements multiplication and division by numbers. (Addition and subtraction of polynomials are not implemented because they are meaningful only for polynomials generated on the same grid. Use `Piecewise...` classes for sums of polynomials.)

`func`

values of the original function

Type
numpy array

`abel`

values of the Abel transform

Type
numpy array

`copy()`

Return an independent copy.

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

Bases: `BasePolynomial`

Polynomial function and its Abel transform.

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; $0 \leq$ **r_min** < **r_max** $\lesssim$ **max r** (**r_max** might exceed maximal **r**, but usually by < 1 pixel; negative **r_min** or **r_max** are allowed for convenience but are interpreted as 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 $c_0 + c_1 ((r - \mathbf{r\_0})/\mathbf{s}) + c_2 ((r - \mathbf{r\_0})/\mathbf{s})^2 + \ldots$
- **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: `BasePolynomial`

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

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

p

`Polynomial` objects corresponding to each piece

Type

list of *Polynomial*

copy()

Make an independent copy.

class `abel.tools.polynomial.`**SPolynomial**(*r*, *cos*, *r_min*, *r_max*, *c*, *r_0=0.0*, *s=1.0*)

Bases: `BasePolynomial`

Bivariate polynomial function $\sum_{mn} c_{mn} r^m \cos^n\theta$ in spherical coordinates and its Abel transform.

Parameters

- **r, cos** (*numpy array*) – *r* and $\cos\theta$ values at which the function is generated; *r* must be non-negative. Arrays for generating a 2D image can be conveniently prepared by the `rcos()` function. On the other hand, the radial dependence alone (for a *single* cosine power) can be obtained by supplying a 1D **r** array and a **cos** array filled with ones.
- **r_min, r_max** (*float*) – *r* domain: the function is defined as the polynomial on [**r_min**, **r_max**] and zero outside it; $0 \le$ **r_min** < **r_max** $\lesssim$ **max r** (**r_max** might exceed maximal **r**, but usually by < 1 pixel; negative **r_min** or **r_max** are allowed for convenience but are interpreted as 0)
- **c** (*2D numpy array*) – polynomial coefficients for *r* and $\cos\theta$ powers: `c[m, n]` is the coefficient for the $r^m \cos^n\theta$ term. This array can be conveniently constructed using `Angular` tools.
- **r_0** (*float, optional*) – *r* domain shift: the polynomial is defined in powers of ($r -$ **r_0**) instead of *r*
- **s** (*float, optional*) – *r* stretching factor (around **r_0**): the polynomial is defined in powers of ($r -$ **r_0**)/**s** instead of *r*

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

Bases: `BasePolynomial`

Piecewise bivariate polynomial function (sum of `SPolynomial`s) in spherical coordinates and its Abel transform.

Parameters

- **r, cos** (*numpy array*) – *r* and $\cos\theta$ values at which the function is generated
- **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 `SPolynomial` conventions. All ranges are independent (may overlap and have gaps, may define polynomials of any degrees) and may include optional `SPolynomial` parameters (`r_0, s`).

`abel.tools.polynomial.rcos`(*rows=None*, *cols=None*, *shape=None*, *origin=None*)

Create arrays with polar coordinates r and $\cos\theta$: either from a pair of Cartesian arrays (**rows**, **cols**) with row and column values for each point *or* for a uniform grid with given dimensions and origin (**shape**, **origin**).

Parameters

- **rows, cols** (*numpy array*) – arrays with respectively row and column values for each point. Must have identical shapes (the output arrays will have the same shape), but might contain any values. For example, can be 2D arrays with integer pixel coordinates, or with floating-point numbers for sampling at subpixel points or on a distorted grid, or 1D arrays for sampling along some curve.
- **shape** (*tuple of int*) – (rows, cols) – create output arrays of given shape, with values corresponding to a uniform pixel grid.
- **origin** (*tuple of float, optional*) – position of the origin ($r = 0$) in the output array. By default, the center of the array is used (center of the middle pixel for odd-sized dimensions; even-sized dimensions will have a corresponding half-pixel shift).

Returns

- **r** (*numpy array*) – radii $r = \sqrt{\text{row}^2 + \text{col}^2}$ for each point
- **cos** (*numpy array*) – cosines of the polar angle $\cos\theta = -\text{row}/r$ for each point (by convention, $\cos\theta = 0$ at $r = 0$)

class `abel.tools.polynomial.Angular`(*c*)

Bases: `object`

Class helping to define angular dependences for *SPolynomial* and *PiecewiseSPolynomial*.

Supports arithmetic operations (addition, subtraction, multiplication of objects; multiplication and division by numbers) and outer product with radial coefficients (any list-like object). For example:

```
[3, 0, -1] * (Angular.cos(4) + Angular.sin(4) / 2)
```

represents $(3 - r^2)\big(\cos^4\theta + (\sin^4\theta)/2\big)$, producing

```
[[ 1.5  0.  -3.  0.   4.5]
 [ 0.   0.  -0.  0.   0. ]
 [-0.5  0.   1.  0.  -1.5]]
```

which can be supplied as the coefficient matrix to *SPolynomial*. Likewise, a list of ranges for *PiecewiseSPolynomial* can be prepared as an outer product with a list of `(r_min, r_max, coeffs)` tuples (with optional other *SPolynomial* parameters), where 1D `coeffs` contain radial coefficients for a polynomial segment.

Parameters

c (*float or iterable of float*) – list of coefficients: `Angular([c₀, c₁, c₂, ...])` means $c_0\cos^0\theta + c_1\cos^1\theta + c_2\cos^2\theta + \ldots$; `Angular(a)` represents the isotropic distribution $a \cdot \cos^0\theta$

c

coefficients for $\cos^n\theta$ powers, passed at instantiation directly (see above) or converted from other representations by the methods below.

Type

numpy array

classmethod `cos`(*n*)

Cosine power: `Angular.cos(n)` means $\cos^n\theta$.

classmethod sin(*n*)

Sine power: `Angular.sin(n)` means $\sin^n\theta$ (n must be even).

classmethod cossin(*m*, *n*)

Product of cosine and sine powers: `Angular.cossin(m, n)` means $\cos^m\theta\cdot\sin^n\theta$ (n must be even).

classmethod legendre(*c*)

Weighted sum of Legendre polynomials in $\cos\theta$: `Angular.legendre([c₀, c₁, c₂, ...])` means $c_0P_0(\cos\theta)+c_1P_1(\cos\theta)+c_2P_2(\cos\theta)+\dots$

This method is intended to be called like

```
Angular.legendre([1, β₁, β₂, ...])
```

where β_i are so-called anisotropy parameters. However, if you really need a single polynomial $P_n(\cos\theta)$, this can be easily achieved by

```
Angular.legendre([0] * n + [1])
```

class abel.tools.polynomial.**ApproxGaussian**(*tol=0.0048*)

Bases: `object`

Piecewise quadratic approximation (non-negative and continuous but not exactly smooth) of the unit-amplitude, unit-SD Gaussian function $\exp(-r^2/2)$, equal to it at endpoints and midpoint of each piece. The forward Abel transform of this approximation will typically have a better relative accuracy than the approximation itself.

Parameters

tol (*float*) – absolute approximation tolerance (maximal deviation). Some reference values yielding the best accuracy for certain number of segments:

tol	Better than	Segments
3.7e-2	5%	3
1.4e-2	2%	5
4.8e-3	0.5%	7 (default)
0.86e-3	0.1%	13
0.99e-4	0.01%	27
0.95e-5	0.001%	59

ranges

list of parameters `(r_min, r_max, [c₀, c₁, c₂], r_0, s)` that can be passed directly to *PiecewisePolynomial* or, after "multiplication" by *Angular*, to *PiecewiseSPolynomial*

Type

lists of tuple

norm

the integral $\int_{-\infty}^{+\infty} f(r)\,dr$ for normalization (equals $\sqrt{2\pi}$ for the ideal Gaussian function, but slightly differs for the approximation)

Type

float

scaled(*A=1.0*, *r_0=0.0*, *sigma=1.0*)

Parameters for piecewise polynomials corresponding to the shifted and scaled Gaussian function $A\exp\left([(r-r_0)/\sigma]^2/2\right)$.

(Useful numbers: a Gaussian normalized to unit integral, that is the standard normal distribution, has $A = 1/\sqrt{2\pi}$; however, see *norm* above. A Gaussian with FWHM = 1 has $\sigma = 1/\sqrt{8\ln 2}$.)

Parameters

- **A** (*float*) – amplitude
- **r_0** (*float*) – peak position
- **sigma** (*float*) – standard deviation

Returns

ranges – parameters for the piecewise polynomial approximating the shifted and scaled Gaussian

Return type

list of tuple

`abel.tools.polynomial.`**`bspline`**(*spl*)

Convert SciPy B-spline to *PiecewisePolynomial* parameters.

Parameters

spl (*tuple or BSpline or UnivariateSpline*) – `scipy.interpolate` B-spline representation, such as `splrep()` results, `BSpline` object (result of `make_interp_spline()`, for example) or `UnivariateSpline` object

Returns

ranges – list of parameters `(r_min, r_max, coeffs, r_0)` that can be passed directly to *PiecewisePolynomial* or, after “multiplication” by *Angular*, to *PiecewiseSPolynomial*

Return type

list of tuple

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 $r_\text{min} \leqslant r \leqslant r_\text{max}$. 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. For example, here are explicit formulas for several low degrees:

k	$\int r^k \, dy$
0	y
1	$\frac{1}{2} r y + \frac{1}{2} x^2 \ln(y + r)$
2	$\left(\frac{1}{3} r^2 + \frac{2}{3} x^2\right) y$
3	$\left(\frac{1}{4} r^3 + \frac{3}{8} r x^2\right) y + \frac{3}{8} x^4 \ln(y + r)$
4	$\left(\frac{1}{5} r^4 + \frac{4}{15} r^2 x^2 + \frac{8}{15} x^4\right) y$
5	$\left(\frac{1}{6} r^5 + \frac{5}{24} r^3 x^2 + \frac{5}{16} r x^4\right) y + \frac{5}{16} x^6 \ln(y + r)$
…	…

The sums over m 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 ($\gtrsim$10 000) 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

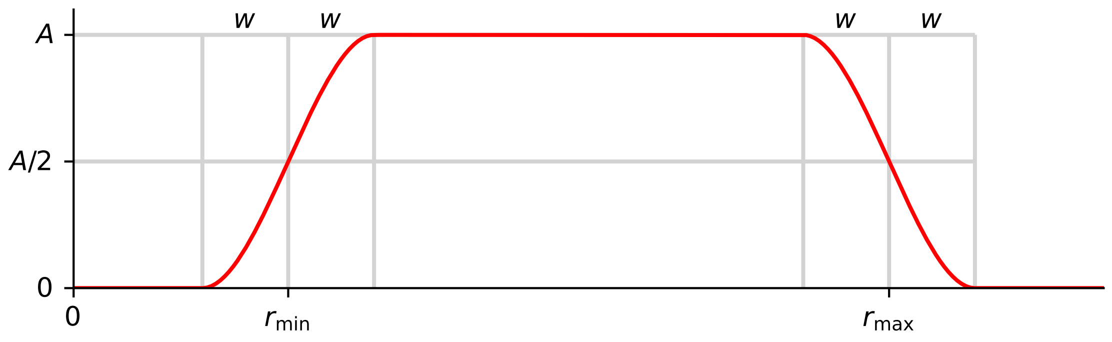
$$\mathbf{c}' = \mathrm{M}\mathbf{c},$$

where the coefficients are represented by a column vector $\mathbf{c} = (c_0, c_1, \dots, c_K)^\mathrm{T}$, and the matrix 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:

```python
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:

```python
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)], ...)
```

3.6.2 In spherical coordinates

Implemented as `SPolynomial`.

Axially symmetric bivariate polynomials in spherical coordinates have the general form

$$\operatorname{func}(\rho, \theta') = \sum_{m,n=0}^{M,N} c_{mn} \rho^m \cos^n \theta'$$

(see *rBasex: mathematical details* for definitions of the coordinate systems).

Abel transform

The forward Abel transform of this function defined on a radial domain $[\rho_\text{min}, \rho_\text{max}]$ (and zero elsewhere) is calculated as

$$\operatorname{abel}(r, \theta) = \sum_{m,n=0}^{M,N} c_{mn} \int \rho^m \cos^n \theta' \, dz,$$

where

$$\rho = \sqrt{r^2 + z^2}, \quad \cos\theta' = \frac{r}{\rho} \cos\theta,$$

and the Abel integral is taken over the domain where $\rho_\text{min} \leqslant \rho \leqslant \rho_\text{max}$. That is, for each term we have

$$\int \rho^m \cos^n \theta' \, dz = 2 \int\limits_{z_\text{min}}^{z_\text{max}} \rho^m \left(\frac{r}{\rho}\right)^n dz \cdot \cos^n \theta = 2r^m \int\limits_{z_\text{min}}^{z_\text{max}} \left(\frac{r}{\rho}\right)^{(n-m)} dz \cdot \cos^n \theta,$$

where

$$z_\text{min,max} = \begin{cases} \sqrt{\rho_\text{min,max}^2 - r^2}, & r < \rho_\text{min,max}, \\ 0 & \text{otherwise.} \end{cases}$$

The antiderivatives

$$F_{n-m}(r, z) = \int \left(\frac{r}{\rho}\right)^{n-m} dz$$

are given in *rBasex: mathematical details*, with the only difference that besides the recurrence relation

$$F_{k+2}(r, z) = \frac{1}{k} \left[z \left(\frac{r}{\rho}\right)^k + (k-1) F_k(r, z) \right]$$

for calculating the terms with positive $k = n - m$, the reverse recurrence relation

$$F_k(r, z) = \frac{1}{1-k} \left[z \left(\frac{r}{\rho}\right)^k - k F_{k+2}(r, z) \right]$$

is also used for negative k, requred for the terms with $m > n$.

The overall Abel transform thus has the form

$$\operatorname{abel}(r, \theta) = \sum_{m,n=0}^{M,N} c_{mn} \, 2r^m [F_{n-m}(r, z_\text{max}) - F_{n-m}(r, z_\text{min})] \cos^n \theta$$

and is calculated using Horner's method in r and $\cos\theta$ after precomputing the $F_{n-m}(r, z_{\text{min,max}})$ pairs for each needed value of $n-m$ (there are at most $M+N+1$ of them, if all $M \times N$ coefficients $c_{mn} \neq 0$).

Notice that these calculations are relatively expensive, since they are done for all pixels with $\rho \leqslant \rho_{\text{max}}$, for each of them involving summation and multiplication of up to $2MN$ terms in the above expression and evaluating transcendent functions present in $F_k(r, z)$. Moreover, the numerical problems for high-degree polynomials thus can be even more severe than for *univariate polynomials*.

3.6.3 Approximate Gaussian

Implemented as `ApproxGaussian`.

The Gaussian function

$$A \exp\left(-\frac{(r-r_0)^2}{2\sigma^2}\right)$$

is useful for representing peaks in simulated data but does not have an analytical Abel transform unless $r_0 = 0$. However, it can be approximated by piecewise polynomials to any accuracy, and these polynomials can be Able-transformed analytically, as shown above, thus providing an arbitrarily accurate approximation to the Abel transform of the initial Gaussian function.

In practice, it is sufficient to find the approximating piecewise polynomial for the Gaussian function $g(r) = \exp(-r^2/2)$ with unit amplitude and unit standard deviation, and the polynomial coefficients can then be scaled as described *above* to account for any A, r_0 and σ.

The goal is therefore to find $f(r)$ such that $|f(r) - g(r)| \leqslant \varepsilon$ for a given tolerance ε. The approximation implemented here uses a piecewise quadratic polynomial:

$$f(r) = \begin{cases} f_n(r) = c_{0,n} + c_{1,n}r + c_{2,n}r^2, & r \in [R_n, R_{n+1}], \\ 0, & r \notin [R_0, R_N], \end{cases}$$

where the domain is split into N intervals $[R_n, R_{n+1}]$, $n = 0, \ldots, N-1$. The strategy used for minimizing the number of intervals is to find the splitting points such that

$$f_n(r) = g(r) \quad \text{for} \quad r = R_n, R_{n+\frac{1}{2}}, R_{n+1}, \text{ where } R_{n+\frac{1}{2}} \equiv \frac{R_n + R_{n+1}}{2},$$

$$\max_{r\in[R_n, R_{n+1}]} \bigl|f_n(r) - g(r)\bigr| = \varepsilon,$$

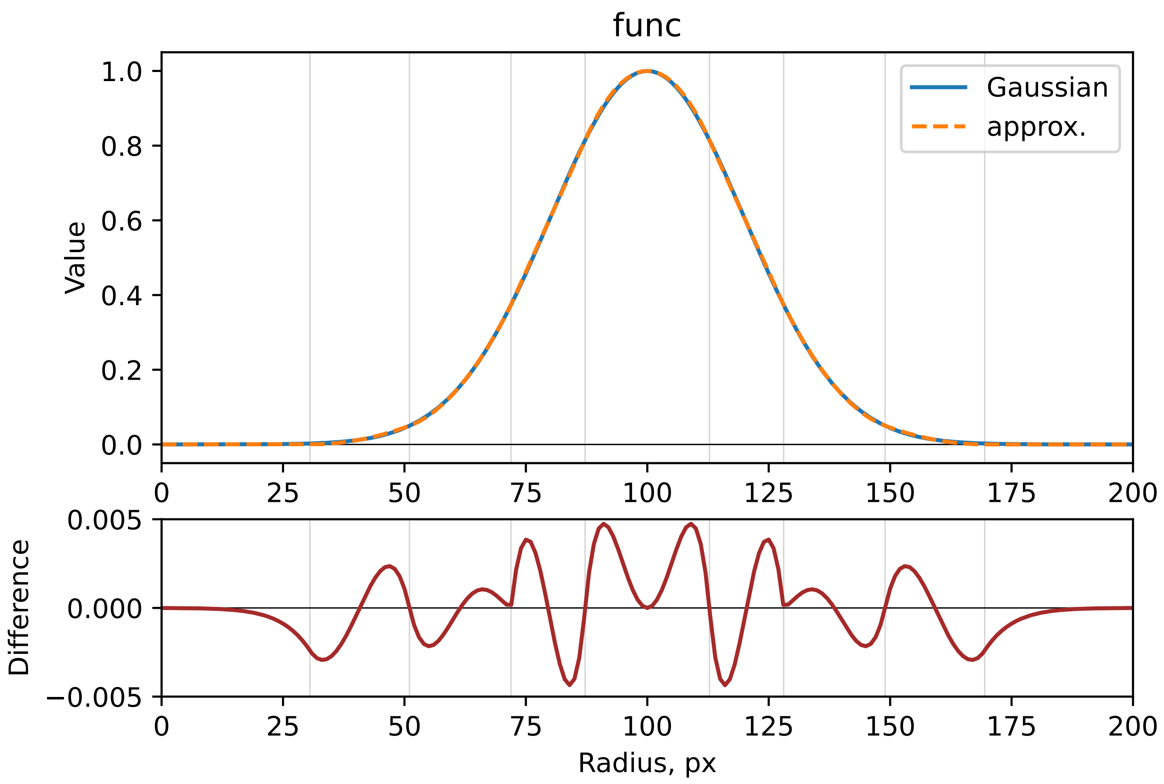
in other words, each parabolic segment matches the $g(r)$ values at the endpoints and midpoint of its interval, and its maximal deviation from $g(r)$ equals ε. The process starts from $R_0 = \sqrt{-2\ln(\varepsilon/2)}$, such that $g(R_0) = \varepsilon/2$, but setting $f_0(R_0) = 0$ for continuity. Then subsequent points $R_1, R_2, \ldots$ are found by solving $\max_{r\in[R_n, R_{n+1}]} \bigl|f_n(r) - g(r)\bigr| \approx \varepsilon$ for R_{n+1} numerically, using the following approximation obtained from the 3rd-order term of the $g(r)$ Taylor series (by construction, $f_n(r)$ reproduces the lower-order terms exactly, and the magnitudes of higher-order terms are much smaller):

$$\begin{aligned} &\max_{r\in[R_n, R_{n+1}]} \bigl|f_n(r) - g(r)\bigr| \approx \\ &\approx \max_{r=R_n, R_{n+\frac{1}{2}}, R_{n+1}} \left|\frac{g'''(r)}{3!}\right| \cdot \max_{r\in[R_n, R_{n+1}]} \bigl|(r-R_n)(r-R_{n+\frac{1}{2}})(r-R_{n+1})\bigr| = \\ &= \max_{r=R_n, R_{n+\frac{1}{2}}, R_{n+1}} \bigl|(3-r^2)rg(r)\bigr| \cdot \frac{|R_n - R_{n+1}|^3}{72\sqrt{3}}. \end{aligned}$$

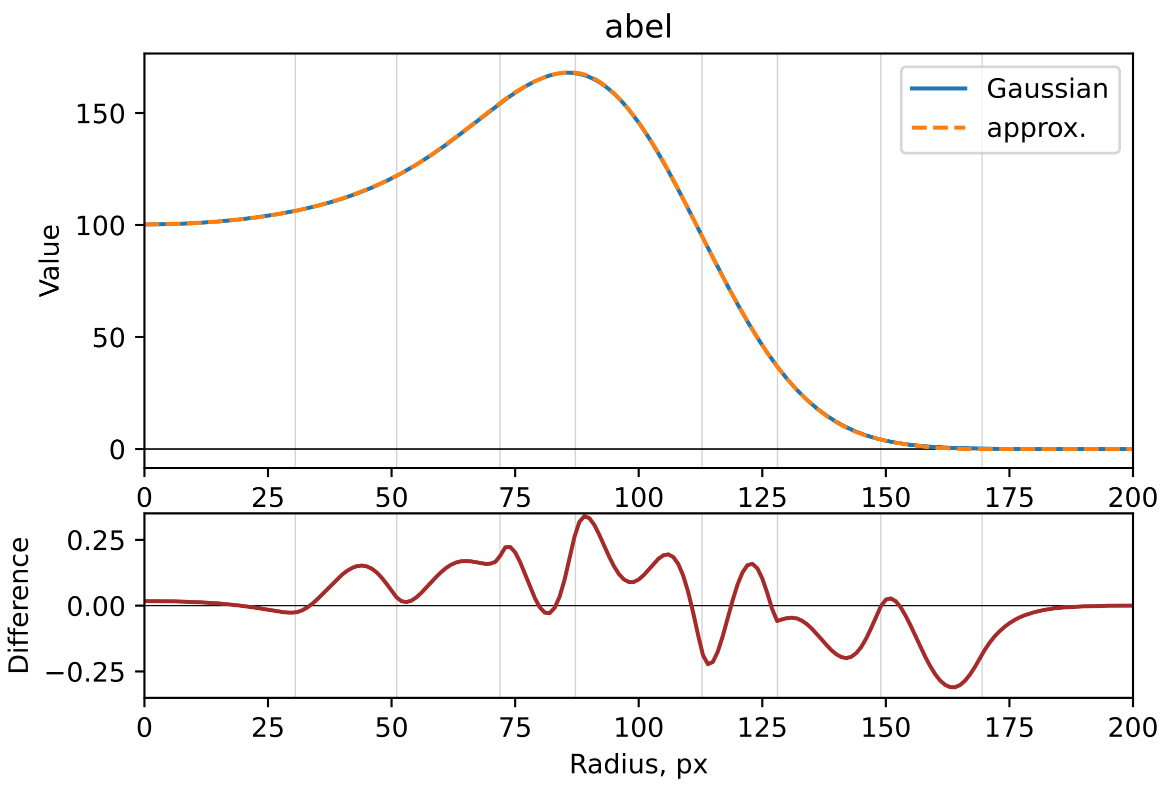
This process is repeated until $r = 0$ is reached, after which the found splitting is symmetrically extended to $-R_0 \leqslant r < 0$, and the polynomial coefficients for each segment are trivially calculated from the equations $f_n(r) = g(r)$ for $r = R_n, R_{n+\frac{1}{2}}, R_{n+1}$.

As an example, here is the outcome for the default approximation accuracy $\lesssim$0.5 %, resulting in just 7 segments:

```python
from abel.tools.polynomial import ApproxGaussian, PiecewisePolynomial
r = np.arange(201)
r0 = 100
sigma = 20
# actual Gaussian function
gauss = np.exp(-((r - r0) / sigma)**2 / 2)
# approximation with default tolerance (~0.5%)
approx = PiecewisePolynomial(r, ApproxGaussian().scaled(1, r0, sigma))
```



The Abel transform of this approximation is even more accurate, having the maximal relative deviation ~0.35/170 ≈ 0.2 %:



A practical example of using *ApproxGaussian* with *PiecewiseSPolynomial* can be found in the *SampleImage* source code.

3.7 abel.tools.transform_pairs module

Analytical function Abel-transform pairs

profiles 1–7:
G. C.-Y. Chan, Gary M. Hieftje, "Estimation of confidence intervals for radial emissivity and optimization of data treatment techniques in Abel inversion", Spectrochimica Acta B 61, 31–41 (2006), Table 1.

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

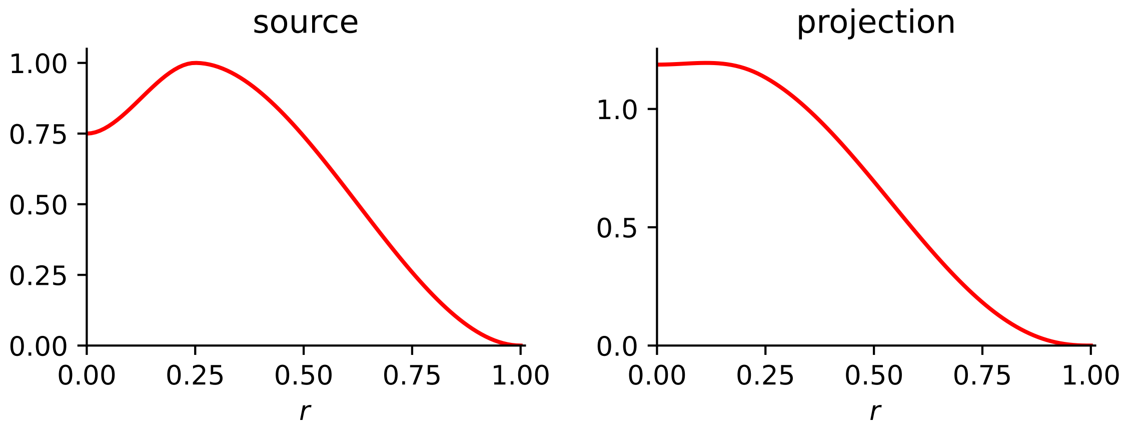
`abel.tools.transform_pairs.`**`a`**`(`n, r`)`

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

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

profile1: C. J. Cremers, R. C. Birkebak, "Application of the Abel Integral Equation to Spectrographic Data", Appl. Opt. 5, 1057–1064 (1966), Eq. (13).

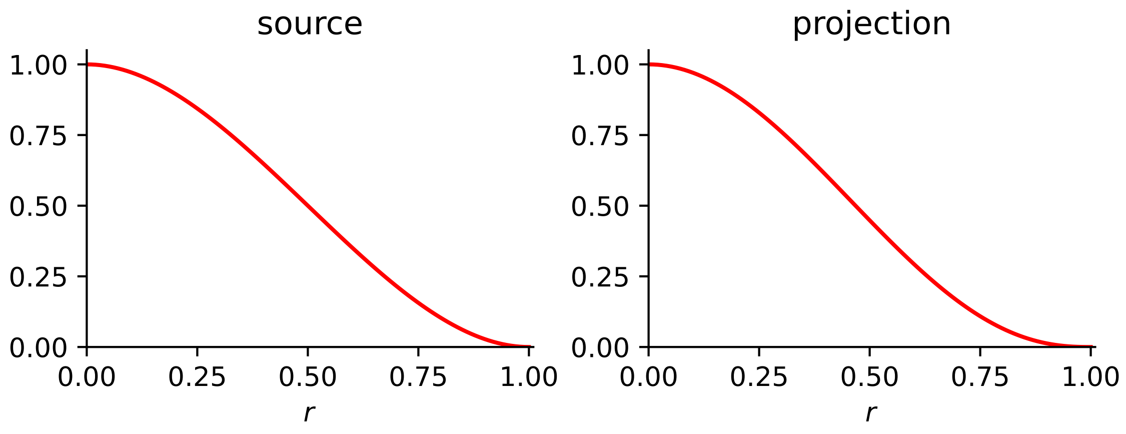
$$\begin{aligned}
\epsilon(r) &= 0.75 + 12r^2 - 32r^3 && 0 \le r \le 0.25 \\
\epsilon(r) &= \frac{16}{27}(1 + 6r - 15r^2 + 8r^3) && 0.25 < r \le 1 \\
I(r) &= \frac{1}{108}(128a_1 + a_{0.25}) + \frac{2}{27}r^2(283a_{0.25} - 112a_1) + \\
&\quad \frac{8}{9}r^2\left[4(1+r^2)\ln\frac{1+a_1}{r} - (4+31r^2)\ln\frac{0.25+a_{0.25}}{r}\right]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



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

profile2: C. J. Cremers, R. C. Birkebak, "Application of the Abel Integral Equation to Spectrographic Data", Appl. Opt. 5, 1057–1064 (1966), Eq. (11).

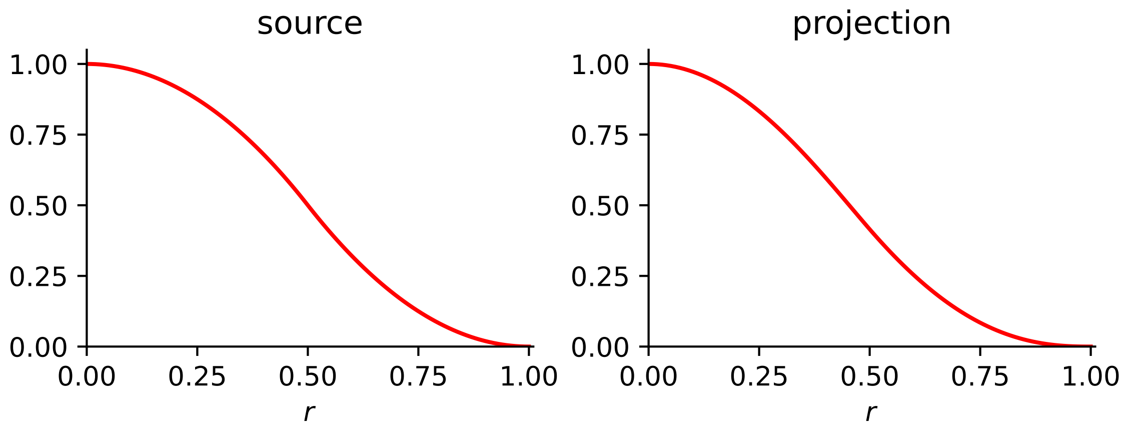
$$\begin{aligned}
\epsilon(r) &= 1 - 3r^2 + 2r^3 && 0 \le r \le 1 \\
I(r) &= a_1\left(1 - \frac{5}{2}r^2\right) + \frac{3}{2}r^4 \ln\frac{1+a_1}{r} && 0 \le r \le 1
\end{aligned}$$



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

profile3: C. J. Cremers, R. C. Birkebak, "Application of the Abel Integral Equation to Spectrographic Data", Appl. Opt. 5, 1057–1064 (1966), Eq. (12).

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



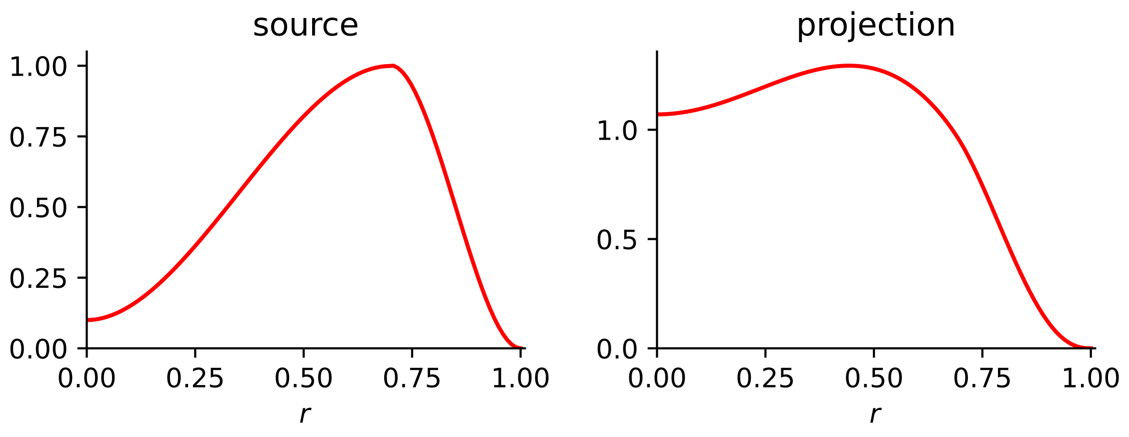
abel.tools.transform_pairs.**profile4**(*r*)

profile4: R. Álvarez, A. Rodero, M. C. Quintero, "An Abel inversion method for radially resolved measurements in the axial injection torch", Spectochim. Acta B 57, 1665–1680 (2002), Eq. (10).

Note: This profile has a small discontinuity at $r = 0.7$.

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

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

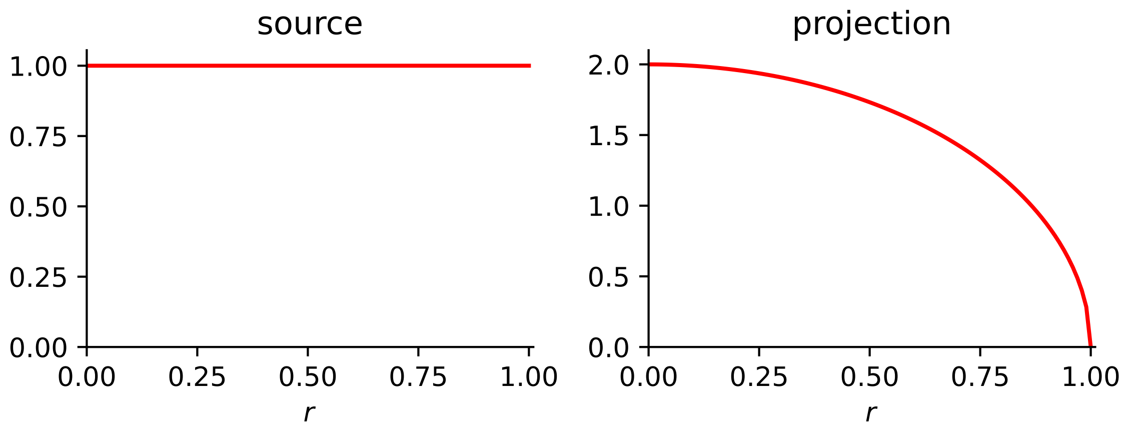


`abel.tools.transform_pairs.`**`profile5`**`(`r`)`

profile5: M. J. Buie, J. T. P. Pender, J. P. Holloway, T. Vincent, P. L. G. Ventzek, M. L. Brake, J. Quant. Spectrosc. Radiat. Transf. 55, 231–243 (1996), Table 1, № 1.

Note: This profile is discontinuous (and its projection is not smooth) at $r = 1$, which can cause different problems in different methods, in particular, depending on their assumptions where the singularity is located within the last pixel.

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

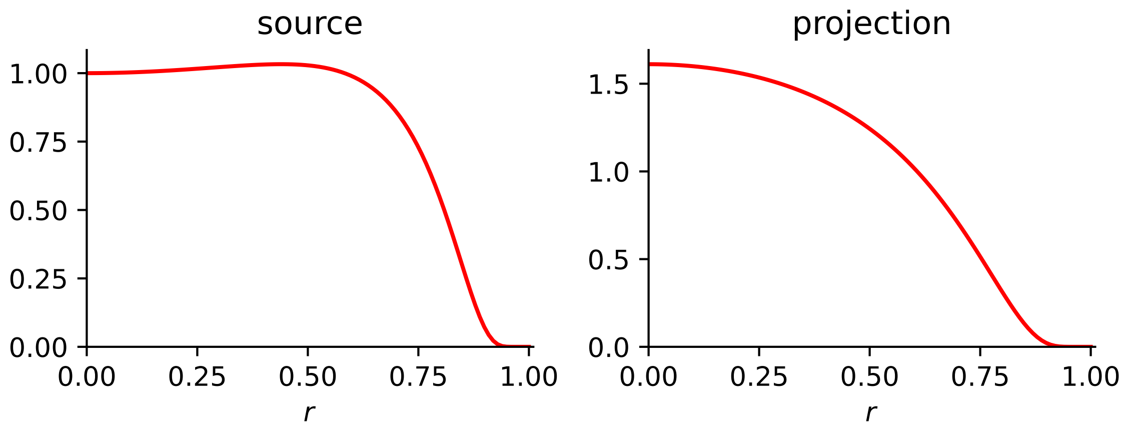


abel.tools.transform_pairs.**profile6**(*r*)

profile6: M. J. Buie, J. T. P. Pender, J. P. Holloway, T. Vincent, P. L. G. Ventzek, M. L. Brake, J. Quant. Spectrosc. Radiat. Transf. 55, 231–243 (1996), Table 1, № 7.

$$\epsilon(r) = (1-r^2)^{-\frac{3}{2}} \exp\left[1.1^2\left(1-\frac{1}{1-r^2}\right)\right] \quad 0 \le r \le 1$$

$$I(r) = \frac{\sqrt{\pi}}{1.1a_1} \exp\left[1.1^2\left(1-\frac{1}{1-r^2}\right)\right] \quad 0 \le r \le 1$$

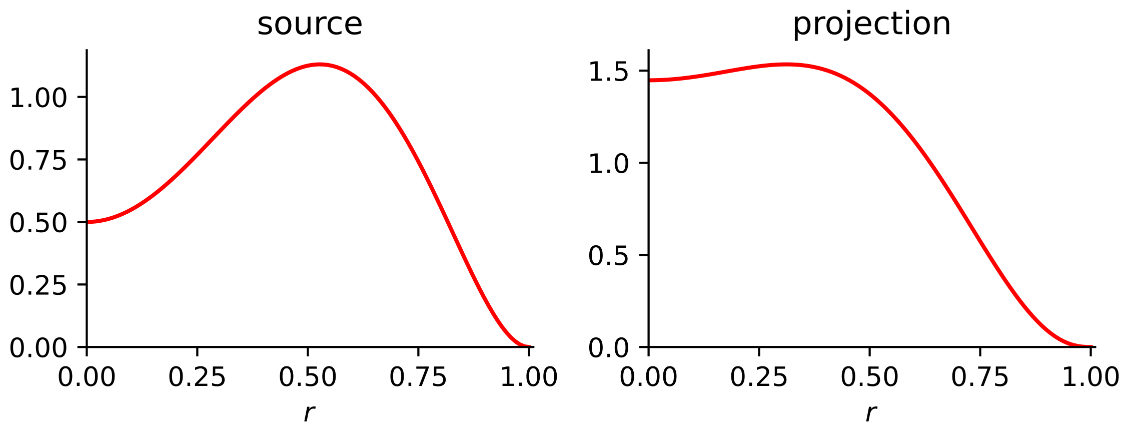


abel.tools.transform_pairs.**profile7**(*r*)

profile7: M. J. Buie, J. T. P. Pender, J. P. Holloway, T. Vincent, P. L. G. Ventzek, M. L. Brake, J. Quant. Spectrosc. Radiat. Transf. 55, 231–243 (1996), Table 1, № 9 (divided by 2).

$$\epsilon(r) = \frac{1}{2}(1 + 10r^2 - 23r^4 + 12r^6) \quad 0 \le r \le 1$$

$$I(r) = \frac{8}{105}a_1(19 + 34r^2 - 125r^4 + 72r^6) \quad 0 \le r \le 1$$



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='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.

  K. R. Overstreet, P. Zabawa, J. Tallant, A. Schwettmann, J. P. Shaffer, "Multiple scattering and the density distribution of a Cs MOT", Optics Express 13, 9672–9682 (2005).

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()` with `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---    ---o----   ---o---    ---o---
   Q2 | Q3    Q2 | Q2    Q1 | Q0    Q1 | Q1
```

Return type
np.array

3.9 abel.tools.vmi module

`abel.tools.vmi.`**`radial_intensity`**(*kind*, *IM*, *origin=None*, *dr=1*, *dt=None*)

Calculate the one-dimensional radial intensity profile by angular integration or averaging of the image, treated either as a two-dimensional distribution or as a central slice of a cylindrically symmetric three-dimensional distribution.

Parameters

- **kind** (*str*) – operation to perform:

  `'int2D'`:
  integration in 2D over polar angles

'int3D':
integration in 3D over solid angles

'avg2D':
averaging in 2D over polar angles

'avg3D':
averaging in 3D over solid angles

- **IM** (*2D numpy.array*) – the image data
- **origin** (*tuple of float or None*) – image origin in the (row, column) format. If `None`, the geometric center of the image (`rows // 2, cols // 2`) is used.
- **dr** (*float*) – radial grid spacing in pixels (default 1). `dr=0.5` may reduce pixel granularity of the radial profile.
- **dt** (*float or None*) – angular grid spacing in radians. If `None`, the number of theta values will be set to largest dimension (the height or the width) of the image, which should typically ensure good sampling.

Returns

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

`abel.tools.vmi.`**`angular_integration_2D`**(*IM*, *origin=None*, *dr=1*, *dt=None*)

Angular integration of the image as a two-dimensional object.

Equivalent to *`radial_intensity('int2D', IM, origin, dr, dt)`*.

`abel.tools.vmi.`**`angular_integration_3D`**(*IM*, *origin=None*, *dr=1*, *dt=None*)

Angular integration of the three-dimensional cylindrically symmetric object represented by the image as its central slice. When applied to the inverse Abel transform of a velocity-mapping image, this yields the speed distribution.

Equivalent to *`radial_intensity('int3D', IM, origin, dr, dt)`*.

`abel.tools.vmi.`**`average_radial_intensity_2D`**(*IM*, *origin=None*, *dr=1*, *dt=None*)

Calculate the average radial intensity of the image as a two-dimensional object.

Equivalent to *`radial_intensity('avg2D', IM, origin, dr, dt)`*.

`abel.tools.vmi.`**`average_radial_intensity_3D`**(*IM*, *origin=None*, *dr=1*, *dt=None*)

Calculate the average radial intensity of the three-dimensional cylindrically symmetric object represented by the image as its central slice.

Equivalent to *`radial_intensity('avg3D', IM, origin, dr, dt)`*.

`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.

Warning: This function behaves incorrectly: misses a factor of π for 3D integration, with `Jacobian=True`, and for `Jacobian=False` returns the *average* (over polar angles) multiplied by 2π instead of integrating. It is currently deprecated and is provided only for backward compatibility, but will be removed in the future.

Please use *radial_intensity()*, *angular_integration_2D()* or *angular_integration_3D()*.

Parameters

- **IM** (*2D numpy.array*) – the image data
- **origin** (*tuple or None*) – image origin in the (row, column) format. If `None`, the geometric center of the image (`rows // 2, cols // 2`) is used.
- **Jacobian** (*bool*) – 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 grid spacing in pixels (default 1). `dr=0.5` may reduce pixel granularity of the speed profile.
- **dt** (*float or None*) – angular grid spacing in radians. If `None`, the number of theta values will be set to largest dimension (the height or the width) 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 2π.

Warning: This function is currently deprecated and is provided only for backward compatibility, but will be removed in the future.

Please use *radial_intensity()*, *average_radial_intensity_2D()* or *average_radial_intensity_3D()*.

Parameters

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

Returns

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

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

Intensity variation in the angular coordinate.

This function is the θ-coordinate complement to *abel.tools.vmi.average_radial_intensity_3D()*.

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

See *examples/example_O2_PES_PAD.py*.

Parameters

- **IM** (*2D numpy.array*) – the image data
- **origin** (*tuple or None*) – image origin in the (row, column) format. If `None`, the geometric center of the image (`rows // 2, cols // 2`) is used.
- **radial_ranges** (*list of tuple or int*) –

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

  int
  radial step. Evaluates the intensity vs angle for the whole radial range `(0, step), (step, 2*step), ..`

Returns

- **Beta** (*list of tuples*) – (beta0, error_beta_fit0), (beta1, error_beta_fit1), … corresponding to the radial ranges
- **Amplitude** (*list of tuples*) – (amp0, error_amp_fit0), (amp1, error_amp_fit1), … corresponding to the radial ranges
- **Rmidpt** (*list of float*) – radial mid-point of each radial range
- **Intensity_vs_theta** (*list of 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 β, for I vs θ 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.

J. Cooper, R. N. 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 or None*) – angular ranges over which to fit `[(theta1, theta2), (theta3, theta4)]`. Allows data to be excluded from fit; default (`None`) is to 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 $O_2{}^-$ photodetachment, the origin band occurs at the electron affinity $EA = 3613$ 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 1.204×10^{-5} cm^{-1}/pixel^2 applies for "examples/data/O2-ANU1024.txt" (with Vrep = -2200 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".
- **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. `1.0e7/454.5` for "examples/data/O2-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. `-2200` (volts), for "examples/data/O2-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='center'*, *rmax='MIN'*, *order=2*, *odd=False*, *use_sin=True*, *weights=None*, *method='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, ≥ 0 (by default, 2). Requesting very high orders ($\gtrsim 15$) can result in excessive noise, especially at small radii and for narrow peaks.
- **odd** (*bool*) – include odd angular orders. By default is `False`, but is enabled automatically if **order** is odd. Notice that although odd orders can be extracted from the upper or lower image part alone, analyzing the whole image is more reliable.
- **use_sin** (*bool*) – use $|\sin\theta|$ weighting (enabled by default). 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*, *order*, *odd*, *valid=None*)

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. Odd angular terms are included only when they are used (**odd** = `True` or **order** is odd), otherwise there are only 1 + **order**/2 rows. The terms can be easily separated like `I, beta2, beta4 = res.Ibeta()`. Python 3 users can also collect all β parameters as `I, *beta = res.Ibeta()` for any **order**. Alternatively, transposing the results as `Ibeta = res.Ibeta().T` allows accessing all terms $\big(I(r), \beta_2(r), \beta_4(r), \ldots\big)$ at particular radius *r* as `Ibeta[r]`.

r

radii from 0 to **rmax**

Type
numpy array

order

highest order in the angular distributions

Type
int

odd

whether odd angular orders are present

Type
bool

orders

orders for all angular terms:

[0, 2, …, **order**] for **odd** = `False`,

[0, 1, 2, …, **order**] for **odd** = `True`

Type
list of int

`sinpowers`

sine powers m in the $\cos^n\theta \cdot \sin^m\theta$ terms from `cossin()`; cosine powers n are given by `orders` (see above)

Type
list of int

`valid`

flags for each radius indicating whether it has valid data (radii that have zero weights for all pixels will have no valid data)

Type
bool array

`cos()`

Radial distributions of $\cos^n\theta$ terms ($0 \le n \le$ **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**, and $n + m$ = **order** $- 1$ for odd orders, with m always even).

For **order** = 0:
$\cos^0\theta$ is the total intensity.
For **order** = 1:
$\cos^0\theta$ is the total intensity,

$\cos^1\theta$ is the antisymmetric component.
For **order** = 2
$\sin^2\theta$ corresponds to "perpendicular" ($\perp$) transitions,

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

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

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

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

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

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

and so forth.

Returns
cosnsinm – radial dependences of the $\cos^n\theta \cdot \sin^m\theta$ terms, ordered from lower to higher $\cos\theta$ powers

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 $n \neq m$; 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 (including even and odd terms)

$$I(r,\theta,\varphi)\,d\Omega = \frac{1}{4\pi}I(r)\big[1 + \beta_1(r)P_1(\cos\theta) + \beta_2(r)P_2(\cos\theta) + \dots\big],$$

or, for distributions with top–bottom symmetry (only even terms),

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

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 β. Since anisotropy parameters are non-linear, the central moving average is applied to the harmonics (which are linear), and then β is calculated from them. In case of well separated peaks, setting **window** to the peak width will result in β 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='cc'*, *rmax='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='cc'*, *rmax='MIN'*, *order=2*, ***kwargs*)

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

`abel.tools.vmi.`**Ibeta**(*IM*, *origin='cc'*, *rmax='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='cc'*, *rmax='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='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='half'*, *rmax='MIN'*, *order=2*, *weight=['none', 'sin', 'sin+array']*, *method='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_∞) with t_1 corresponding to single-image (non-cached) performance, and t_∞ 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

Return type
a binary array with the symmetry condition for the corresponding quadrants

Notes

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

See issue #34 comment for the defintion of a center of the image.

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

4.1 Comparison of Abel Transform Methods

4.1.1 Abstract

This document provides a comparison of the quality and efficiency of the various Abel transform methods that are implemented in PyAbel. Some of the information presented here is adapted from [hickstein2019].

4.1.2 Introduction

The projection of a three-dimensional (3D) object onto a two-dimensional (2D) surface takes place in many measurement processes; a simple example is the recording of an X-ray image of a soup bowl, donut, egg, wineglass, or other cylindrically symmetric object Fig. 4.1, where the axis of cylindrical symmetry is parallel to the plane of the detector. Such a projection is an example of a *forward* Abel transform and occurs in numerous experiments, including photoelectron/photoion spectroscopy ([dribinski2002], [bordas1996], [chandler1987]) the studies of plasma plumes ([glasser1978]), flames ([deiluliis1998], [cignoli2001], [snelling1999], [das2017]), and solar occulation of planetary atmospheres ([gladstone2016], [lumpe2007]). The analysis of data from these experiments requires the use of the *inverse* Abel transform to recover the 3D object from its 2D projection.

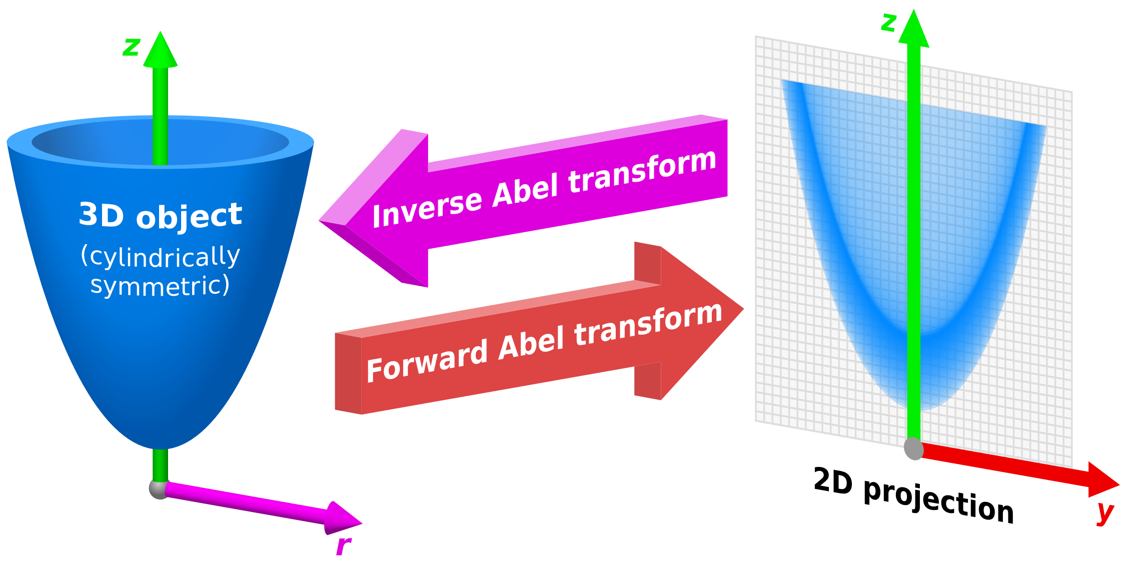


Fig. 4.1: The forward Abel transform maps a cylindrically symmetric three-dimensional (3D) object to its two-dimensional (2D) projection, a physical process that occurs in many experimental situations. For example, an X-ray image of the object on the left would produce the projection shown on the right. The *inverse* Abel transform takes the 2D projection and mathematically reconstructs the 3D object. As indicated by the Abel transform equations (below), the 3D object is described in terms of $(r,\ z)$ coordinates, while the 2D projection is recorded in $(y,\ z)$ coordinates.

While the forward and inverse Abel transforms may be written as simple, analytical expressions, attempts to naively evaluate them numerically for experimental images does not yield reliable results [whitaker2003]. Consequently, many numerical methods have been developed to provide approximate solutions to the Abel transform (for example: [dribinski2002], [bordas1996], [chandler1987], [dasch1992], [rallis2014], [gerber2013], [harrison2018], [demicheli2017], [dick2014]). Each method was created with specific goals in mind, with some taking advantage of pre-existing knowledge about the shape of the object, some prioritizing robustness to noise, and others offering enhanced computational efficiency. Each algorithm was originally implemented with somewhat different mathematical conventions and with often conflicting requirements for the size and format of the input data. Fortunately, PyAbel provides a consistent interface for the Abel-transform methods via the Python programming language, which allows for a straightforward, quantitative comparison of the output.

The following sections present several comparisons of the quality and speed of the various Abel-transform algorithms presented in PyAbel. In general, all of the methods provide reasonably quality results, with some methods providing options for additional smoothing of the data. However, some methods are orders-of-magnitude more efficient than others.

4.1.3 Math

The **forward Abel transform** is given by

$$F(y,z) = 2\int_y^\infty \frac{f(r,z)\,r}{\sqrt{r^2-y^2}}\,dr,$$

where y, r, and z are the spatial coordinates as shown in Fig. 4.1, $f(r,\ z)$ is the density of the 3D object at $(r,\ z)$, and $F(y,\ z)$ is the intensity of the projection in the 2D plane.

The **inverse Abel transform** is given by

$$f(r,z) = -\frac{1}{\pi}\int_r^\infty \frac{dF(y,z)}{dy}\,\frac{1}{\sqrt{y^2-r^2}}\,dy.$$

While the transform equations can be evaluated analytically for some mathematical functions, experiments typically generate discrete data (e.g., images collected with a digital camera), which must be evaluated numerically. Several issues arise when attempting to evaluate the Abel transform numerically. First, the simplest computational interpretation of inverse Abel transform equation involves three loops: over z, r, and y, respectively. Such nested loops can be computationally expensive. Additionally, $y = r$ presents a singularity where the denominator goes to zero and the integrand goes to infinity. Finally, a simple approach requires a large number of sampling points in order to provide an accurate transform. Indeed, a simple numerical integration of the above equations has been shown to provide unreliable results [whitaker2003].

Various algorithms have been developed to address these issues. PyAbel incorporates numerous algorithms for the inverse Abel transform, and some of these algorithms also support the forward Abel transform. The following comparisons focus on the results of the inverse Abel transform, because it is the inverse Abel transform that is used most frequently to interpret experimental data.

Note that the forward and inverse Abel transforms are defined on the whole space, with infinite integration limits, but in reality, experimental data are limited to finite ranges of r or y. Thus the intensity distributions f and F must be zero outside these ranges, otherwise the transforms cannot be performed correctly. In other words, only localized objects can be transformed, and the object must be contained entirely within the image frame. If the image has any background, it must be subtracted before applying the transform, such that the image intensity goes to zero near the edge (however, the *Direct* and *Hansen–Law* methods effectively disregard a constant background).

4.1.4 List of Abel-Transform Methods in PyAbel

Below is a list that describes the basic approach and characteristics of all the Abel-transform algorithms implemented in PyAbel. The title of each algorithm is the keyword that can be passed to the `method` argument in *`abel.transform.Transform()`*. Algorithms that pre-compute matrices for a specific image size, and (optionally) save them to disk for subsequent reuse, are indicated with the letter S. All methods implement the inverse Abel transform, while methods that also implement a forward transform are indicated with an F.

- `basex` (F, S) – The "BAsis Set EXpansion" (BASEX) method of Dribinski and co-workers [dribinski2002] uses a basis set of Gaussian-like functions. This is one of the *de facto* standard methods in photoelectron/photoion spectroscopy [whitaker2003] and is highly recommended for general-purpose Abel transforms. The number of basis functions and their width can be varied. However, following the basis set provided with the original BASEX.exe program, by default the `basex` algorithm use a basis set where the full width at $1/e^2$ of the maximum is equal to 2 pixels and the basis functions are located at each pixel. Thus, the resolution of the image is roughly maintained. The `basex` algorithms allows a "Tikhonov regularization" to be applied, which suppresses intensity oscillations, producing a less noisy image. In the experimental comparisons presented below, the Tikhonov regularization factor is set to 200, which provides reasonable suppression of noise while still preserving the fine features in the image. See *BASEX* and *`abel.basex.basex_transform()`*.
- `onion_peeling` (S) – This method, and the following two methods (`three_point`, `two_point`), are adapted from the 1992 paper by Dasch [dasch1992]. All of these methods reduce the core Abel transform to a simple matrix-algebra operation, which allows a computationally efficient transform. Dasch emphasizes that these techniques work best in cases where the difference between adjacent points is much greater than the noise in the projections (i.e., where the raw data is not oversampled). This "onion-peeling deconvolution" method is one of the simpler and faster inverse Abel-transform methods. See *Onion Peeling (Dasch)* and *`abel.dasch.onion_peeling_transform()`*.
- `three_point` (S) – This "three-point" algorithm [dasch1992] provides slightly more smoothing than the similar `two_point` or `onion_peeling` methods. The name refers to the fact that three neighboring pixels are considered, which improves the accuracy of the method for transforming smooth functions, as well as reducing the noise in the transformed image. The trade-off is that the ability of the method to transform very sharp (single-pixel) features is reduced. This is an excellent general-purpose algorithm for the Abel transform. See *Three Point* and *`abel.dasch.three_point_transform()`*.

- `two_point` (S) – The "two-point method" (also described by Dasch [dasch1992]) is a simplified version of the `three_point` algorithm and provides similar transform speeds. Since it only considers two adjacent points in the function, it allows sharper features to be transformed than the `three_point` method, but does not offer as much noise suppression. This method is also appropriate for most Abel transforms. See *Two Point (Dasch)* and *`abel.dasch.two_point_transform()`*.
- `direct` (F) – The "direct" algorithms [yurchak2015] uses a simple numerical integration, which closely resembles the basic Abel-transform equations (above). If the `direct` algorithm is used in its most naive form, the agreement with analytical solutions is poor, due to the singularity in the integral when $r = y$. However, a correction can be applied, where the function is assumed to be piecewise-linear across the pixel where this condition is met. This simple approximation allows a reasonably accurate transform to be completed. Fundamentally, the `direct` algorithm requires that the input function be finely sampled to achieve good results. PyAbel incorporates two implementations of the `direct` algorithm, which produce identical results, but with different calculation speeds. The `direct_Python` implementation is written in pure Python, for easy interpretation and modification. The `direct_C` implementation is written in Cython, a Python-like language that is converted to C and compiled, providing higher computational efficiency. This method is included mainly for educational and comparison purposes. In most cases, other methods will provide more reliable results and higher computational efficiency. See *Direct* and *`abel.direct.direct_transform()`*.
- `hansenlaw` (F) – The recursive method of Hansen and Law ([hansen1985], [hansen1985b], [gascooke2000]) interprets the Abel transform as a linear space-variant state-variable equation, to provide a reliable, computationally efficient transform. The `hansenlaw` method also provides an efficient forward Abel transform. It is recommended for most applications. See *Hansen–Law* and *`abel.hansenlaw.hansenlaw_transform()`*.
- `linbasex` (S) – The "lin-BASEX" method of Gerber et al. [gerber2013] models the 2D projection using spherical functions, which evolve slowly as a function of polar angle. Thus, it can offer a substantial increase in signal-to-noise ratio in many situations, but **it is only appropriate for transforming certain projections that are appropriately described by these basis functions**. This is the case for typical velocity-map-imaging photoelectron/photoion spectroscopy [chandler1987] experiments, for which the algorithm was designed. However, for example, it would not be appropriate for transforming the object shown in Fig. 4.1. The algorithm directly produces the coefficients of the involved spherical functions, which allows both the angular and radially integrated distributions to be produced analytically. This ability, combined with the strong noise-suppressing capability of using smooth basis functions, can aid the interpretation of photoelectron/photoion distributions. See *Lin-Basex* and *`abel.linbasex.linbasex_transform()`*.
- `onion_bordas` – The onion-peeling method of Bordas et al. [bordas1996] is a Python adaptation of the MATLAB implementation of Rallis et al. [rallis2014]. While it is conceptually similar to the `onion_peeling` method, the numerical implementation is significantly different. This method is reasonably slow, and is therefore not recommended for general use. See *Onion Peeling (Bordas)* and *`abel.onion_bordas.onion_bordas_transform()`*
- `rbasex` (F, S) – The rBasex method is based on the pBasex method of Garcia et al. [garcia2004], using basis functions developed by Ryazanov [ryazanov2012]. This method evaluates radial distributions of velocity-map images and transforms them to radial distributions of the reconstructed 3D distributions. Similar to `linbasex`, the `rbasex` method makes additional assumptions about the symmetry of the data is not applicable to all situations. See *rBasex* and *`abel.rbasex.rbasex_transform()`*.
- `daun` (F, S) – The method by Daun et al. [daun2006] applies Tikhonov regularization to onion-peeling deconvolution. It is conceptually similar to "BASEX" (`basex`), but instead of L_2 regularization uses the first-order difference operator (approximating the derivative operator) as the Tikhonov matrix to suppress high-frequency oscillations, making the transform less sensitive to perturbations in the projected data. The PyAbel implementation also includes several extensions to the original method. First, in addition to the rectangular basis functions implied in onion peeling, explicit basis sets of piecewise polynomials up to 3rd degree (cubic splines) can be chosen. Second, the L_2 regularization (as in BASEX) is implemented for comparison. And most importantly, the non-negative least-squares solution to the deconvolution problem can be obtained, which produces meaningful results in situations where the transformed intensities must not be negative, and at the same time greatly reduces the baseline noise. See *Daun* and *`abel.daun.daun_transform()`*.

4.1.5 Implementation

The *abel.transform.Transform()* class provides a uniform interface to all of the transform methods, as well as numerous related functions for centering and symmetrizing the input images. So, this interface can be used to quickly switch between transform methods to determine which method works best for a specific dataset.

Generating a sample image, performing a forward Abel transform, and completing an inverse Abel transform requires just a few lines of Python code:

```
import abel
im0 = abel.tools.analytical.SampleImage().func
im1 = abel.Transform(im0,
                     direction='forward',
                     method='hansenlaw').transform
im2 = abel.Transform(im1,
                     direction='inverse',
                     method='three_point').transform
```

Choosing a different method for the forward or inverse transform requires only that the `method` argument be changed. Additional arguments can be passed to the individual transform functions using the `transform_options` argument. A basic graphical user interface (GUI) for PyAbel is also available as example_GUI.py in the examples directory.

In addition to the transform methods themselves, PyAbel provides many of the pre-processing methods required to obtain optimal Abel transforms. For example, an accurate Abel transform requires that the center of the image is properly identified. Several approaches allow to perform this identification in PyAbel, including the center-of-mass, convolution, and Gaussian-fitting. Additionally, PyAbel incorporates a “circularization” method [gascooke2017], which allows the correction of images that contain features that are expected to be circular (such as photoelectron and photoion momentum distributions). Moreover, the `abel.tools` module contains a host of *post*-processing algorithms, which provide, for example, efficient projection into polar coordinates and radial or angular integration.

4.1.6 Conventions

The conventions for PyAbel are listed in the *Conventions* section of the *PyAbel README*.

In order to provide similar results, PyAbel ensures that the numerical conventions are consistent across the various transform methods. For example. when dealing with pixel data, an ambiguity arises: do intensity values of the pixels represent the value of the data at $r = \{0, 1, 2, ..., n-1\}$, where n is an integer, or do they correspond to $r = \{0.5, 1.5, 2.5, ..., n-0.5\}$? Either convention is reasonable, but comparing results from methods that adopt differing conventions can lead to small but significant shifts. PyAbel adopts the convention that the pixel values correspond to $r = \{0, 1, 2, ..., n-1\}$. One consequence of this is that, when considering an experimental image that contains both the left and right sides of the image, the total image width must be odd, such that $r = \{1-n, ..., -2, -1, 0, 1, 2, ..., n-1\}$. A potential disadvantage of our “odd image” convention is that 2D detectors typically have a grid of pixels with an *even* width (for example, a 512×512-pixel camera). If the image were perfectly centered on the detector, the odd-image convention would not match the data, and a half-pixel shift would be required. However, in nearly all real-world experiments, the image is not perfectly centered on the detector and a shift of *several* pixels is required, so the additional half-pixel shift is of no significance.

A similar ambiguity exists with regards to the left–right and top–bottom symmetry of the image. In principle, since the Abel transform assumes cylindrical symmetry, left–right symmetry should always exist, and it should only be necessary to record one side of the projection. However, many experiments record both sides of the projection. Additionally, many experiments record object that possess top–bottom symmetry. Thus, in some situations, it is best to average all of the image quadrants into a single quadrant and perform a single Abel transform on this quadrant. On the other hand, the quadrants may not be perfectly symmetric due to imperfections or noise in the experiment, and it may be best to perform the Abel transform on each quadrant separately and select the quadrant that produces the highest-quality data. PyAbel offers full flexibility, providing the ability to selectively enforce top–bottom and left–right symmetry, and to specify which quadrants are averaged. By default, each quadrant is processed separately and recombined into in composite image that does not assume either top–bottom or left–right symmetry. For more details, see `abel.transform.Transform()`.

In the following performance benchmarks, left–right symmetry is assumed, because this is the most common benchmark presented in other studies ([rallis2014], [harrison2018]). However, the image size is listed as the width of a square image. For example, $n = 513$ corresponds to the time for the transformation of a 513×513-pixel image with the axis of symmetry located in the center. Since the Abel transform makes the assumption of cylindrical symmetry, both sides of the image are identical, and it is sufficient to perform the Abel transform on only one side of the image, or on an average of the two sides. So, to complete an Abel transform of a typical 513×513-pixel image, it is only necessary to perform the Abel transform on a 513×257-pixel array.

Another fundamental question about real-world Abel transforms is whether negative values are allowed in the transform result. In most situations, negative values are not physical, and some implementations set all negative values to zero. In contrast, PyAbel allows negative values, which enables its use in situations where negative values are physically reasonable. Moreover, maintaining negative values keeps the transform methods linear and gives users the option to average, smooth, or fit images either before or after the Abel transform without causing a systematic error in the baseline. Suppression of negative values in a transformed image `im` can easily be achieved by executing `im[im<0] = 0`. On the other hand, the `daun` and `rbasex` methods offer optional non-linear regularization methods specifically designed to produce non-negative values without systematically shifting the baseline. It is recommended to use them instead of artificially zeroing negative values in situations where negative values are undesirable but the transform speed is not essential.

4.1.7 Comparison of Transform Results

Since PyAbel incorporates numerous Abel-transform methods into the same interface, it is straightforward to directly compare the results. Consequently, a good approach is to simply try several (or all!) of the transform methods and see which produces the best results or performance for a specific application. Nevertheless, the following provides a brief comparison of the various transform methods in several cases. First, the methods are applied to a simple Gaussian function (for which an analytical Abel transform exists) in order to assess the accuracy of each transform method. Second, each method is applied to a "comb" function constructed of narrow peaks with noise added in order to closely examine the fundamental resolution of each method and how noise accumulates. Third, each method is used to provide the inverse Abel transform a high-resolution photoelectron-spectroscopy image in order to examine the ability of each method to handle real-world data.

The Abel transform of a Gaussian is simply a Gaussian, which allows a comparison of each numerical transform method with the analytical result in the case of a one-dimensional (1D) Gaussian (Fig. 4.2). As expected, each transform method exhibits a small discrepancy compared with the analytical result. However, as the number of pixels is increased, the agreement between the transform and the analytical result improves. Even with only 70 points (the case shown in Fig. 4.2), all of the method produce reasonable agreement. While all methods show a systematic error as r approaches zero, the `basex`, `daun` (especially with 3rd-degree basis functions), `three_point`, and `onion_peeling` methods seem to provide the best agreement with the analytical result. The direct methods show fairly good agreement with the analytical curve, which is a result of the "correction" discussed above. We note that the results from the `direct_Python` and the `direct_C` methods produce identical results to within a factor of 10^{-9}.

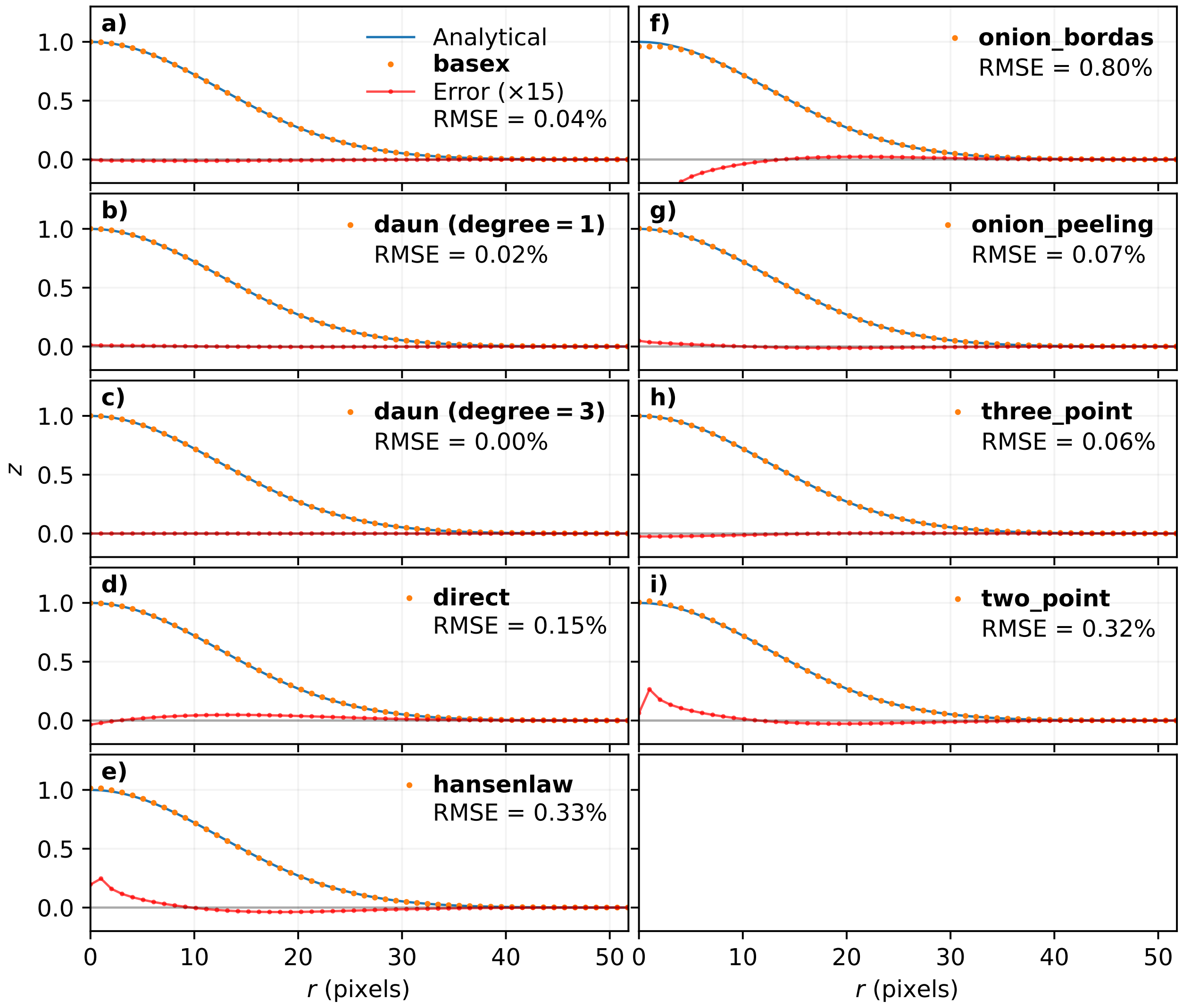


Fig. 4.2: Comparison of inverse Abel-transform methods for a 1D Gaussian function with 70 points. All of the inverse Abel transform methods show reasonable agreement for the inverse Abel transform of a Gaussian function. The root-mean-square error (RMSE) for each method is listed in the figure legend. In the limit of many pixels, the error trends to zero. However, when a small number of pixels is used, systematic errors are seen, especially near the origin ($r = 0$). The error near the origin is more pronounced in some methods than others. The lowest error seen from the `basex`, `daun`, `three_point`, and `onion_peeling` methods. The `daun` method with degree=0 is identical to `onion_peeling` and with degree=2 is slightly better (RMSE=0.05%). The `linbasex` and `rbasex` methods are not included in this figure because they are not applicable to 1D functions.

Applying the various transform methods to a synthetic "comb" function that consists of triangular peaks with one-pixel halfwidth – the sharpest features representable on the pixel grid – allows the fundamental resolution of each method to be visualized (Fig. 4.3). In order to provide an understanding of how each method responds to noise, the function transformed in Fig. 4.3 also has uniformly distributed random noise added to each pixel. The figure reveals that some methods (`basex`, `daun`, `hansenlaw`, `onion_peeling`, and `two_point`) are capable of faithfully reproducing the sharpest features, while other methods (`direct`, `onion_bordas`, and `three_point`) provide some degree of smoothing. In general, the methods that provide the highest resolution also produce the highest noise, which is most obvious at low r values. The exception is the `basex` and `daun` methods using a moderate regularization factor (Fig. 4.3 b, c), which exhibit low noise near the center, while still displaying good resolution. The `daun` method with non-negativity regularization (Fig. 4.3 d), besides producing no negative values, significantly suppresses the baseline noise without affecting the sharp features. Thus, it seems that experiments that benefit from an optimal balance of noise suppression and resolution would benefit from inverse Abel-transform methods that incorporate regularization.

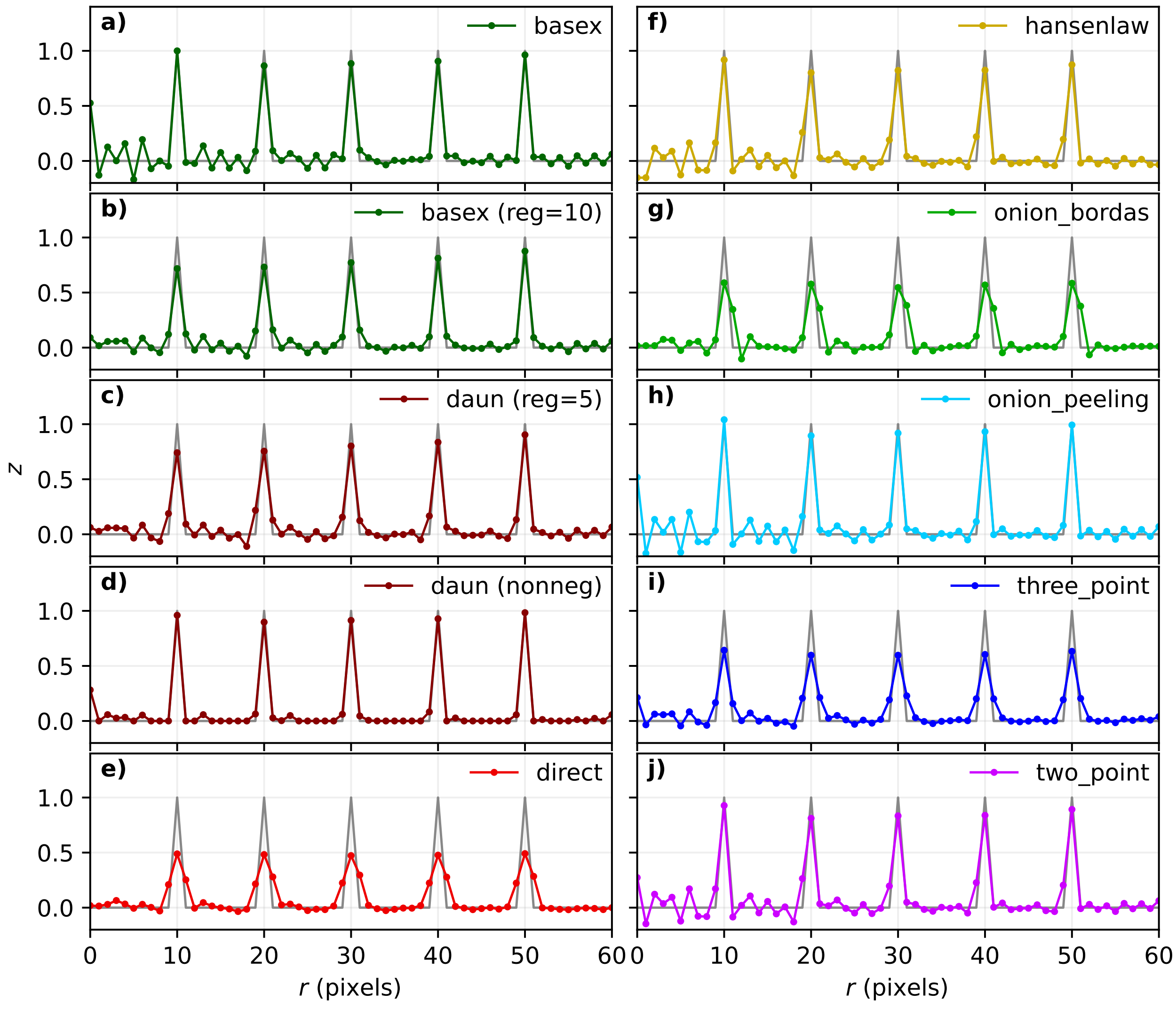


Fig. 4.3: Inverse Abel-transform methods applied to a synthetic "comb" function of one-pixel-width peaks with noise added. The gray line represents the analytical inverse Abel transform in the absence of noise. Some methods reproduce the height of the peaks, while other methods reduce noise while somewhat smoothing the peaks. The regularization in the `basex` and `daun` methods provides strong noise suppression near the origin, while maintaining peak height at higher values of r. The `daun` method without regularization is identical to `onion_peeling`, and its L_2 regularization is very similar to `basex` regularization.

Applying the various inverse Abel-transform methods to an experimental photoelectron-spectroscopy image (photoelectron spectrum of O_2^- photodetachment using a 455 nm laser, as described by Van Duzor et al. [vanduzor2010]) provides a comparison of how the noise in the reconstructed image depends on the transform method (Fig. 4.4).

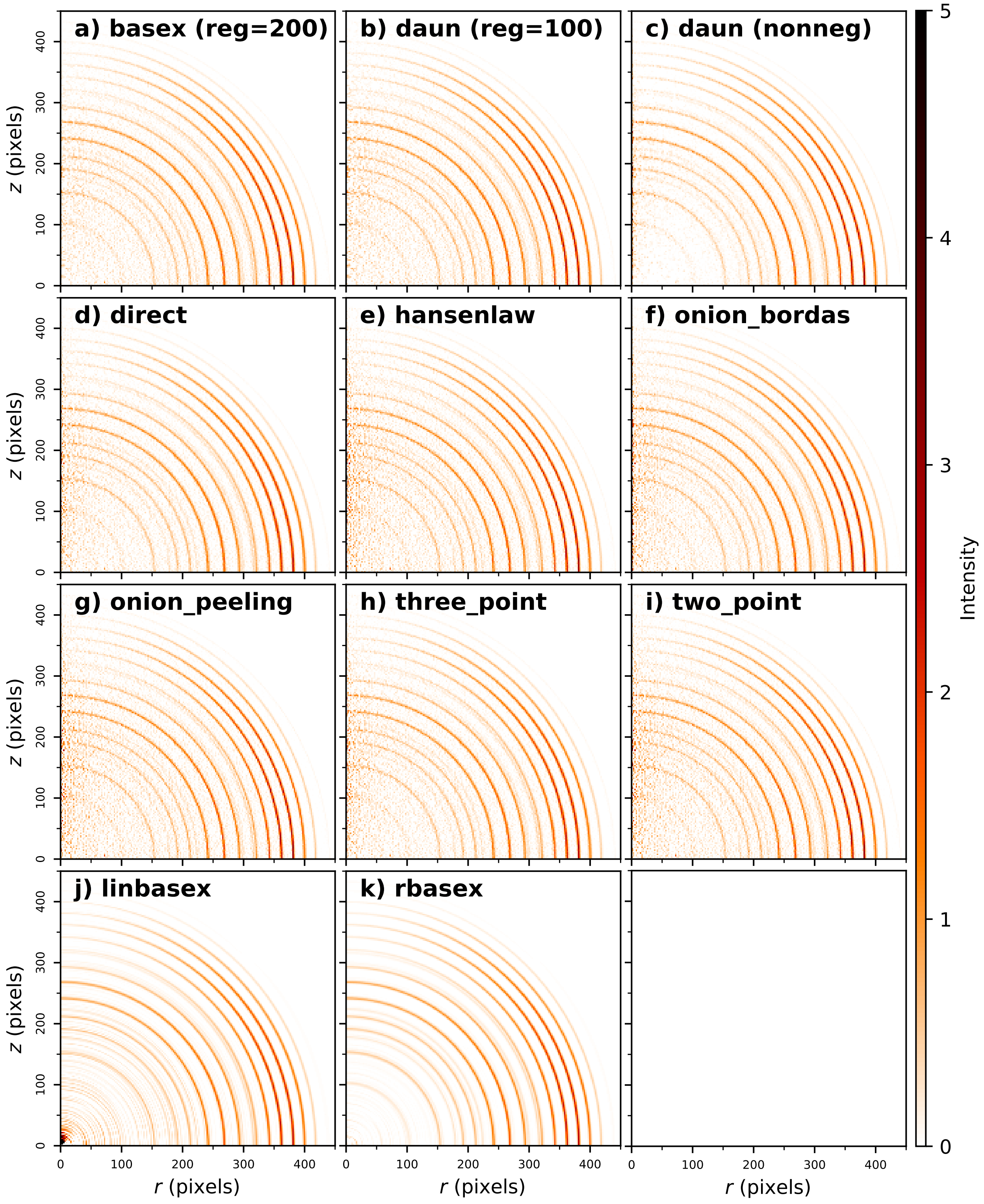


Fig. 4.4: Comparison of inverse Abel-transform methods applied to an experimental photoelectron velocity-map image. While all methods provide a faithful reconstruction of the experimental image, some of them cause a greater amplification of the noise present in the original image. The `linbasex` and `rbasex` methods models the image using a basis set of functions that vary slowly as a function of angle, which strongly reduces the high-frequency noise seen in the other transform methods. Besides the `basex` and `daun` method with regularization, the `direct` and `three_point` methods seem particularly suited for providing a low-noise transform. The `daun` method without regularization is identical to `onion_peeling`, and its L_2 regularization is very similar to `basex` regularization.

To a first approximation, the results of all the transform methods look similar. The `rbasex` and `linbasex` methods produces the "smoothest" image, which is a result of the fact that it models the projection using functions fitted to the image, that vary only slowly as a function of angle. The `basex` and `daun` methods incorporate a user-adjustable Tikhonov regularization factor, which tends to suppress noise, especially near the symmetry axis. Here, we set the regularization factor to 200 for `basex` and 100 for `daun`, which provides significant noise suppression without noticeable broadening of the narrow features. When the regularization factor is set to zero, the `basex` and `daun` methods provide a transform that appears very similar to the `onion_peeling` method. For the other transform methods, the `direct` and `three_point` methods appear to have the strongest noise-filtering properties.

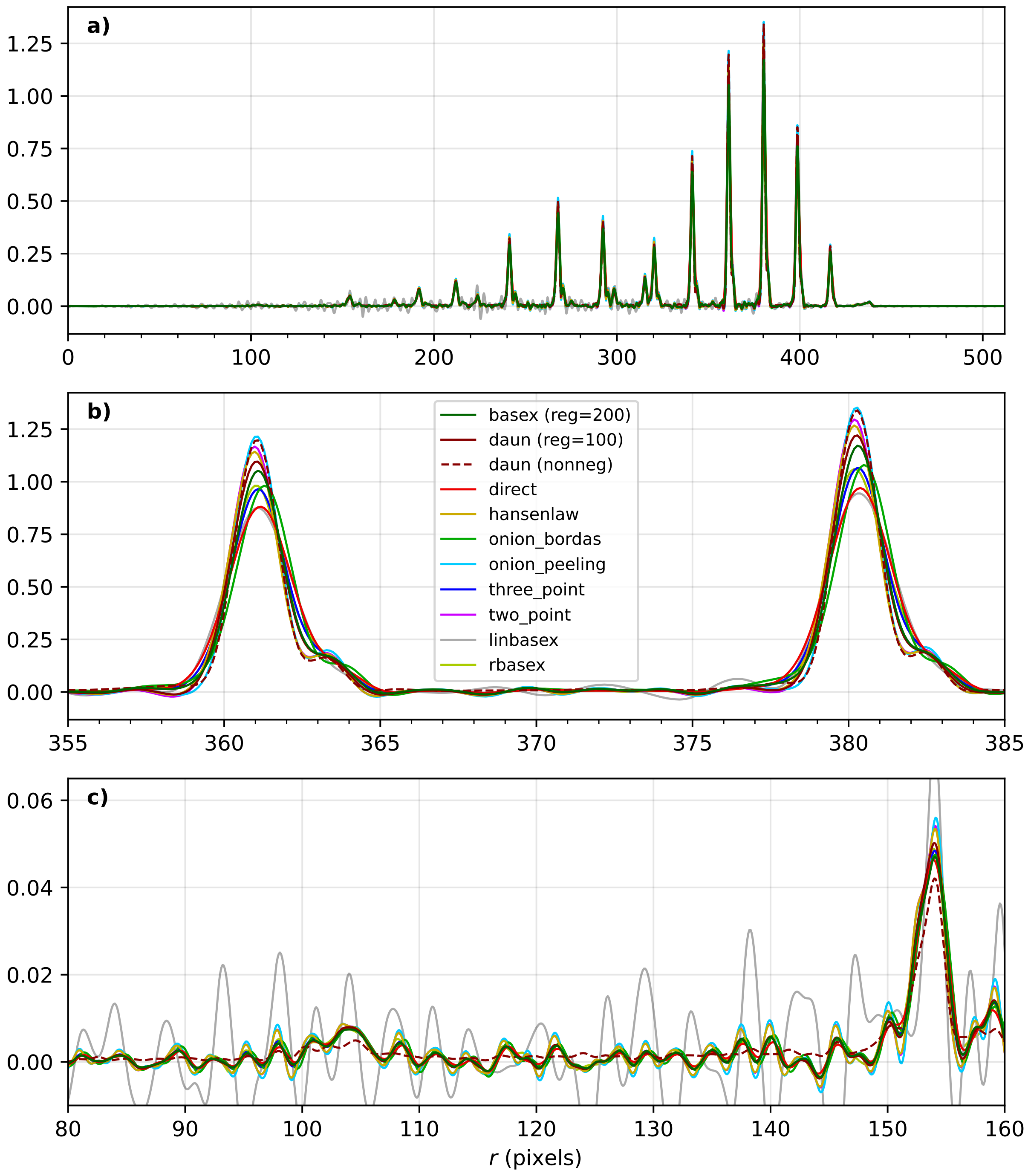


Fig. 4.5: Comparison of photoelectron spectra obtained by angular integration of the transformed images shown in Fig. 4.4, corresponding to various inverse Abel-transform methods applied to the same experimental velocity-map image. a) Looking at the entire photoelectron speed distribution, all of the transform methods appear to produce similar results. b) Closely examining two of the peaks shows that all of the methods produce similar results, but that some methods produce broader peaks than others. c) Examining the small peaks in the low-energy region reveals that some methods accumulate somewhat more noise than others. Notice the absence on negative intensities in the daun method with non-negativity regularization and the corresponding suppression of baseline oscillations.

Fig. 4.5 uses the same dataset as Fig. 4.4, but with an angular integration performed to show the 1D photoelectron spectrum. Good agreement is seen between most of the methods, even on a one-pixel level. Small but noticeable differences can be seen in the broadness of the peaks (Fig. 4.5b). The `hansenlaw`, `onion_peeling` and `two_point` methods show the sharpest peaks, suggesting that they provide enhanced ability to resolve sharp features. Of course, the differences between the methods are emphasized by the very high resolution of this dataset. In most cases, more pixels per peak yield a much better agreement between the transform methods. Interestingly, the `linbasex` method shows more baseline noise than the other methods. Fig. 4.5c shows a close examination of the two lowest-energy peaks in the image. The methods that produce that sharpest peaks (`hansenlaw`, `onion_peeling`, and `two_point`) also exhibit somewhat more noise than the rest (except `linbasex`).

4.1.8 Efficiency optimization

High-level efficiency optimization

For many applications of the inverse Abel transform, the speed at which transform can be completed is important. Even for those who are only aiming to transform a few images, the ability to perform Abel transforms efficiently may enable more effective data analysis. For example, faster Abel-transform method allow many different schemes for noise removal, smoothing, centering, and circularization to be explored more rapidly and effectively.

While PyAbel offers improvements to the raw computational efficiency of each transform method, it also provides improvements to the efficiency of the overall workflow, which are likely to provide a significant improvements for most applications. For example, since PyAbel provides a straightforward interface to switch between different transform methods (using *`abel.transform.Transform()`*), a comparison of the results from each method can easily be made and the fastest method that produces acceptable results can be selected. Additionally, PyAbel provides fast algorithms for angular and radial integration, which can be the rate-limiting step for some data-processing workflows.

In addition, when the computational efficiency of the various Abel transform methods is evaluated, a distinction must be made between those methods that can pre-compute, save, and re-use information for a specific image size (`basex`, `daun`, `linbasex`, `onion_peeling`, `rbasex`, `three_point`, `two_point`) and those that do not (`direct`, `hansenlaw`, `onion_bordas`). Often, the time required for the pre-computation is orders of magnitude longer than the time required to complete the transform. One solution to this problem is to pre-compute information for a specific image size and provide this data as part of the software. Indeed, the popular BASEX application [dribinski2002] includes a "basis set" for transforming 1000×1000-pixel images. While this approach relieves the end user of the computational cost of generating basis sets, it often means that the ideal basis set for efficiently transforming an image of a specific size is not available. Thus, "padding" is necessary for smaller images, resulting in increased computational time, while larger higher-resolution images must be downsampled or cropped.

PyAbel provides the ability to pre-compute information for any image size and cache it to disk for future use. Moreover, a cached basis set intended for transforming a larger image can be automatically cropped for use on a smaller image, avoiding unnecessary computations. The `basex` algorithm in PyAbel also includes the ability to extend a basis set intended for transforming a smaller image for use on a larger image. This allows the ideal basis set to be efficiently generated for an arbitrary image size.

Low-level computational efficiency

General Advice

Transforming very large images, or a large number of images, requires inverse Abel-transform methods with high computational efficiency. PyAbel is written in Python, a high-level programming language that is easy to read, understand, and modify. A common criticism of high-level interpreted (non-compiled) languages like Python is that they provide significantly lower computational efficiency than low-level compiled languages, such as C or Fortran. However, such slowdowns can be avoided by calling functions from optimized math libraries for the key operations that serve as bottlenecks. For most of the transform methods (and indeed, all of the fastest methods), the operation that bottlenecks the transform process is a matrix-algebra operation, such as matrix multiplication. PyAbel uses matrix-algebra functions provided by the NumPy library, which are, in turn, provided by the Basic Linear Algebra Subprograms (BLAS) library. Thus, the algorithms in PyAbel have comparable performance to optimized C/Fortran.

One subtle consequence of this reliance on the BLAS algorithms is that the performance is dependent on the exact implementation of BLAS that is installed, and users seeking the highest level of performance may wish to experiment with different implementations. Different NumPy/SciPy distributions use different libraries by default, and some also provide a choice between several libraries. If the transform speed is important, it is advisable to run the benchmarks on all available configurations to select the fastest for the specific combination of the transform method, operating system and hardware.

Among the widely available options, the Intel Math Kernel Library (MKL) generally provides the best performance for Intel CPUs, although its installed size is rather huge and its performance on AMD CPUs is quite poor. It is used by default in Anaconda Python. OpenBLAS generally provides the best performance for AMD CPUs and reasonably good performance for Intel CPUs. It is used by default in some distributions. AMD develops numerical libraries optimized for its own CPUs, but they are not yet officially integrated with NumPy/SciPy.

Another important issue for modern Intel CPUs is that they suffer a performance degradation when denormal numbers are encountered, which sometimes happens in the intermediate calculations even if the input and output are “normal”. In this case, configuring the CPU to treat denormals as zeros does help. There is no official way to achieve this in NumPy/SciPy, but a third-party module daz can be used for this purpose. At least some modern AMD CPUs are less or not affected by this issue, although it’s always better to run the tests to make sure.

Speed benchmarks

The `abel.benchmark.AbelTiming` class provides the ability to benchmark the speeds of the Abel transform algorithms. Here we show these benchmarks completed using a personal computer equipped with a 3.0 GHz Intel i7-9700 processor and 32 GB RAM running GNU/Linux (see also *Additional speed benchmarks* for some other systems).

A comparison of the time required to complete an inverse Abel transform versus the width of a square image is presented in Fig. 4.6. All methods are benchmarked using their default parameters, with the following exceptions:

- **basex(var)** and **daun(var)** mean “variable regularization”, that is changing the regularization parameter for each transformed image.
- **daun(nonneg)** shows only the result of transforming the O_2^- image (see Fig. 4.4) with non-negativity regularization. Since the time needed for this non-linear transform strongly depends on the data, it is impossible to provide “universal” benchmarks; however, the general scaling is also expected to be roughly cubic.
- **direct_C** and **direct_Python** correspond to the “direct” method using its C (Cython) and Python backends respectively.
- **linbasex** and **rbasex** show whole-image ($n \times n$) transforms, while all other methods show half-image (n rows, $(n+1)/2$ columns) transforms.
- **rbasex(None)** means no output-image creation (only the transformed radial distributions).

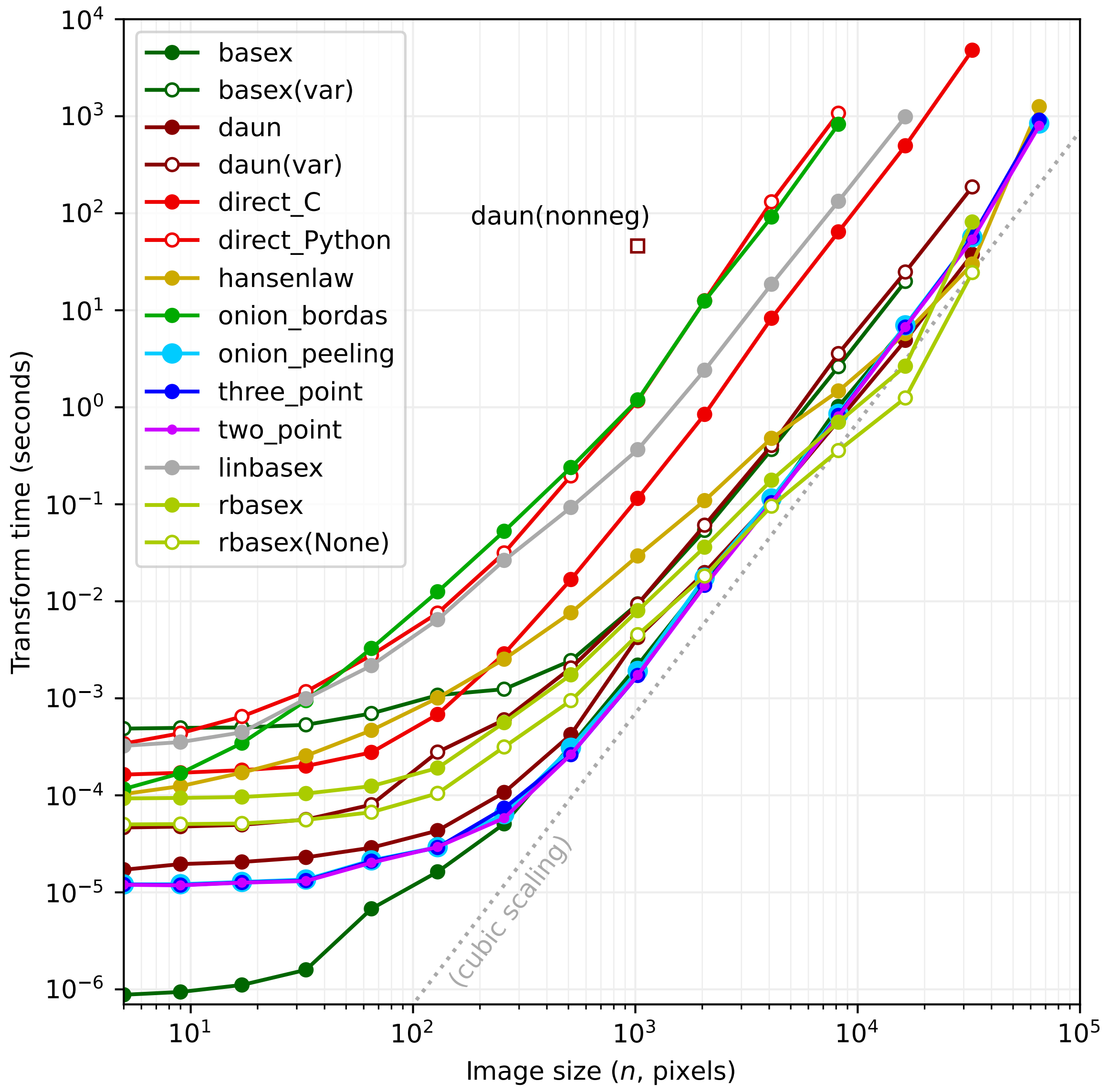


Fig. 4.6: Computational efficiency of inverse Abel-transform methods. The time to complete an inverse Abel transform increases with the size of the image. Most of the methods display a roughly n^3 scaling (dashed gray line). The `basex`, `onion_peeling`, `three_point`, and `two_point` methods all rely on similar matrix-algebra operations as their rate-limiting step, and consequently exhibit identical performance for typical experimental image sizes.

Fig. 4.6 reveals the computational scaling of each method as the image size is increased. At image sizes below $n = 100$, most of the transform methods exhibit a fairly flat relationship between image size and transform time, suggesting that the calculation is limited by the computational overhead. For image sizes of 1000 pixels and above, all the methods show a steep increase in transform time with increasing image size. A direct interpretation of the integral for the inverse Abel transform involves three nested loops, one over z, one over r, and one over y, and we should expect n^3 scaling. Indeed, the `direct_C` and `direct_Python` methods scale as nearly n^3. Several of the fastest methods (`basex`, `onion_peeling`, `three_point`, and `two_point`) rely on matrix multiplication (or back substitution in case of `daun`). These methods also scale roughly as n^3, which is approximately the expected scaling for matrix-multiplication operations [coppersmith1990]. For typical image sizes (~500–1000 pixels width), `basex`, `daun` and the methods of Dasch [dasch1992] consistently out-perform other methods, often by several orders of magnitude. Interestingly, the `hansenlaw` and `rbasex` algorithms exhibits a nearly n^2 scaling and should outperform other algorithms for large image sizes. While the `linbasex` method does not provide the fastest transform, we note that it analytically provides the angular-integrated intensity and anisotropy parameters. Thus, if those parameters are desired outcomes – as they often are during the analysis of photoelectron spectroscopy datasets – then `linbasex` may provide an efficient analysis. The `rbasex` method also provides the intensity and anisotropy distributions directly. Moreover, if only these qualities are needed, without the transformed image, the transform can be completed faster and starts to outperform the fastest general-purpose methods for image sizes of $\gtrsim$1000 pixels (extracting the desired distributions from the results of these methods requires additional time, not included in their plotted transform times).

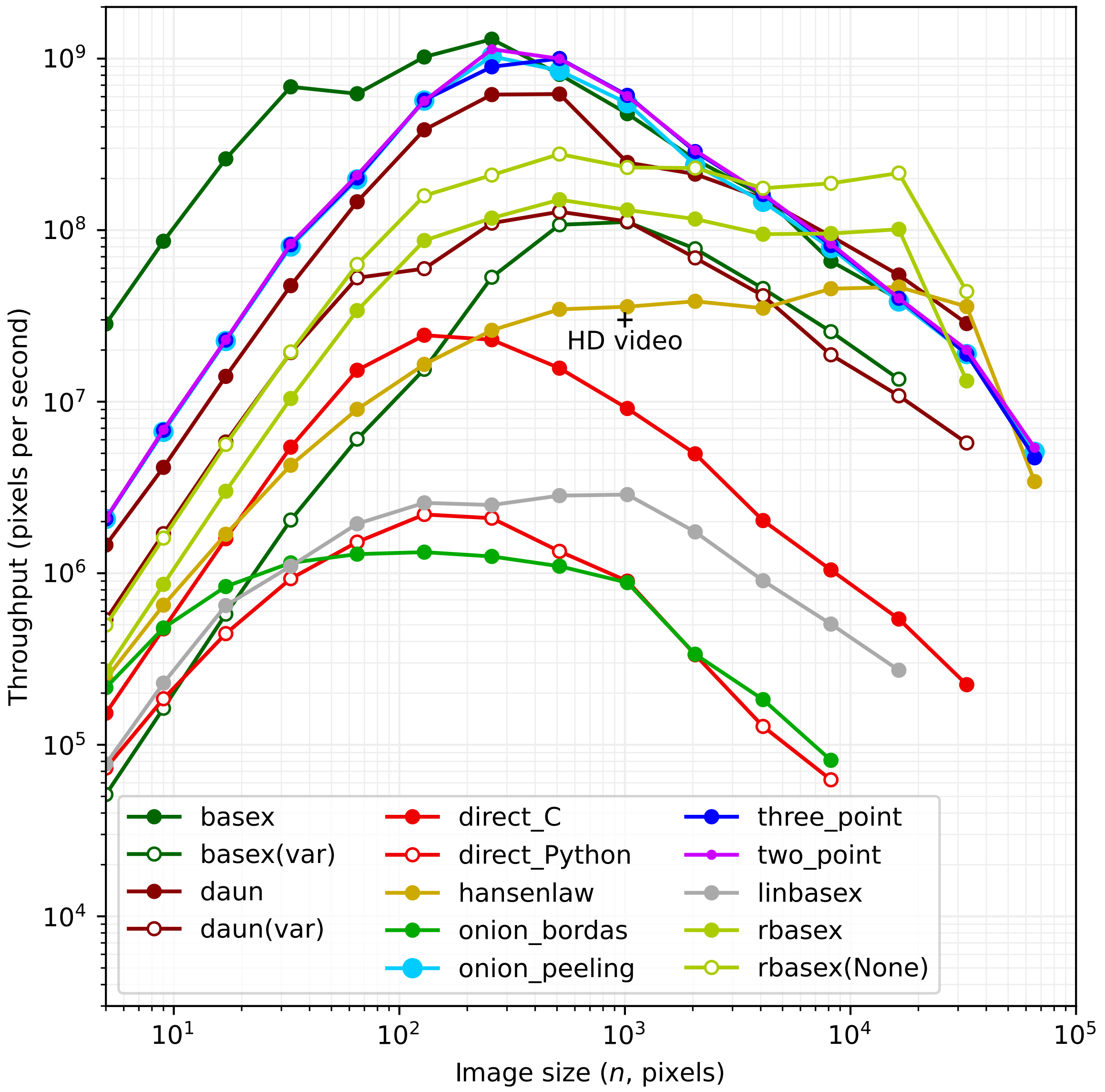


Fig. 4.7: The performance can also be viewed as a pixels-per-second rate. Here, it is clear that some methods provide sufficient throughput to transform images at rates far exceeding high-definition video (1000×1000 pixels at 30 frames per second is 3×10^7 pixels per second).

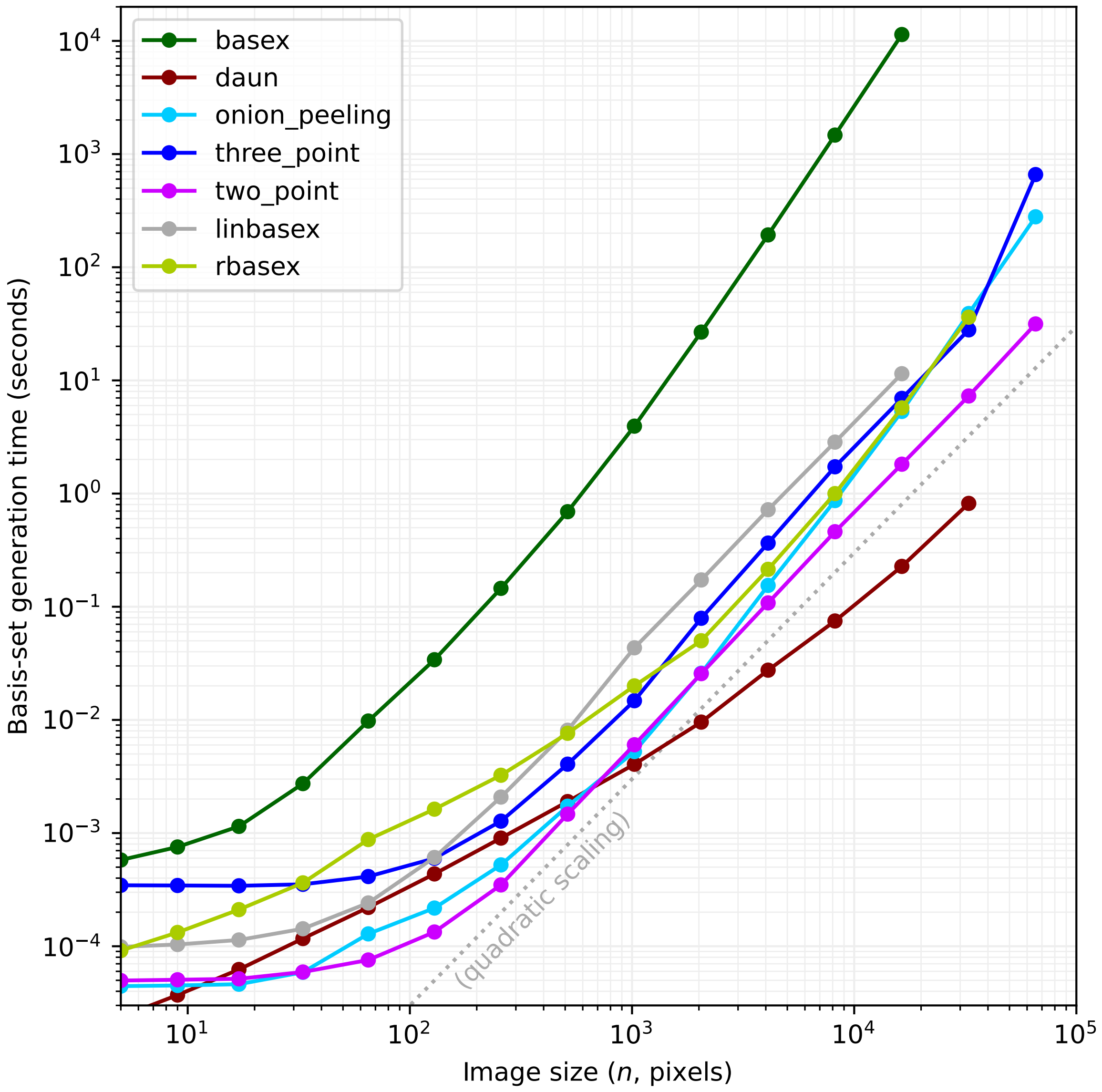


Fig. 4.8: Computational efficiency of the basis-set generation calculation.

The `basex`, `onion_peeling`, `three_point`, and `two_point` methods run much faster if appropriately sized basis sets have been pre-calculated. For the `basex` method, the time for this pre-calculation is orders of magnitude longer than the transform time (Fig. 4.8). For the Dasch methods (`onion_peeling`, `three_point`, and `two_point`), the pre-calculation is significantly longer than the transform time for image sizes smaller than 2000 pixels. For larger image sizes, the pre-calculation of the basis sets approaches the same speed as the transform itself. In particular, for the `two_point` method, the pre-calculation of the basis sets actually becomes faster than the image transform for $n \gtrsim 4000$. For the `daun` and `linbasex` methods, the pre-calculation of the basis sets is consistently faster than the transform itself, suggesting that the pre-calculation of basis sets isn't necessary for these methods.

4.1.9 Conclusion

The various Abel-transform methods in PyAbel provide advantages for different situations. Nevertheless, certain recommendations can be made.

Methods recommended for general-purpose Abel transforms:

- `basex`
- `daun`
- `hansenlaw`
- `three-point`
- `two-point`
- `onion-peeling`
- `direct`

Methods recommended for photoelectron/photoion datasets, or for images with similar shape:

- `rbasex`
- `linbasex`

Methods recommended for educational purposes only (these methods are generally slower and somewhat less accurate than competing transform methods):

- `onion_bordas`

Additional speed benchmarks

- *Intel i7-9700 (Linux)*
- *Intel i7-6700 (Linux)*
- *AMD Ryzen 5 5600G (Linux)*
- *AMD Ryzen 5 5600G (Windows)*
- *Raspberry Pi 4B (Linux)*

All methods are benchmarked using their default parameters, with the following exceptions:

- **basex(var)** and **daun(var)** mean “variable regularization”, that is changing the regularization parameter for each transformed image.
- **direct_C** and **direct_Python** correspond to the “direct” method using its C (Cython) and Python backends respectively.
- **linbasex** and **rbasex** show whole-image ($n \times n$) transforms, while all other methods show half-image (n rows, $(n + 1)/2$ columns) transforms.
- **rbasex(None)** means no output-image creation (only the transformed radial distributions).

Intel i7-9700 (Linux)

CPU
Intel Core i7-9700 (8 cores, 8 threads; 3.0 GHz base, 4.7 GHz max)

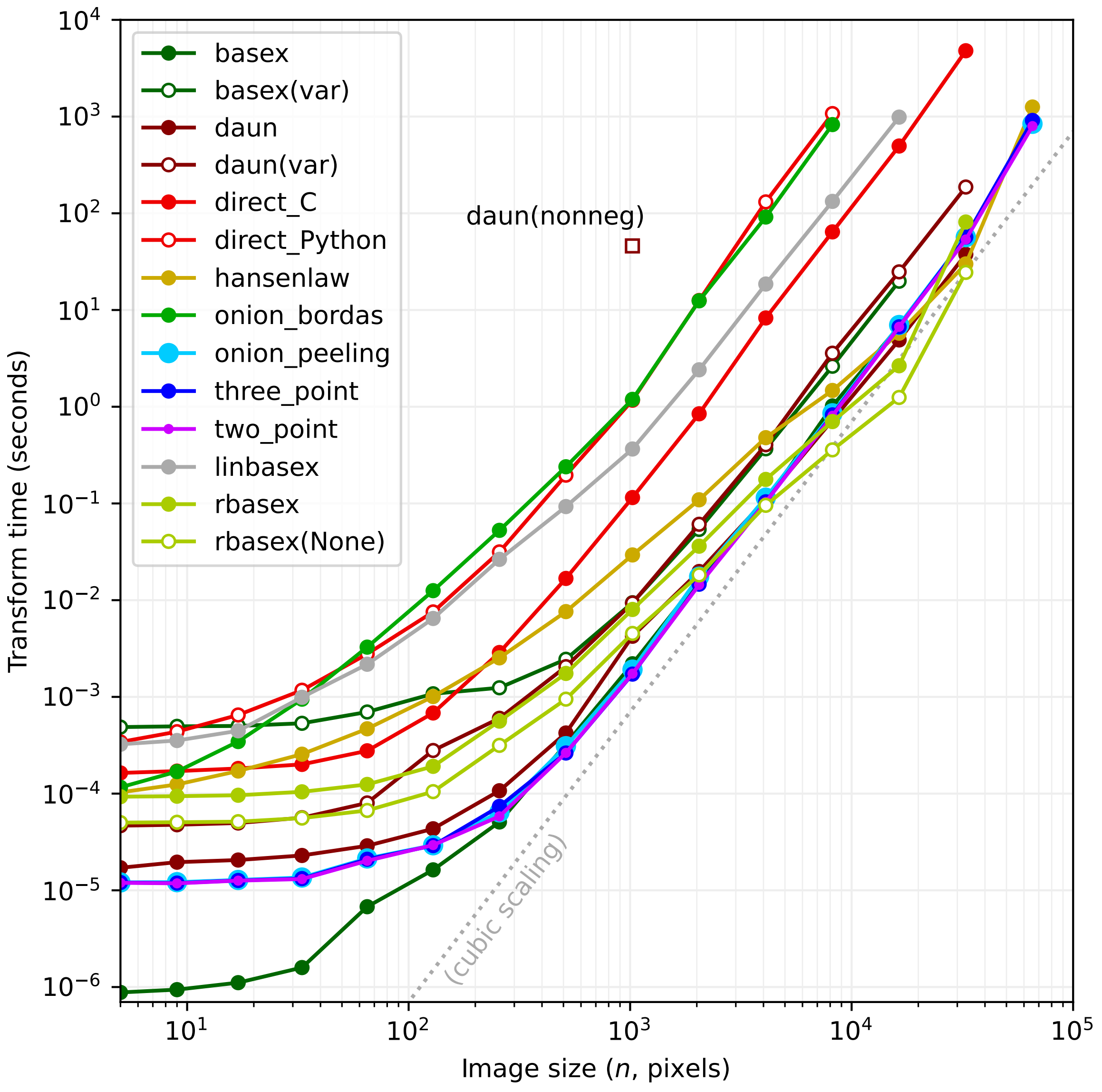
RAM
32 GB DDR4-2666

OS
Ubuntu 20.04 LTS

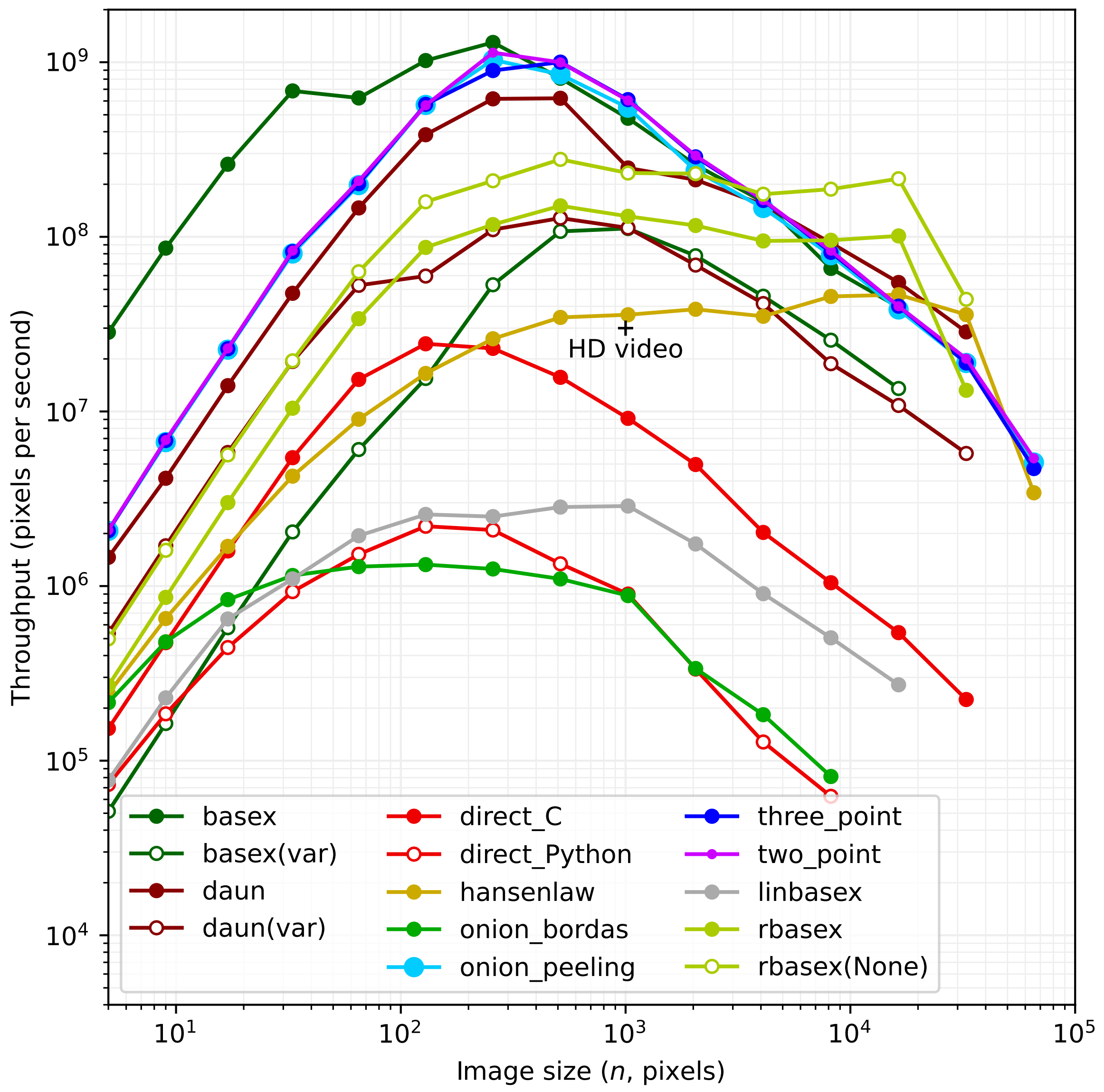
Libraries

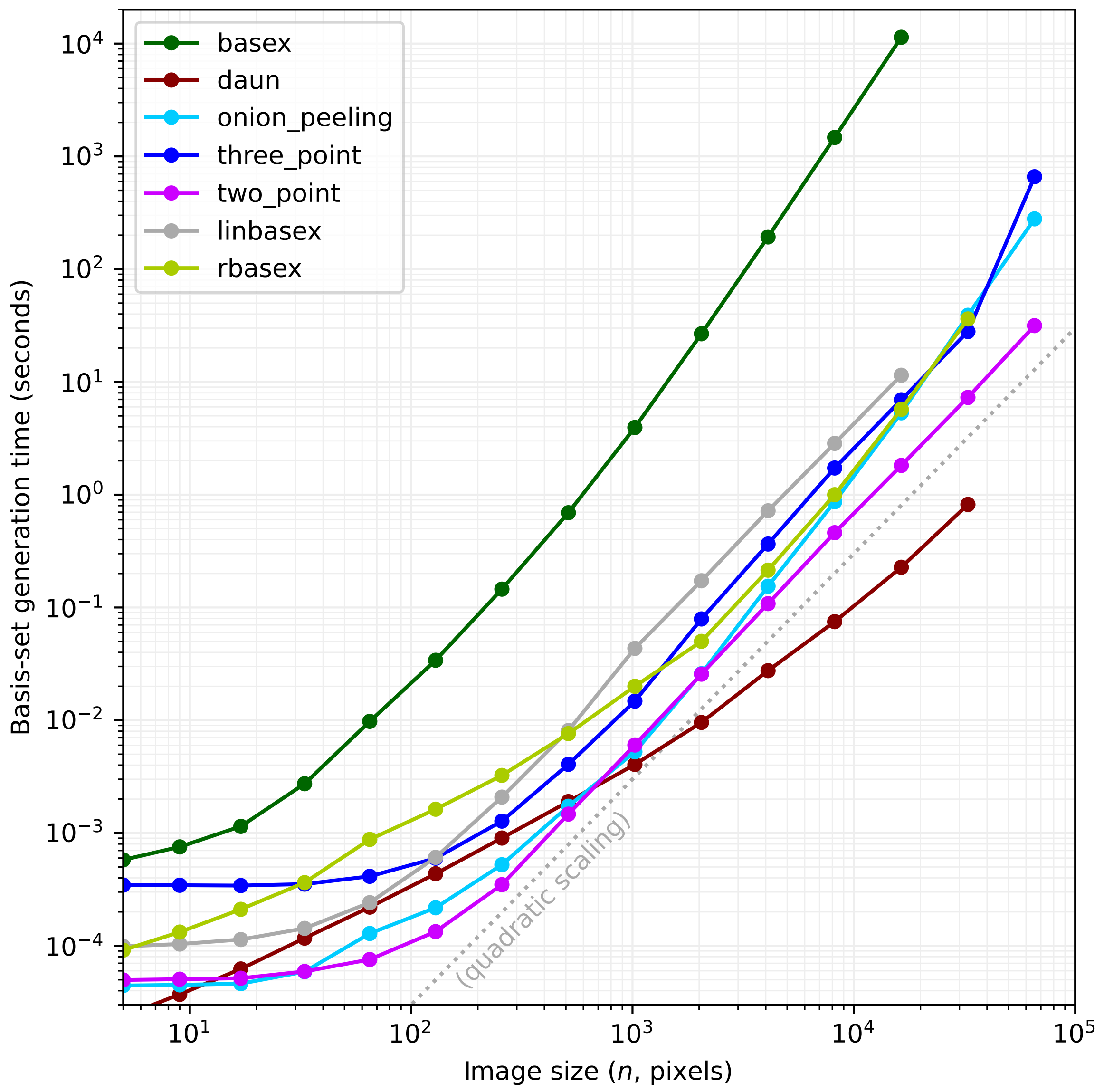
- NumPy 1.18.1
- SciPy 1.4.1
- MKL 2020
- daz

Results



basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
daun(nonneg)
(cubic scaling)
Transform time (seconds)
Image size (n, pixels)


Throughput (pixels per second)
10^4
10^5
10^6
10^7
10^8
10^9
Image size (n, pixels)
10^1
10^2
10^3
10^4
10^5
HD video
basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)



Intel i7-6700 (Linux)

CPU
Intel Core i7-6700 (4 cores, 8 threads; 3.4 GHz base, 4.0 GHz max)

RAM
32 GB DDR4-2133

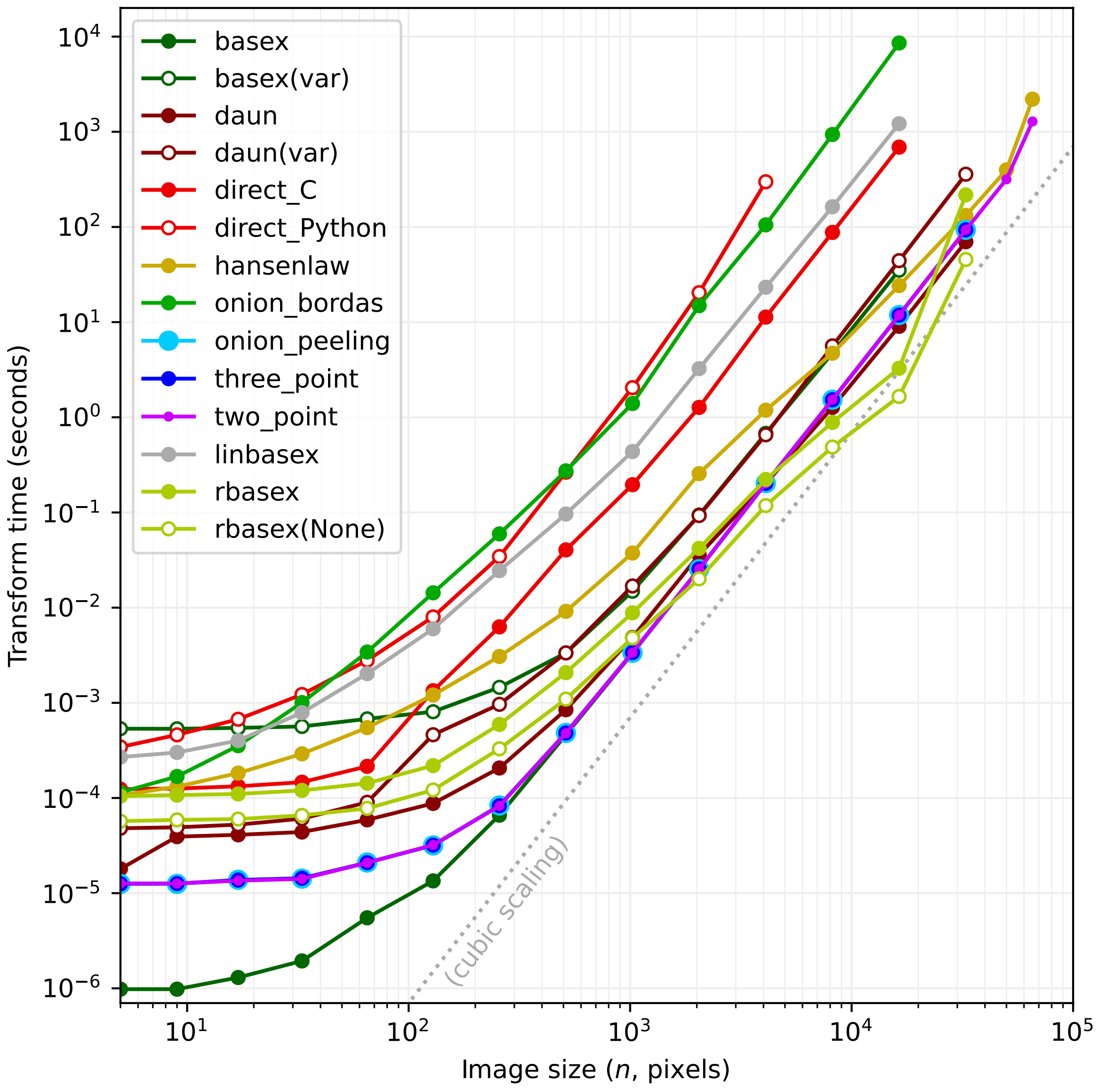
OS
Ubuntu 19.10

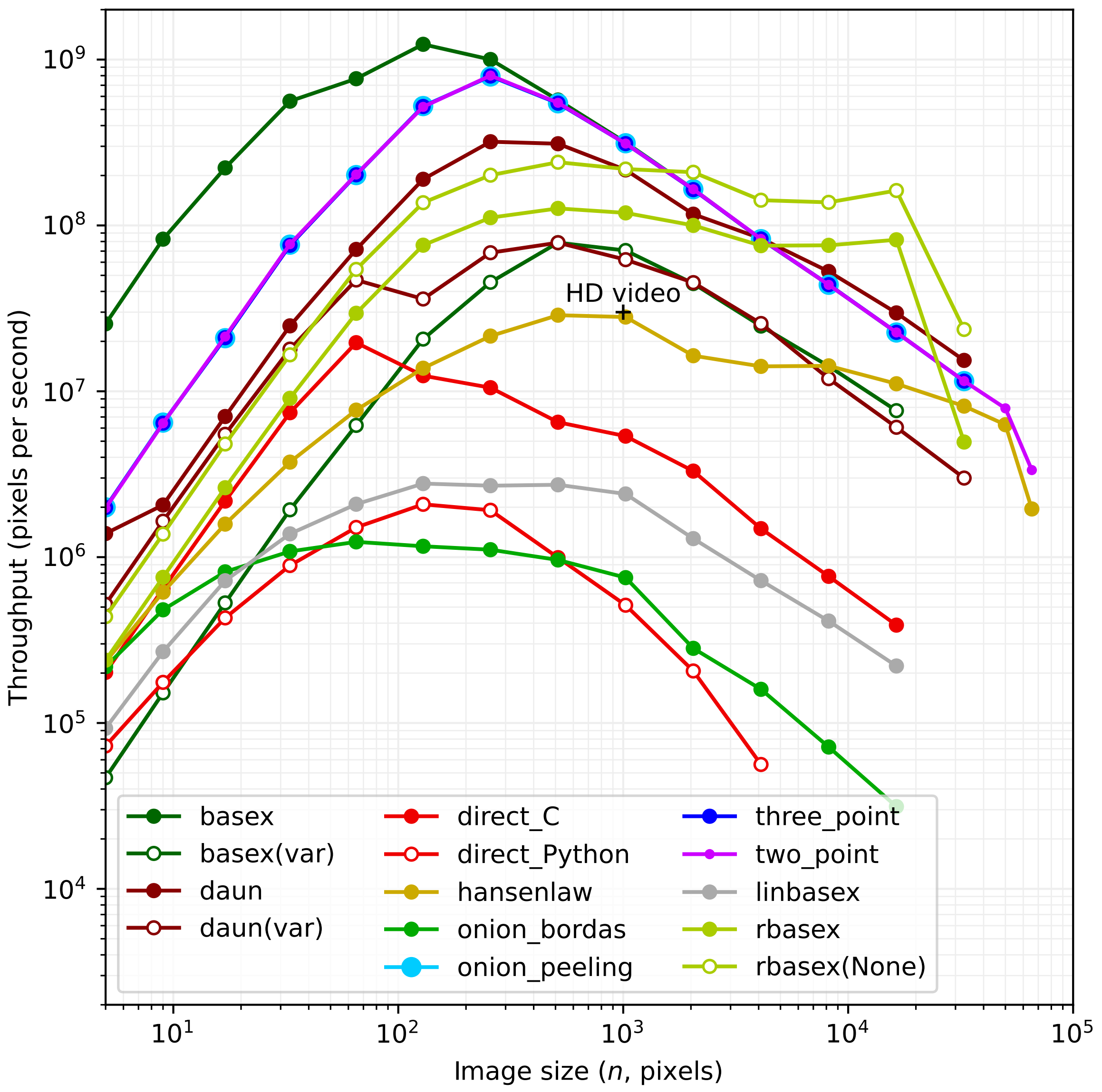
Libraries

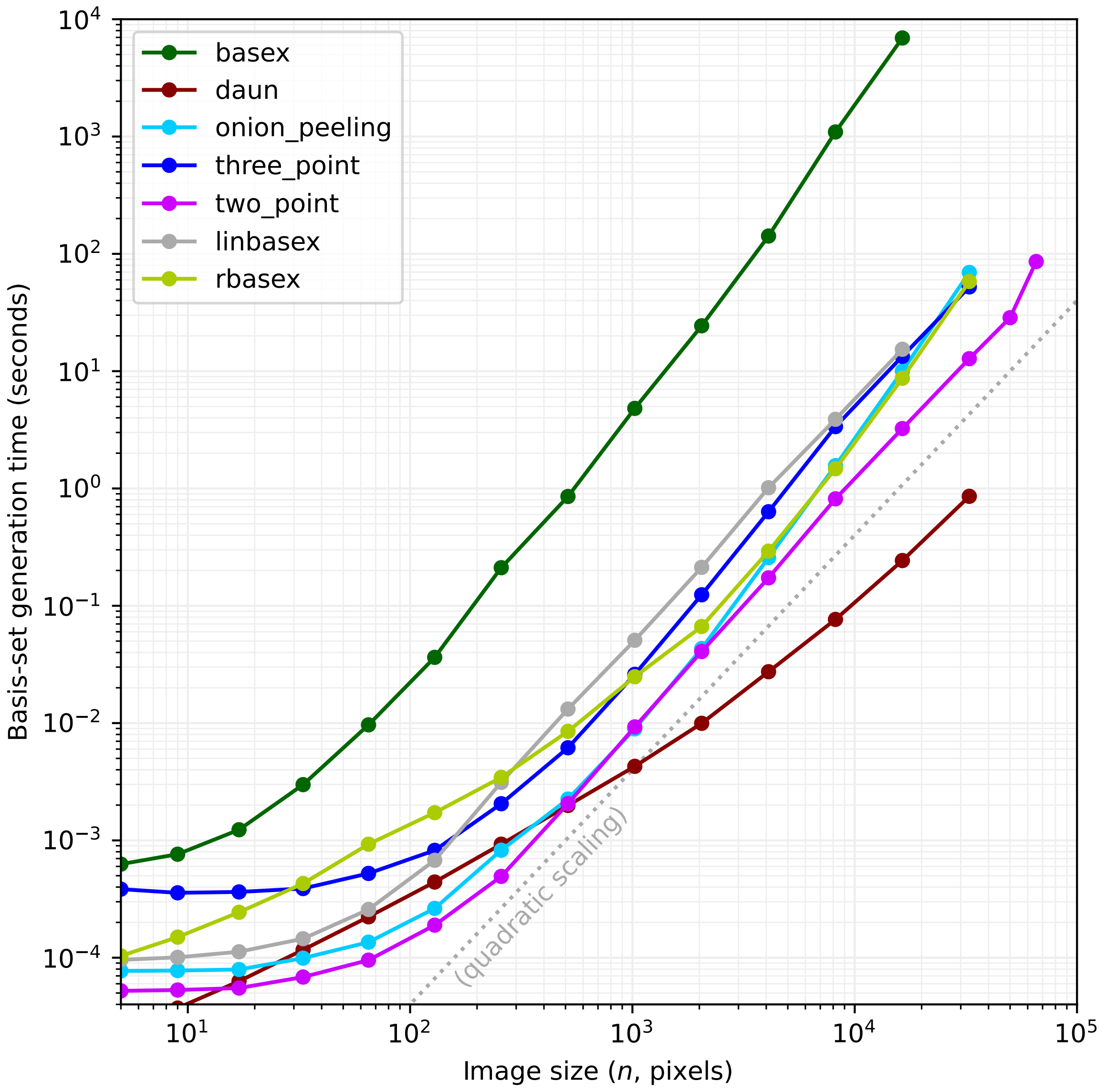
- NumPy 1.18.1
- SciPy 1.4.1
- MKL 2019 Update 5

- daz

Results


basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
(cubic scaling)
Transform time (seconds)
Image size (n, pixels)


Throughput (pixels per second)
Image size (n, pixels)
HD video
basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}
10^{9}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}



AMD Ryzen 5 5600G (Linux)

CPU
AMD Ryzen 5 5600G (6 cores, 12 threads; 3.9 GHz base, 4.4 GHz max)

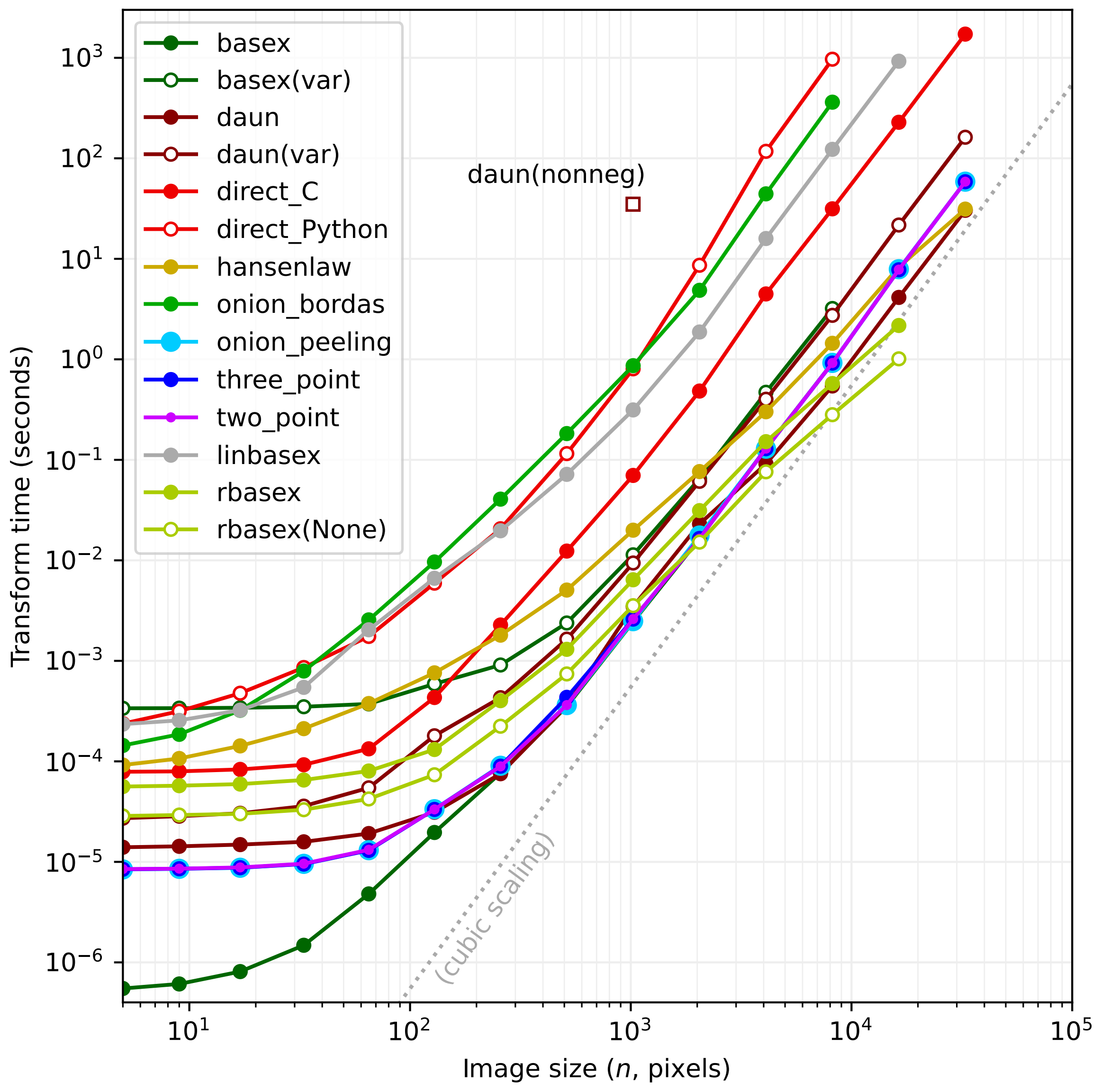
RAM
32 GB DDR4-3200

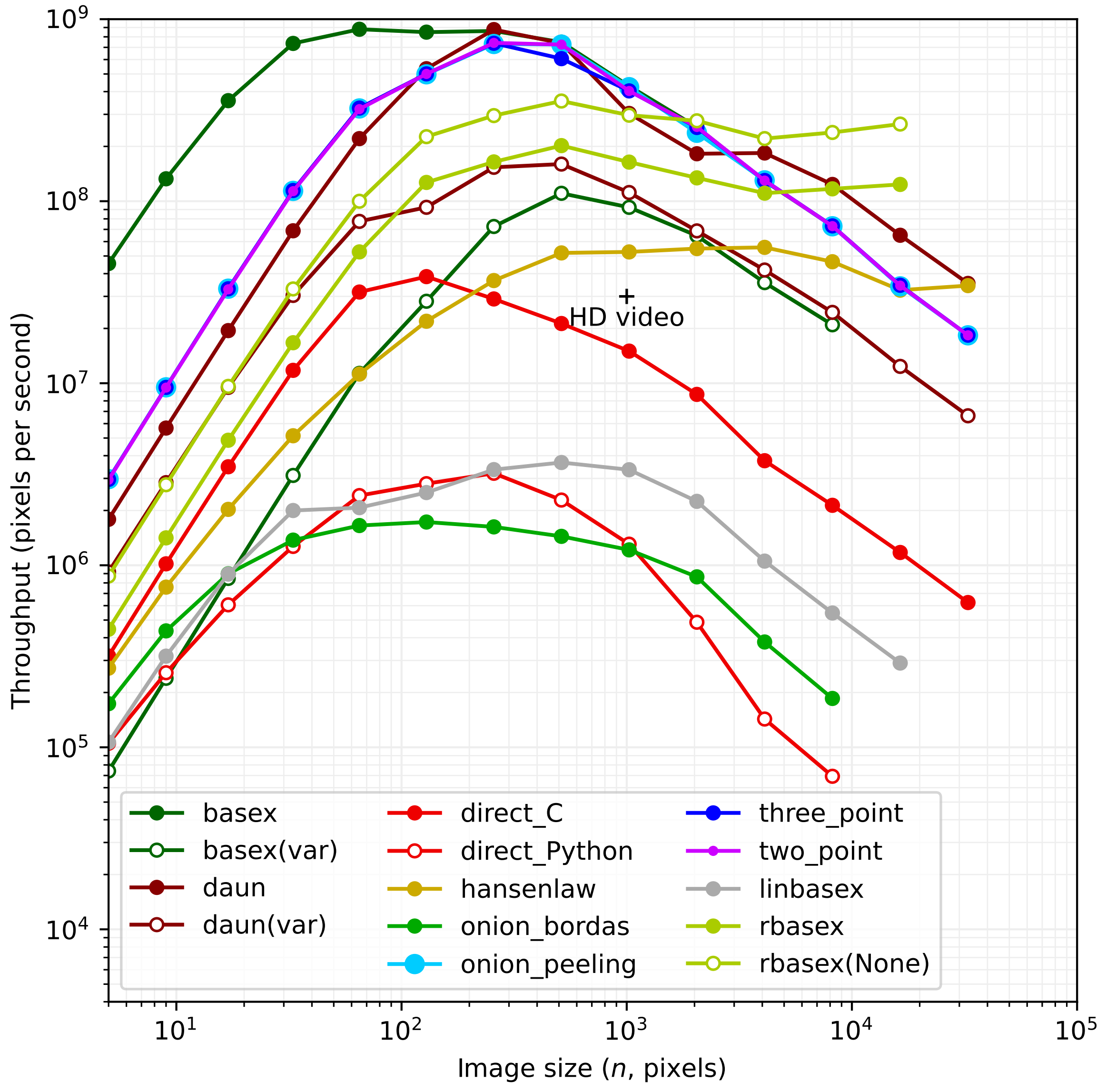
OS
Debian GNU/Linux 12

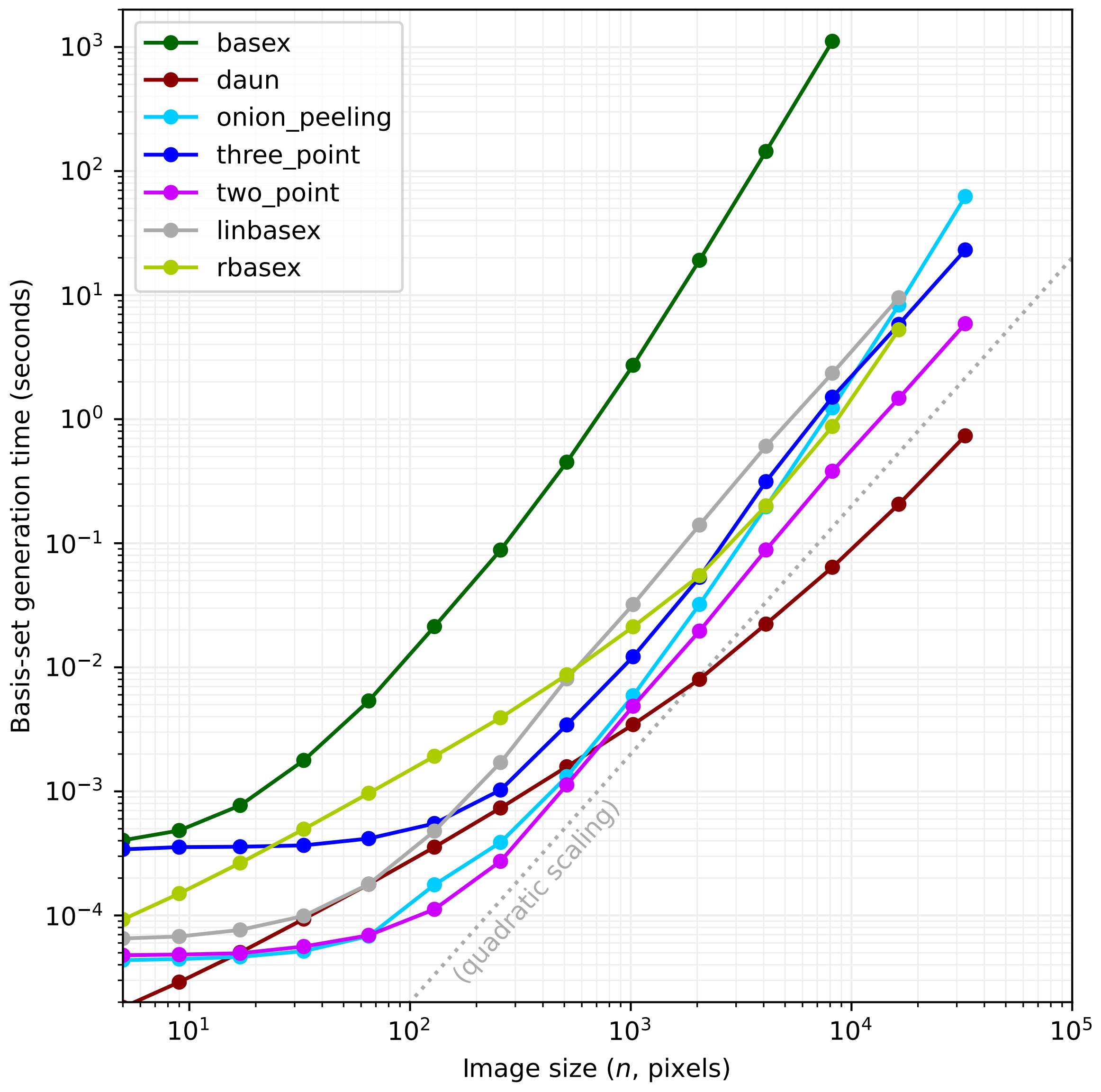
Libraries

- NumPy 1.24.2
- SciPy 1.10.1
- OpenBLAS 0.3.21

Results


basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
daun(nonneg)
(cubic scaling)
Transform time (seconds)
Image size (n, pixels)
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}


10^9
10^8
10^7
10^6
10^5
10^4
Throughput (pixels per second)
10^1
10^2
10^3
10^4
10^5
Image size (n, pixels)
HD video
basex
basex(var)
daun
daun(var)
direct_C
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)



AMD Ryzen 5 5600G (Windows)

CPU
AMD Ryzen 5 5600G (6 cores, 12 threads; 3.9 GHz base, 4.4 GHz max)

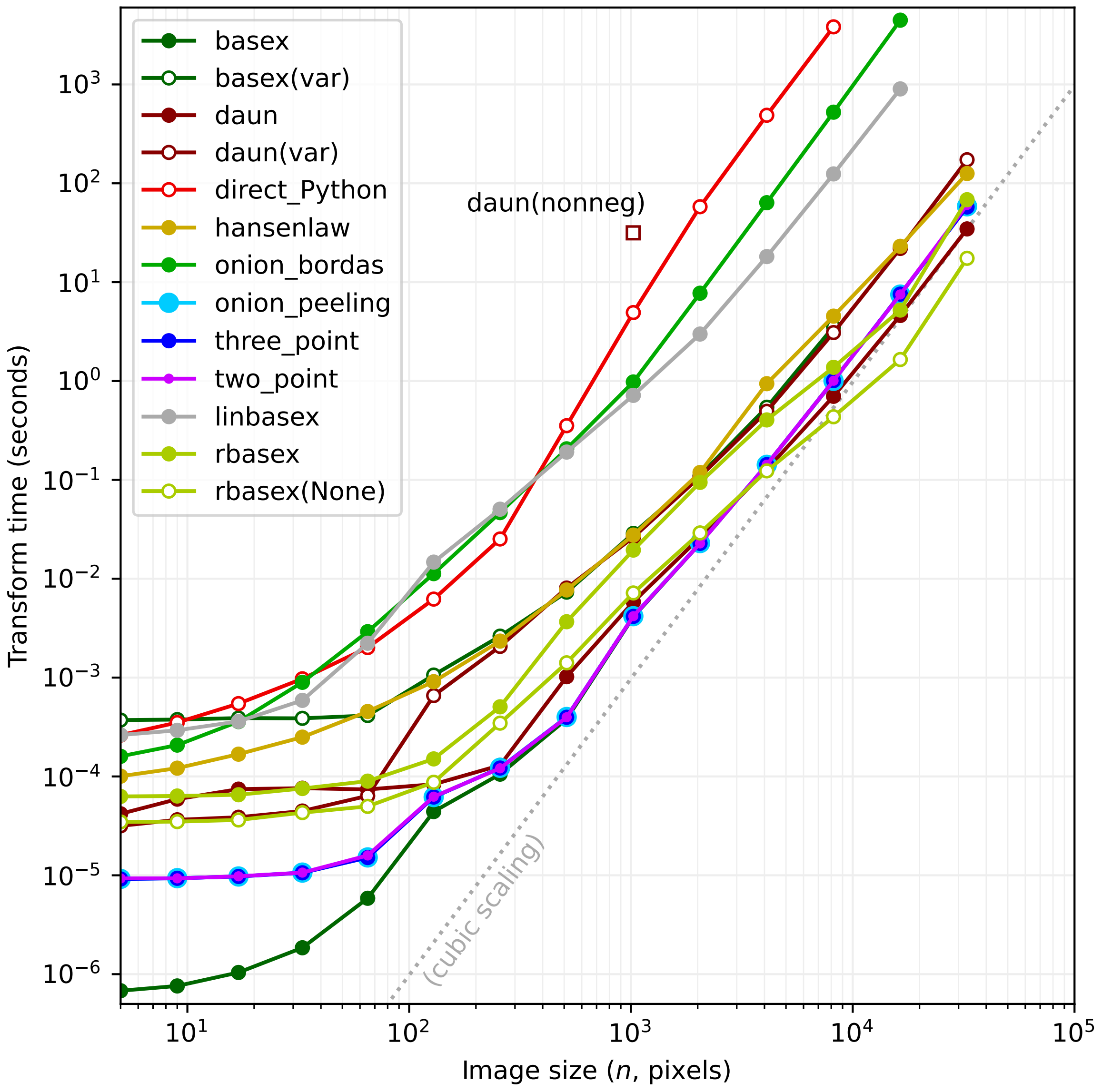
RAM
32 GB DDR4-3200

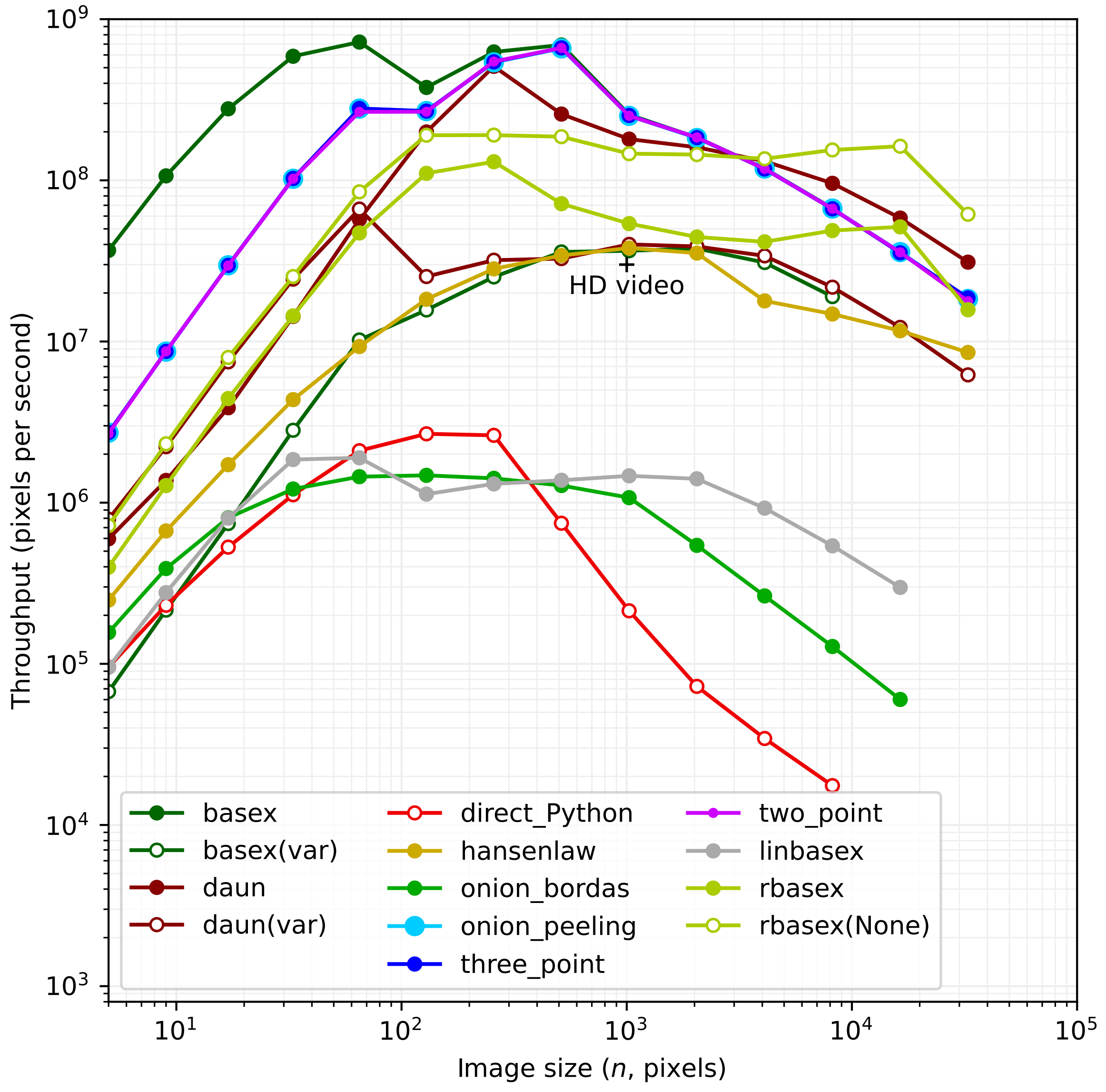
OS
Microsoft Windows 11

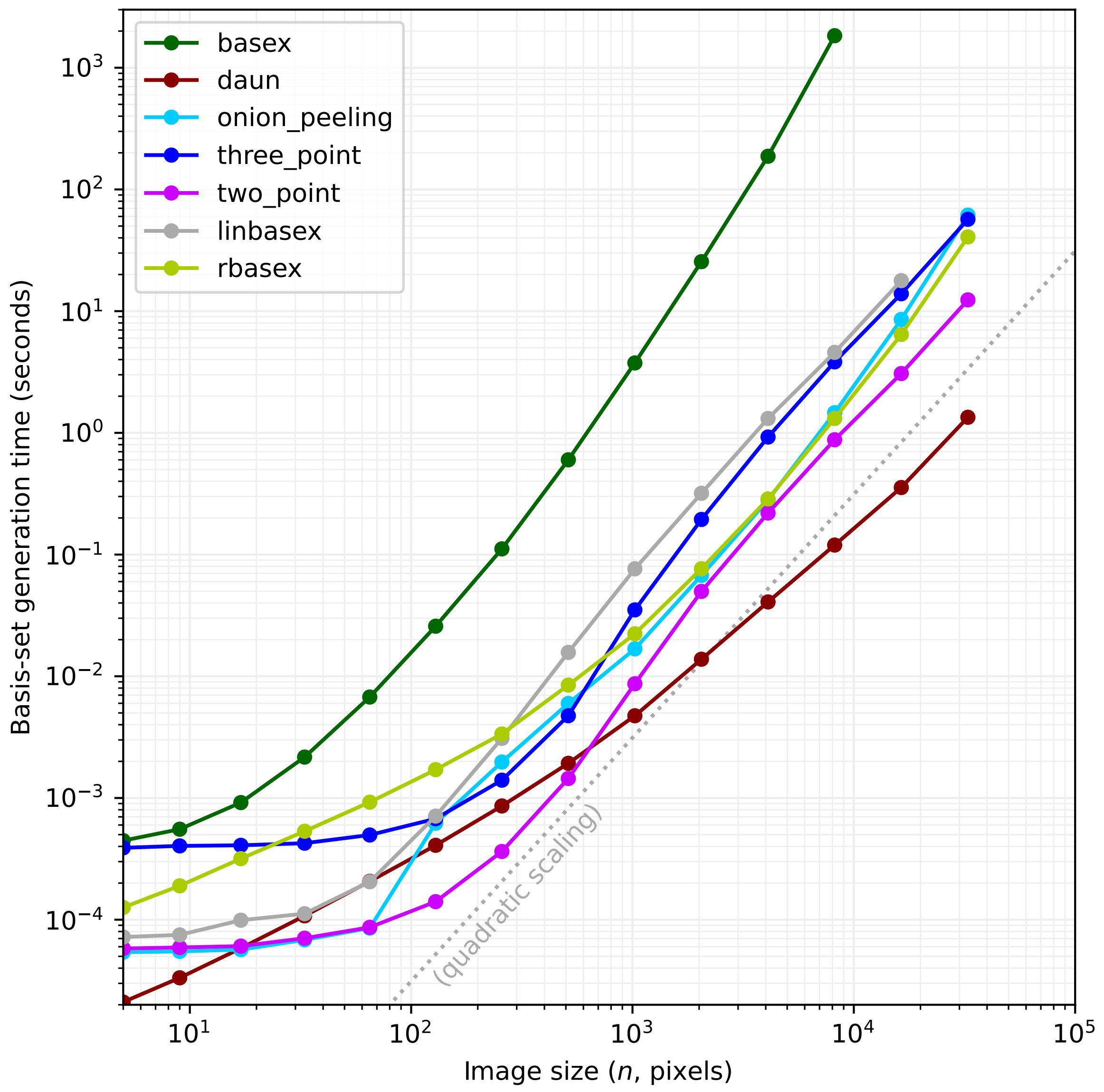
Libraries

- NumPy 1.26.0
- SciPy 1.11.2
- OpenBLAS 0.3.23

Results


basex
basex(var)
daun
daun(var)
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
daun(nonneg)
(cubic scaling)
Transform time (seconds)
Image size (n, pixels)
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}


Throughput (pixels per second)
Image size (n, pixels)
10^9
10^8
10^7
10^6
10^5
10^4
10^3
10^1
10^2
10^3
10^4
10^5
HD video
basex
basex(var)
daun
daun(var)
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)



Raspberry Pi 4B (Linux)

CPU
Broadcom BCM2711 (4 cores; 1.5 GHz)

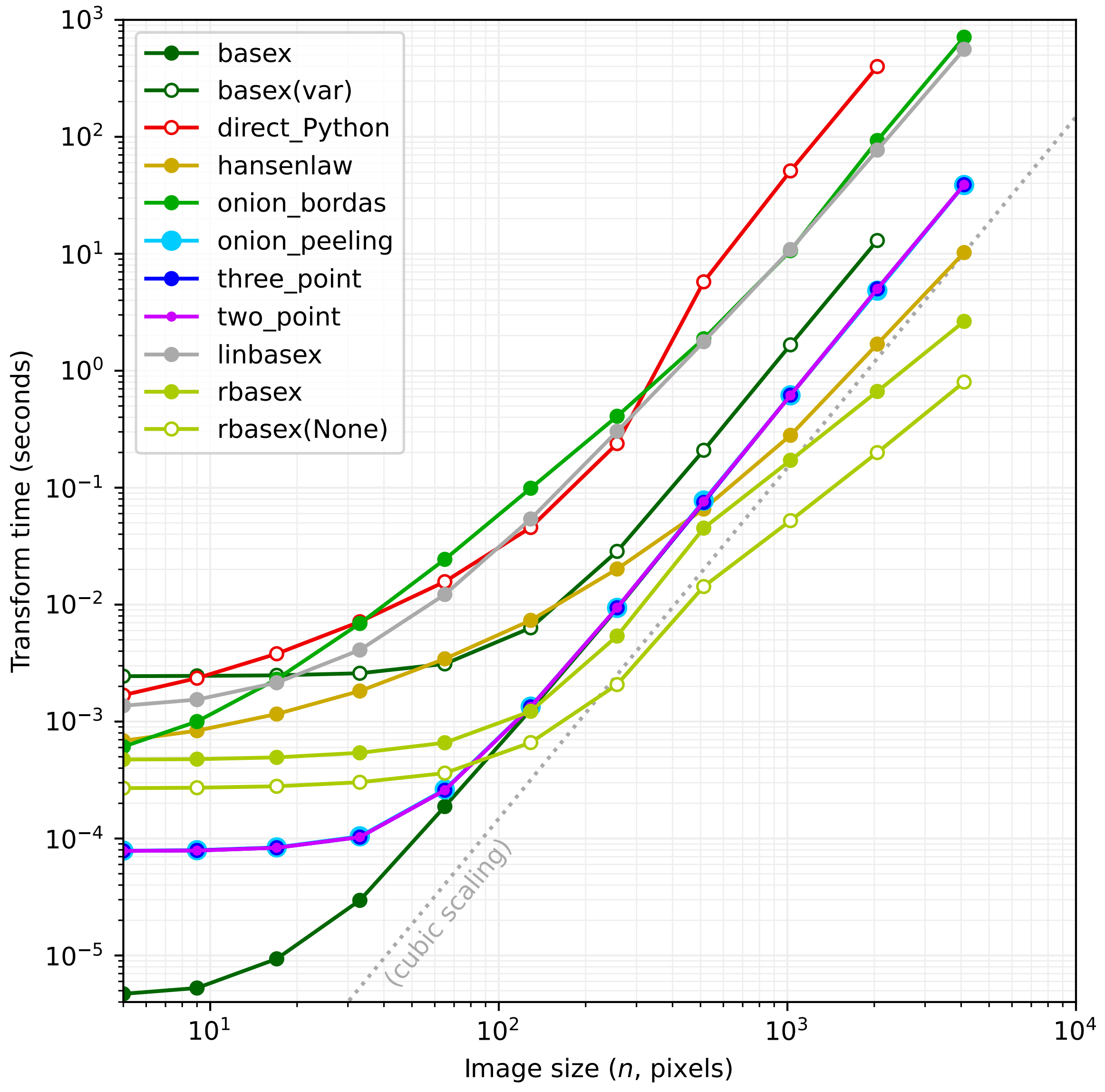
RAM
4 GB LPDDR4-3200

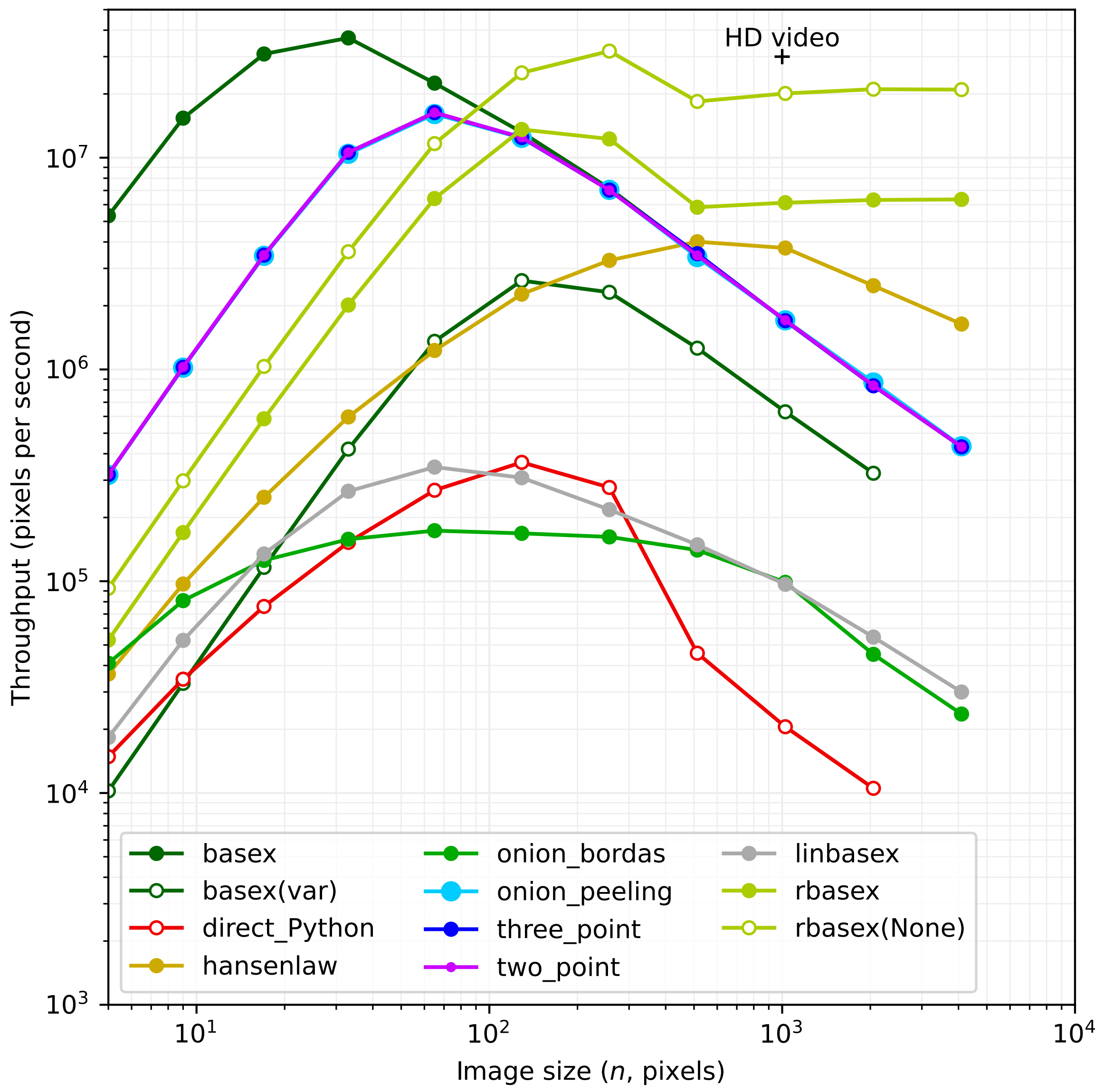
OS
Raspbian GNU/Linux 10

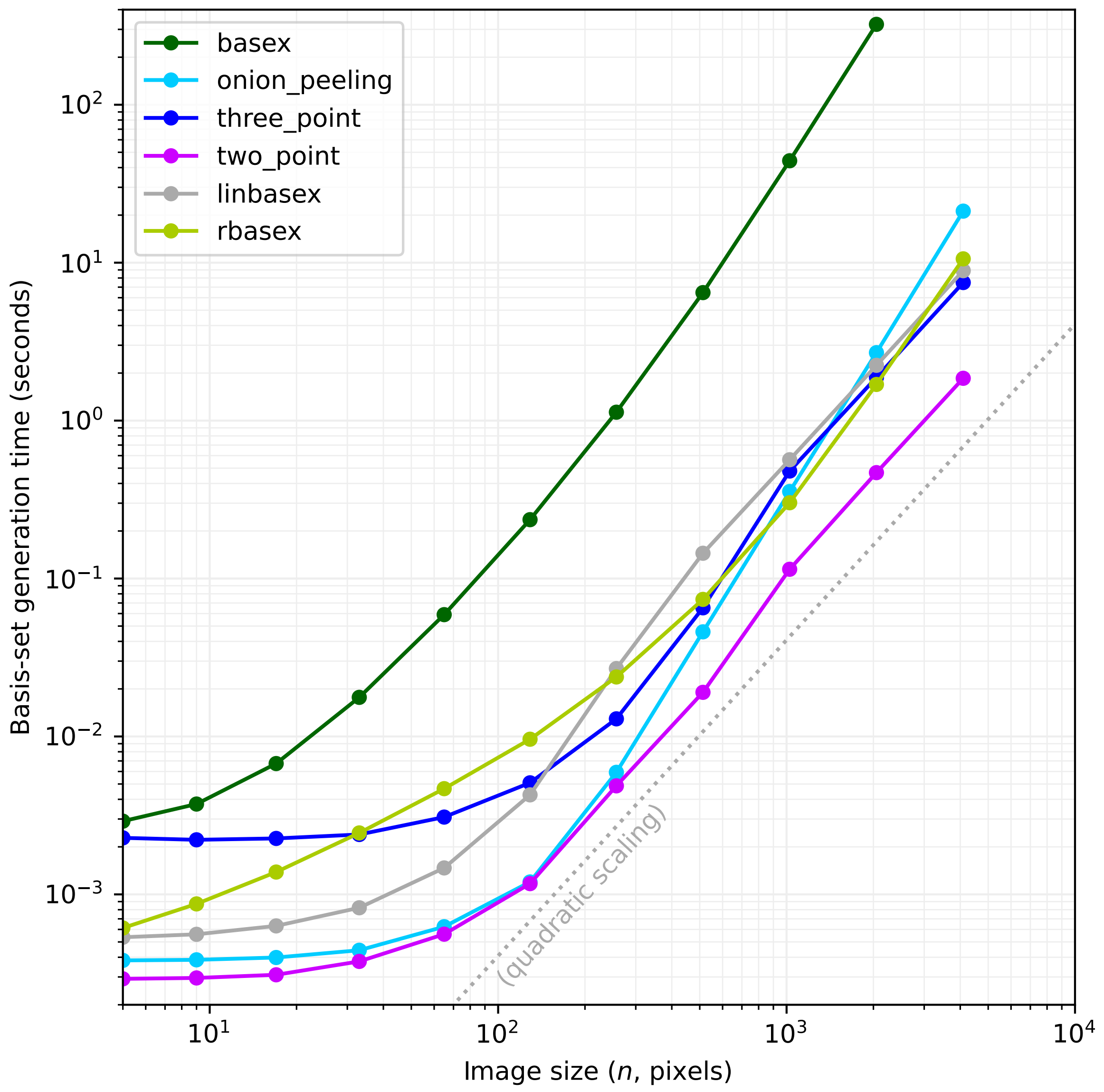
Libraries

- NumPy 1.16.2
- SciPy 1.1.0
- Reference BLAS 3.8.0

Results


basex
basex(var)
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)
(cubic scaling)
Transform time (seconds)
Image size (n, pixels)
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{1}
10^{2}
10^{3}
10^{4}


HD video
Throughput (pixels per second)
Image size (n, pixels)
10^3
10^4
10^5
10^6
10^7
10^1
10^2
10^3
10^4
basex
basex(var)
direct_Python
hansenlaw
onion_bordas
onion_peeling
three_point
two_point
linbasex
rbasex
rbasex(None)



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[1].

[1] V. Dribinski, A. Ossadtchi, V. A. Mandelshtam, H. Reisler, "Reconstruction of Abel-transformable images: The Gaussian basis-set expansion Abel transform method", Rev. Sci. Instrum. 73, 2634–2642 (2002), (PDF).

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[Page 108, 1]). 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* class:

```
abel.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[Page 108, 1] and with other methods implemented in PyAbel.
- Basis sets for any image size are generated automatically.
- Basis functions with any width parameter σ (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 2σ, 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 σ, 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*.

4.2.6 Citation

- V. Dribinski, A. Ossadtchi, V. A. Mandelshtam, H. Reisler, "Reconstruction of Abel-transformable images: The Gaussian basis-set expansion Abel transform method", Rev. Sci. Instrum. 73, 2634–2642 (2002), (PDF).

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

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

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 of "$\sqrt{\pi}$" in calculations of the basis projections χ_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

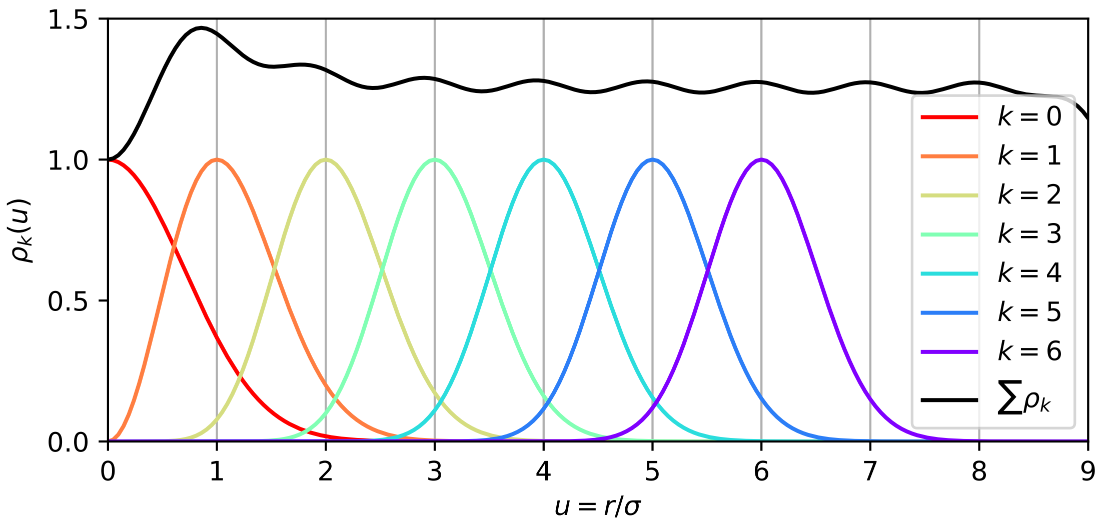
$$\delta > \sqrt{-2\ln\varepsilon_{\text{FP}}}\, u \approx 8.6\, u.$$

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

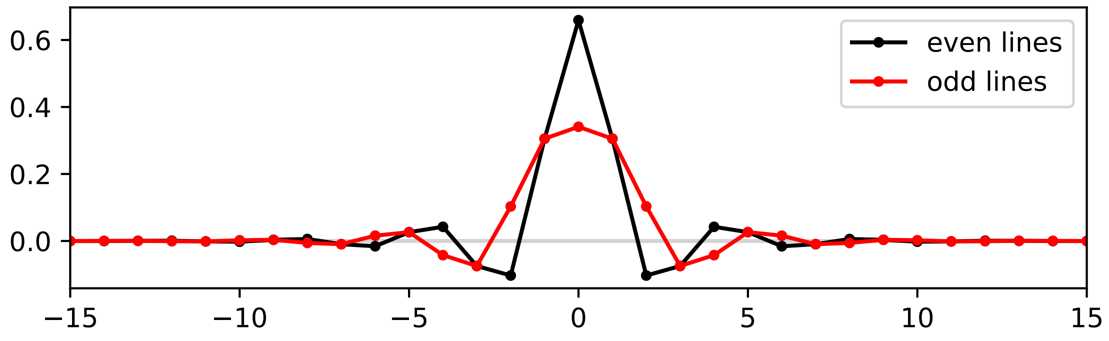
$$\mathbf{BZ} = \mathbf{Z}^\mathrm{T}\left(\mathbf{ZZ}^\mathrm{T}\right)^{-1}\mathbf{Z} = \mathbf{I},$$

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`.

4.3 Daun

4.3.1 Introduction

This suite of methods is based on the deconvolution procedure with Tikhonov regularization described by Daun at al.[1] and extends it with additional, smoother, approximations and regularization types.

4.3.2 How it works

The original method formulates the numerical Abel transform in a form equivalent to the "onion peeling" method, where the original distribution is approximated with a step function (piecewise constant function with 1-pixel-wide radial intervals). The forward transform thus can be described by a system on linear equations in a matrix form

$$\mathbf{A}_{\text{OP}}\mathbf{x} = \mathbf{b}, \tag{4.1}$$

where the vector $\mathbf{x}$ consists of the original distribution values sampled at a uniform radial grid $r_i = i\Delta r$, the vector $\mathbf{b}$ consists of the projection values sampled at the corresponding uniform grid $y_i = i\Delta r$, and the matrix $\mathbf{A}_{\text{OP}}$ corresponds to the "onion peeling" forward Abel transform. Its elements are

$$A_{\text{OP},ij} = \begin{cases} 0, & j < i, \\ 2\Delta r\left[(j+1/2)^2 - i^2\right]^{1/2}, & j = i, \\ 2\Delta r\left(\left[(j+1/2)^2 - i^2\right]^{1/2} - \left[(j-1/2)^2 - i^2\right]^{1/2}\right), & j > i \end{cases}$$

and represent contributions of each distribution interval to each projection interval.

However, instead of performing the inverse transform by using the inverse matrix directly:

$$\mathbf{x} = \mathbf{A}_{\text{OP}}^{-1}\mathbf{b},$$

as it is done in the onion peeling method, the equation (4.1) is solved by applying Tikhonov regularization:

$$\tilde{\mathbf{x}} = \arg\min_{\mathbf{x}} \left(\|\mathbf{A}_{\text{OP}}\mathbf{x} - \mathbf{b}\|^2 + \alpha\|\mathbf{L}\mathbf{x}\|^2\right), \tag{4.2}$$

where α is the regularization parameter, and the finite-difference matrix

$$\mathbf{L} = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix}$$

(approximation of the derivative operator) is used as the Tikhonov matrix. The idea is that the admixture of the derivative norm to the minimization problem leads to a smoother solution. The regularization parameter α controls how much attention is paid to the derivative: when $\alpha = 0$, the exact solution to (4.1) is obtained, even if very noisy; when $\alpha \to \infty$, the solution becomes very smooth, even if reproducing the data poorly. A reasonably chosen value of α can result in a significant suppression of high-frequency noise without noticeably affecting the signal.

The minimization problem (4.2) leads again to a linear matrix equation, and the regularized inverse transform is obtained by using the regularized matrix inverse:

$$\tilde{\mathbf{x}} = \mathbf{A}_{\text{Tik}}^{-1}\mathbf{b}_{\text{Tik}}, \quad A_{\text{Tik}} = (\mathbf{A}^T\mathbf{A} + \alpha\mathbf{A}\mathbf{L}^T\mathbf{L}), \quad \mathbf{b}_{\text{Tik}} = A^T\mathbf{b}. \tag{4.3}$$

(Note: here $\mathbf{x}$ and $\mathbf{b}$ are column vectors, but in PyAbel they are row vectors corresponding to image rows, so all the equations are transposed; moreover, instead of processing row vectors separately, they are transformed as the image matrix at once.)

[1] K. J. Daun, K. A. Thomson, F. Liu, G. J. Smallwood, "Deconvolution of axisymmetric flame properties using Tikhonov regularization", Appl. Opt. 45, 4638–4646 (2006).

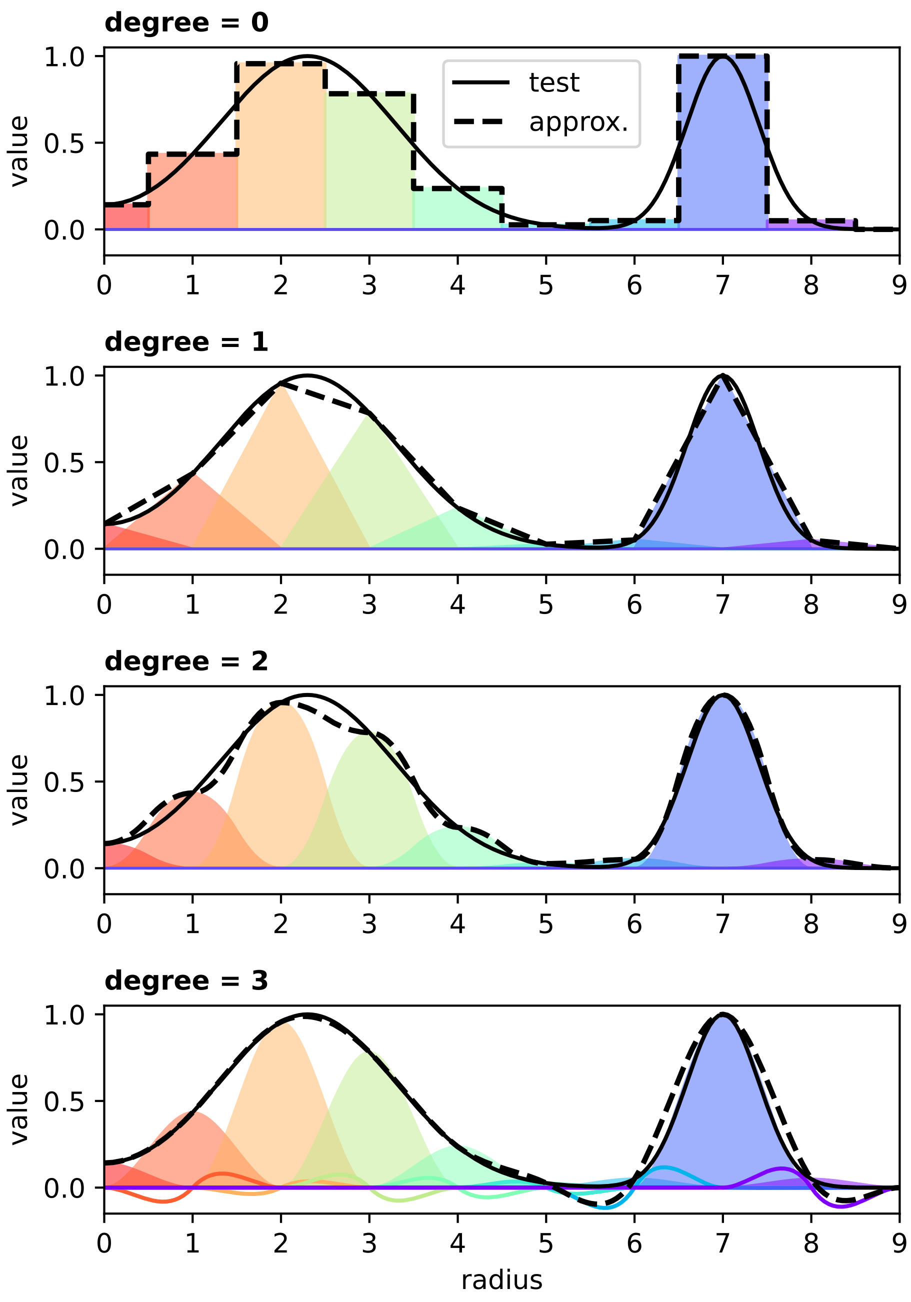
4.3.3 PyAbel additions

Basis sets

The step-function approximation used in the original method implies a basis set consisting of rectangular functions

$$f_i(r) = \begin{cases} 1, & r \in [i - 1/2, i + 1/2], \\ 0 & \text{otherwise.} \end{cases}$$

This approximation can be considered rather coarse, so in addition to these zero-degree piecewise polynomials we also implement basis sets consisting of piecewise polynomials up to 3rd degree. An example of a test function composed of broad and narrow Gaussian peaks and its approximations of various degrees is shown below:



Here the solid black line is the test function, and the dashed black line is its approximation of degree n, equal to the sum of the colored basis functions.

degree = 0:

Rectangular functions produce a stepwise approximation. This is the only approach mentioned in the original article and corresponds to the usual "onion peeling" transform.

degree = 1:

Triangular functions produce a continuous piecewise linear approximation. Besides being continuous (although not smooth), this also corresponds to how numerical data is usually plotted (with points connected by straight lines), so such plots would faithfully convey the underlying method assumptions.

degree = 2:

Piecewise quadratic functions

$$f_i(r) = \begin{cases} 2[r-(i-1)]^2, & r \in [i-1, i-1/2], \\ 1-2[r-i]^2, & r \in [i-1/2, i+1/2], \\ 2[r-(i+1)]^2, & r \in [i+1/2, i+1], \\ 0 & \text{otherwise.} \end{cases}$$

produce a smooth piecewise quadratic approximation. While resembling *BASEX basis functions* in shape, these are localized within ± 1 pixel, sum to unity (although produce oscillations on slopes), and their projections are much faster to compute.

degree = 3:

Combinations of cubic Hermite basis functions produce a cubic-spline approximation (with endpoint derivatives clamped to zero for 2D smoothness). Offers the most accurate representation for sufficiently smooth distributions, but produces ringing artifacts around sharp features, which can result in negative interpolated intensities even for non-negative data points.

(The projections of all these basis functions are calculated as described in *Polynomials*.)

In practice, however, the choice of the basis set has negligible effect on the transform results, as can be seen from an example *below*. Nevertheless, cubic splines might be useful for transforming smooth functions, in which case they yield very accurate results.

Regularization methods

L_2 norm

In addition to the original derivative (difference) Tikhonov regularization, PyAbel also implements the usual L_2 regularization, as in *BASEX*, with the identity matrix $\mathbf{I}$ used instead of $\mathbf{L}$ in (4.3). The results are practically identical to the BASEX method, especially with **degree** = 2, except that the basis set is computed much faster.

Non-negativity

A more substantial addition is the implementation of the non-negativity regularization. Namely, instead of solving the unconstrained quadratic problem (4.2), non-negativity constraints are imposed on the original problem:

$$\tilde{\mathbf{x}} = \arg\min_{\mathbf{x} \geqslant 0} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2.$$

This non-negative least-squares solution yields the distribution without negative intensities that reproduces the input data as well as possible. In situations where the distribution must be non-negative, this is the best physically meaningful solution.

The noise-filtering properties of this method come from the fact that noise in the inverse Abel transform is strongly oscillating, so if negative-going spikes are forbidden in the solution, the positive-going spikes must also be reduced in order to preserve the overall intensity. Thus the method is most beneficial for very noisy images, for which linear methods produce a large amount of noise reaching negative values. For clean images of non-negative distributions, the constrained solution exactly matches the solution of the original problem (4.1). And unlike Tikhonov regularization, it does not blur legitimate sharp features in any case.

Notice that constrained quadratic minimization remains a *non-linear* problem. This has several important implications. First, it is much more computationally challenging, so that transforming a megapixel image takes many seconds instead of several milliseconds (and depends on the image itself). Second, the average of transformed images is generally not equal to the transform of the averaged image. It is thus recommended to perform as much averaging (image symmetrization and summation of multiple images if applicable) as possible before applying the transform. In particular, using `symmetry_axis=(0, 1)` in *abel.transform.Transform* would in fact require transforming only one quadrant, which is 4 times faster that transforming the whole image. Third, the method is only *asymptotically* unbiased, but for sufficiently noisy data can systematically shift or blur the intensities towards the symmetry axis. Thus while this regularization can be very helpful in revealing the structure in raw images, it is not recommended when further processing (like model fitting) is involved. In particular, for extraction of velocity and anisotropy distributions from velocity-map images, the *rBasex* method, possibly with its "positive" regularization, is more appropriate.

4.3.4 When to use it

This method with default parameters (0th degree, 0 regularization parameter) is identical to the *"onion peeling"* method, but can also be used for the forward transform.

The original (derivative/difference) Tikhonov regularization with non-zero regularization parameter helps to remove high-frequency oscillations from the transformed image. However, an excessively large regularization parameter can lead to oversmoothing and broadening of the useful signal and under/overshoots around sharp features. As recommended by Daun et al., by systematically adjusting the heuristic regularization parameter, the analyst can find a solution that represents an acceptable compromise between accuracy and regularity.

The L_2 Tikhonov regularization approach is equivalent to that in the *BASEX* method and has the same use cases and [dis]advantages.

The non-negativity regularization is recommended for visual inspection of very noisy images and images with sharp features without a broad background. However, due to its slowness, it cannot be used for real-time data processing.

4.3.5 How to use it

The inverse Abel transform of a full image can be done with the *abel.Transform* class:

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

For the forward Abel transform, simply add `direction='forward'`:

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

Additional parameters can be passed through the `transform_options` parameter. For example, to use the original regularization method with the regularization parameter set to 100:

```
abel.Transform(myImage, method='daun',
               transform_options=dict{reg=100}).transform
```

The L_2 regularization can be applied using

```
abel.Transform(myImage, method='daun',
               transform_options=dict{reg=('L2', 100)}).transform
```

And the non-negative solution is obtained by

```
abel.Transform(myImage, method='daun',
               transform_options=dict{reg='nonneg'}).transform
```

In this case, it is recommended to use symmetrization:

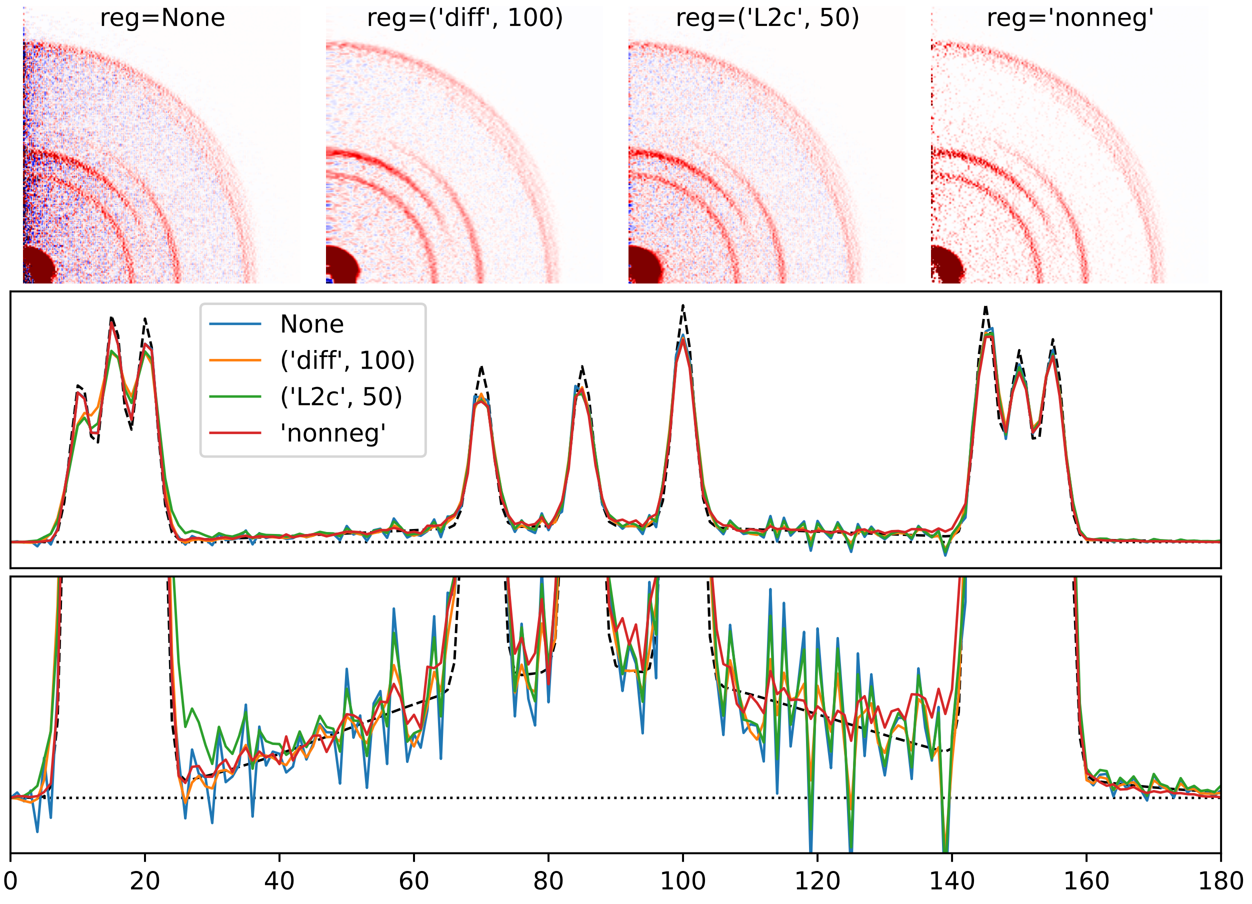
```
abel.Transform(myImage, method='daun',
               symmetry_axis=0,  # or symmetry_axis=(0, 1) if applicable
               transform_options=dict{reg='nonneg'}).transform
```

unless independent inspection of all image parts is desired.

The algorithm can be also accessed directly (to transform a right-side half-image or properly oriented quadrants) through the *abel.daun.daun_transform()* function.

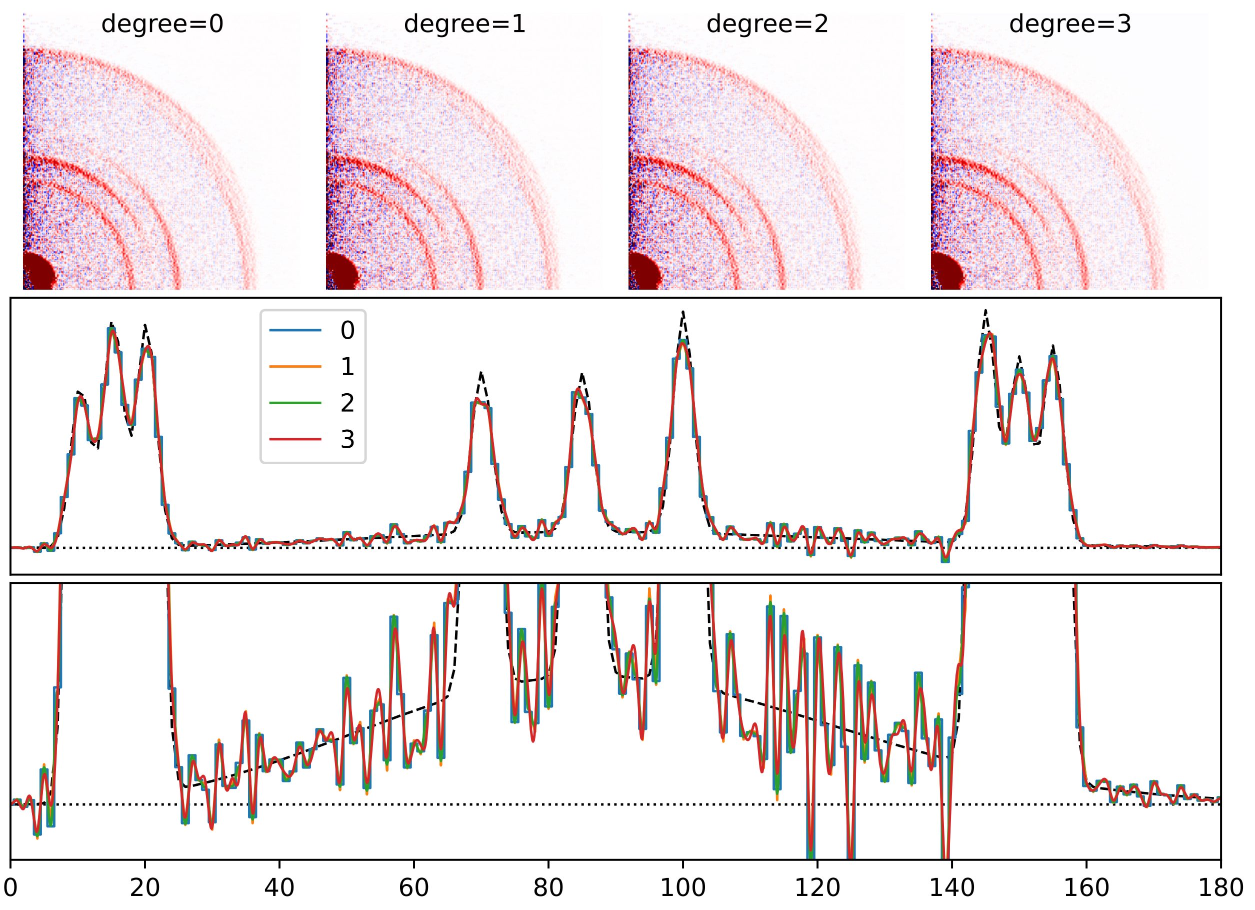
4.3.6 Examples

Performance of various regularization methods for the Dribinski sample image with added Poissonian noise:



(source code)

The degree of basis-set polynomials has almost no effect on the results (shown here for `reg=0`):



(source code)

4.3.7 Citation

- K. J. Daun, K. A. Thomson, F. Liu, G. J. Smallwood, "Deconvolution of axisymmetric flame properties using Tikhonov regularization", Appl. Opt. 45, 4638–4646 (2006).

Note: If you use any non-default options (degree, regularization), please cite not only the article by Daun et al. and the PyAbel article, but also *this PyAbel release* (DOI: 10.5281/zenodo.7438595), because these capabilities are not present in the original work by Daun et al. and were added to PyAbel after the RSI publication.

4.4 Direct

4.4.1 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.

4.4.2 How it works

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

4.4.3 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.

4.4.4 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()*.

4.5 Hansen–Law

4.5.1 Introduction

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

In their words, Hansen and Law[Page 122, 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.

4.5.2 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

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

[1] E. W. Hansen, P.-L. Law, "Recursive methods for computing the Abel transform and its inverse", J. Opt. Soc. Am. A 2, 510–520 (1985).
[2] E. W. Hansen, "Fast Hankel transform algorithm", IEEE Trans. Acoust. Speech Signal Proc. 33, 666–671 (1985)

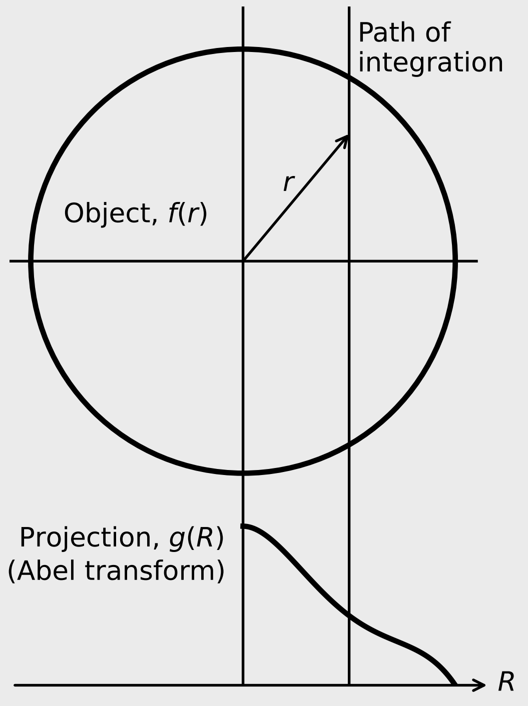


Fig. 4.9: Projection geometry (Fig. 1[1])

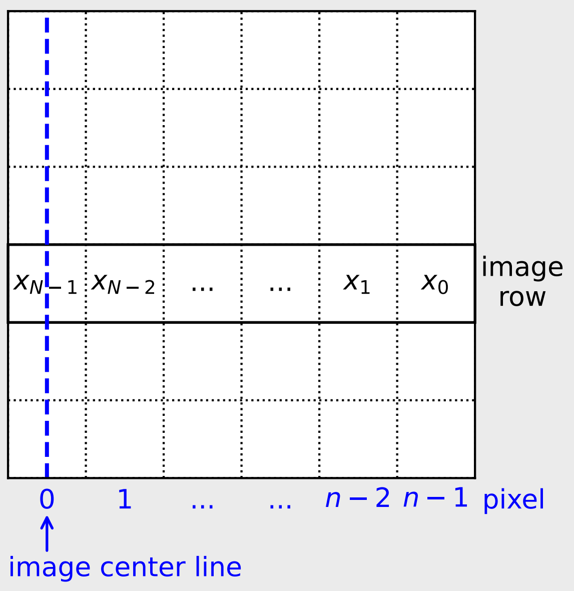


Fig. 4.10: Recursion: pixel value from adjacent outer-pixel

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 *below*.

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 Φ, B_{0n}, and B_{1n} are given *below*.

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.

4.5.3 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.

4.5.4 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
```

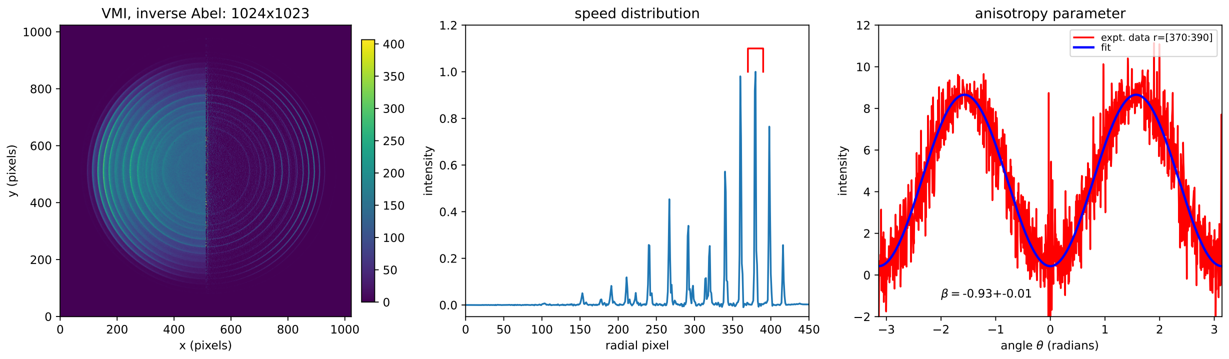
If you would like to access the Hansen-Law algorithm directly (to transform a right-side half-image), you can use *`abel.hansenlaw.hansenlaw_transform()`*.

4.5.5 Tips

hansenlaw tends to perform better with images of large size $n > 1001$ pixel width. For smaller images the angular_integration (speed) profile may look better if sub-pixel sampling is used:

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

4.5.6 Example



Source code

4.5.7 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 guidance, algebra, and explanations, to aid the implementation of his first-order hold algorithm, described in Ref.[Page 122, 2] (April 2018).

4.5.8 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,

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

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

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 λ_k. For the inverse transform, there is an additional factor $\frac{1}{\pi r}$ in the state equation, and hence the integrand has λ_k power, reduced by -1. While, for the first-order hold approximation, the linear ϵ term increases λ_k by +1.

4.5.9 Citation

- E. W. Hansen, P.-L. Law, "Recursive methods for computing the Abel transform and its inverse", J. Opt. Soc. Am. A 2, 510–520 (1985).
- E. W. Hansen, "Fast Hankel transform algorithm", IEEE Trans. Acoust. Speech Signal Proc. 33, 666–671 (1985)
- J. R. Gascooke, PhD Thesis: "Energy Transfer in Polyatomic-Rare Gas Collisions and Van Der Waals Molecule Dissociation", Flinders University (2000), (record, PDF).

4.6 Lin-Basex

4.6.1 Introduction

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

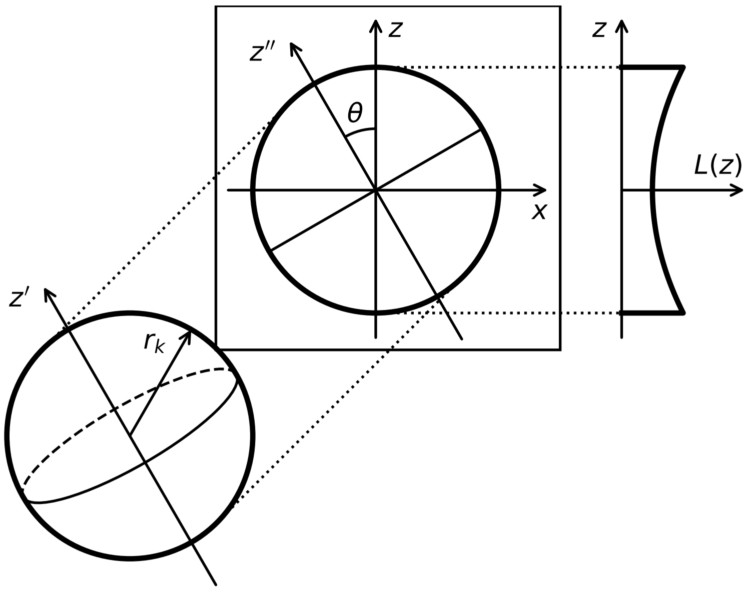
[*from the abstract*]

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.

[1] Th. Gerber, Yu. Liu, G. Knopp, P. Hemberger, A. Bodi, P. Radi, Ya. Sych, "Charged particle velocity map image reconstruction with one-dimensional projections of spherical functions", Rev. Sci. Instrum. 84, 033101 (2013).

4.6.2 How it works

A projection of 3D Newton spheres onto the detector plane followed by a projection of the resulting 2D image 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 Legendre polynomials $P_\ell(z/r_k)$. There exists for a given base a unique set of coefficients $p_{\ell k}$ producing a least-squares fit to the function $L(z,u)$.

[*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".

4.6.3 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 $1/\sqrt{\text{counts}}$.

4.6.4 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.

4.6.5 Tips

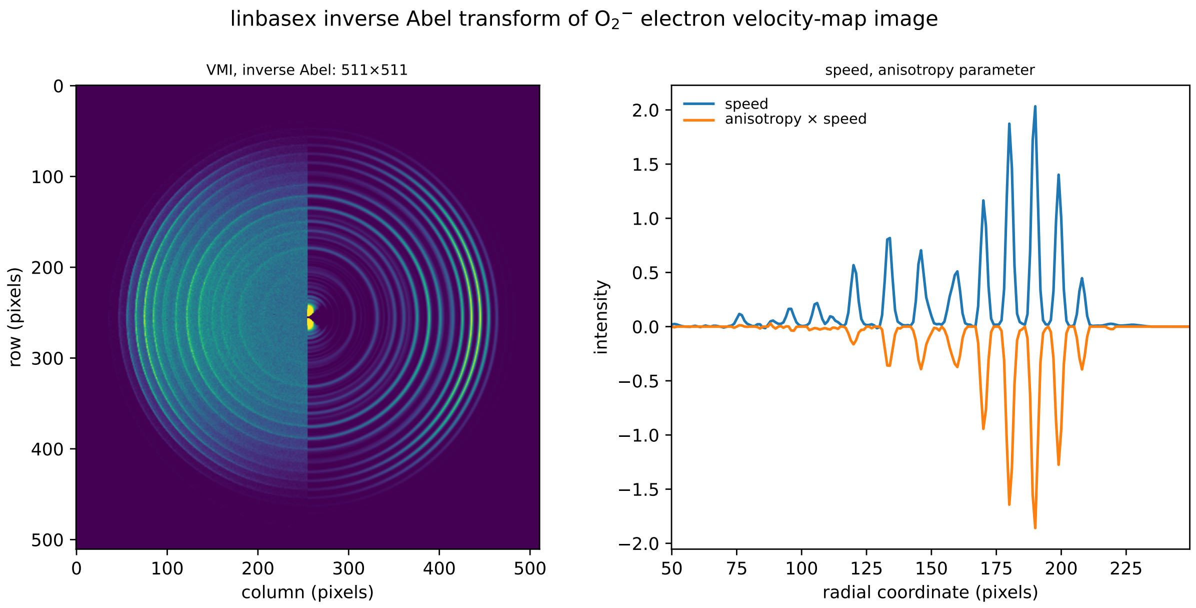
Including more projection angles may improve the transform:

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

or

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

4.6.6 Example



Source code

4.6.7 Historical

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

4.6.8 Citation

- Th. Gerber, Yu. Liu, G. Knopp, P. Hemberger, A. Bodi, P. Radi, Ya. Sych, "Charged particle velocity map image reconstruction with one-dimensional projections of spherical functions", Rev. Sci. Instrum. 84, 033101 (2013).

4.7 Onion Peeling (Bordas)

4.7.1 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[1]. The algorithm actually originates from Bordas et al.[2].

See the discussion in issue #56.

[1] C. E. Rallis, T. G. Burwitz, P. R. Andrews, M. Zohrabi, R. Averin, S. De, B. Bergues, B. Jochim, A. V. Voznyuk, N. Gregerson, B. Gaire, I. Znakovskaya, J. McKenna, K. D. Carnes, M. F. Kling, I. Ben-Itzhak, E. Wells, "Incorporating real time velocity map image reconstruction into closed-loop coherent control", Rev. Sci. Instrum. 85, 113105 (2014).

[2] C. Bordas, F. Paulig, "Photoelectron imaging spectrometry: Principle and inversion method", Rev. Sci. Instrum. 67, 2257–2268 (1996).

4.7.2 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.

4.7.3 When to use it

This is a historical implementation of the onion-peeling method.

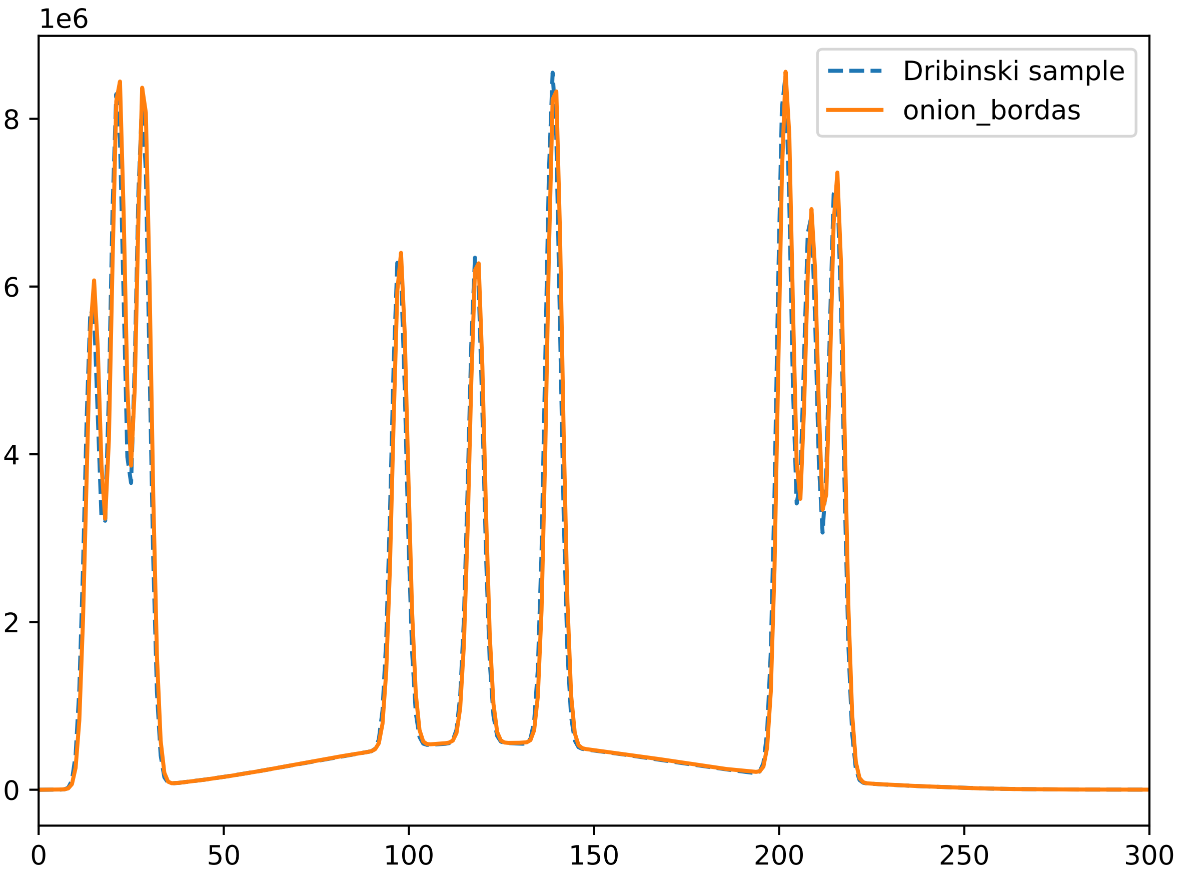
4.7.4 How to use it

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()`*.

4.7.5 Example



Source code

4.7.6 Citation

- C. E. Rallis, T. G. Burwitz, P. R. Andrews, M. Zohrabi, R. Averin, S. De, B. Bergues, B. Jochim, A. V. Voznyuk, N. Gregerson, B. Gaire, I. Znakovskaya, J. McKenna, K. D. Carnes, M. F. Kling, I. Ben-Itzhak, E. Wells, "Incorporating real time velocity map image reconstruction into closed-loop coherent control", Rev. Sci. Instrum. 85, 113105 (2014).
- C. Bordas, F. Paulig, "Photoelectron imaging spectrometry: Principle and inversion method", Rev. Sci. Instrum. 67, 2257–2268 (1996).

4.8 Onion Peeling (Dasch)

4.8.1 Introduction

The "Dasch onion peeling" deconvolution algorithm is one of several described in the Dasch paper[1]. 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

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

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

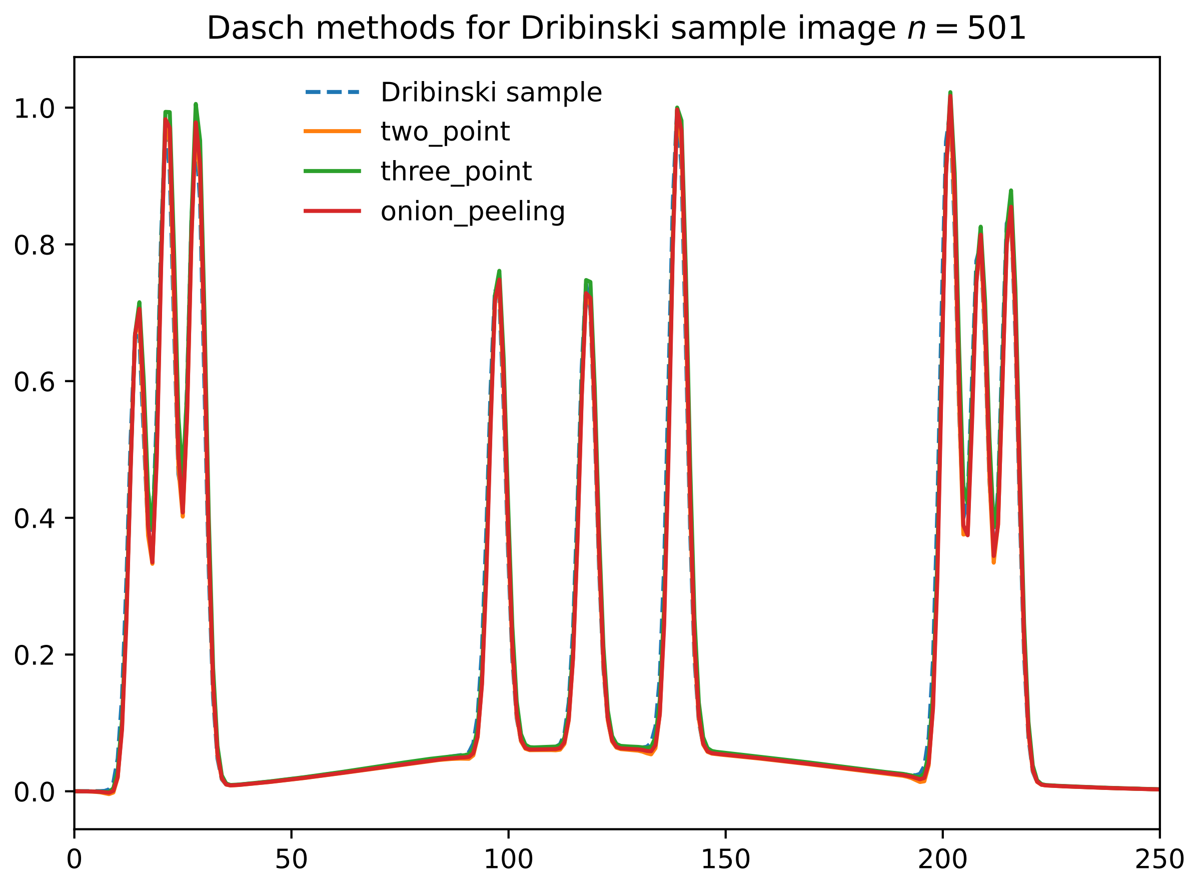
```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

"""example_dasch_methods.py.
"""

import numpy as np
```

(continues on next page)

[1] C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

(continued from previous page)

```python
import abel
import matplotlib.pyplot as plt

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

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

# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration_3D(origQ[0], origin=(-1, 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)
# speed distribution
    speed = abel.tools.vmi.angular_integration_3D(AQ0, origin=(-1, 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.xlim((0, 250))
plt.legend(loc='upper center', bbox_to_anchor=(0.35, 1), frameon=False)
plt.tight_layout()
# plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```



For more information on the PyAbel implementation of the `onion_peeling` algorithm, please see PR #155.

4.8.6 Citation

- C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

4.9 rBasex

4.9.1 Introduction

This method resembles the pBasex[1] approach of expanding a velocity-map image over a 2D basis set in polar coordinates, but uses more convenient basis functions with analytical Abel transforms, developed by M. Ryazanov[2].

[1] G. A. Garcia, L. Nahon, I. Powis, "Two-dimensional charged particle image inversion using a polar basis function expansion", Rev. Sci. Instrum. 75, 4989–4996 (2004).

[2] M. Ryazanov, "Development and implementation of methods for sliced velocity map imaging. Studies of overtone-induced dissociation and isomerization dynamics of hydroxymethyl radical (CH_2OH and CD_2OH)", Ph.D. dissertation, University of Southern California, 2012. (ProQuest, USC).

4.9.2 How it works

In velocity-map imaging (VMI) with cylindrically symmetric photodissociation (in a broad sense, including photoionization and photodetachment) the 3D velocity distribution at each speed (3D radius) consists of a finite number of spherical harmonics $Y_{nm}(\theta, \varphi)$ with $m = 0$, which are also representable as Legendre polynomials $P_n(\cos\theta)$. This means that an $N \times N$ image has only $N_r \times N_a$ degrees of freedom, where N_r is the number of radial samples, usually $N/2$, and N_a is the number of angular terms, a small number depending on the studied process. These degrees of freedom correspond to the "radial distribution" extracted from the transformed image in other, general Abel-inversion methods.

However, if these radial distributions are considered as a basis, the 3D distribution can be represented as a linear combination of these basis functions with some coefficients. And the corresponding image, being the forward Abel transform of the 3D distribution, will be represented as a linear combination of basis-function projections, that is, their forward Abel transforms, with the same coefficients. The reverse is also true: finding the expansion coefficients of an experimental velocity-map image over the projected basis directly gives the expansion coefficients of the initial 3D velocity direction and thus the sought radial distributions.

Finding the expansion coefficients is a simple linear problem, and the forward Abel transforms of the basis functions can be calculated easily if the basis is chosen wisely.

See *rBasex: mathematical details* for the complete description.

Differences from pBasex

While rBasex is similar to pBasex in the idea of using VMI-oriented 3D basis functions, it has several key differences:

1. Triangular radial basis functions are used instead of Gaussians. They are more compact/orthogonal (only the adjacent functions overlap) and have analytical Abel transforms.
2. Cosine powers are used instead of Legendre polynomials for angular basis functions. This makes the projected basis functions also separable into radial and angular parts.
3. The basis separability allows decomposition of the problem in two steps: first, radial distributions are extracted from the image (without intermediate rebinning to polar grid, thus faster and avoiding accumulation of resampling errors); second, these radial distributions are expanded over radial bases for each angular order. This eliminates the necessity to work with large matrices.
4. Custom pixel weighting can be used, for example, to exclude image areas "damaged" in some way (obscured by a beam block, contaminated by parasitic signals, affected by detector imperfections and so on). Partial images (not including the whole angular range) can be reconstructed as well.
5. The forward Abel transform is implemented in addition to the inverse transform.
6. Additional (better) regularization methods are implemented.

Differences from the reconstruction method described in[Page 134, 2]

Many ideas used in rBasex, including the analytically transformable basis functions, are taken from the previous work[Page 134, 2], but with some omissions, additions and modifications.

1. Instead of working with individual pixels and weighting them according to Poisson statistics, the binned radial distributions (not weighted by default) are transformed. This is less accurate, but much faster, especially in Python.
2. Slicing is not implemented.
3. Only the non-negativity constraints are implemented. However, several linear regularization options are added.
4. Odd angular orders can be included.

4.9.3 When to use it

This method makes additional assumptions (beyond cylindrical symmetry) about the data, so it can be applied only to velocity-map images or in other similar situations involving a finite number of spherical harmonics. However, in this special case, it offers several benefits:

1. The reconstructed radial distributions, which are often the primary interest in VMI studies, are obtained directly.
2. Limitations on the angular behavior of the distribution also put strong constraints on the reconstruction noise, making the reconstructed images much cleaner.
3. Several optional *regularization* methods help to further reduce noise in reconstructed images, especially near the center. Regularization strengths can be adjusted to produce a desirable balance between noise reduction and blurring of sharp features.
4. Unlike general Abel-transform methods, which have time complexity with *cubic* dependence on the image size, this method is only *quadratic*, once the transform matrix is computed. Computing the transform matrix is still cubic, but after it is done, transforming a series of images is faster, especially for large images.
5. The optional non-negativity constraints implemented in this method allow obtaining physically meaningful intensity and anisotropy distributions. They can also help in denoising experimental images with very low event counts.

4.9.4 How to use it

The method can be accessed through the universal `abel.Transform` class:

```
res = abel.Transform(image, method='rbasex')
recon = res.transform
distr = res.distr
```

optionally using other `Transform` arguments and passing additional rBasex parameters (see `abel.rbasex.rbasex_transform()` documentation for their full description) through the `transform_options` argument. Alternatively, it might be more convenient to use the method by calling its transform function directly:

```
recon, distr = abel.rbasex.rbasex_transform(image)
r, I, beta = distr.rIbeta()
```

It returns the transformed image `recon` and a `Distributions.Results` object `distr`, from which various radial distributions can be retrieved, such as the intensity and anisotropy-parameter distributions in this example.

If only the distributions are needed, but not the transformed image itself, the calculations can be accelerated by disabling the creation of the output image:

```
_, distr = abel.rbasex.rbasex_transform(image, out=None)
r, I, beta = distr.rIbeta()
```

Note that rBasex does not require the input image to be centered. Thus instead of centering it with `center_image()` (or using the `origin` argument of `Transform`), which will crop some data or fill it with zeros, it is better to pass the image origin directly to the transform function, determining it automatically, if needed:

```
origin = abel.tools.center.find_origin(image, method='convolution')
recon, distr = abel.rbasex.rbasex_transform(image, origin=origin)
```

This also *must* be done if optional pixel weighting is used, since otherwise the centered image would become inconsistent with the weights array. For example, when using the `Transform` class, pass the origin as follows:

```
res = abel.Transform(image, method='rbasex',
                     transform_options=dict(origin=..., weights=...))
```

The weights array can also be used as a mask, using zero weights to exclude unwanted pixels, as demonstrated in *Example: rBasex beam block*. In practice, instead of defining the mask geometry in the code, it might be more convenient to save the analyzed data as an image file:

```
# save as an RGB image using a chosen colormap
plt.imsave('imagemask.png', image, cmap='hot')
```

then open it in any raster graphics editor, paint the areas to be excluded with some distinct color (for example, blue in case of `cmap='hot'`) and save it. This painted image then can be loaded in the program, and the mask is easily extracted from it:

```
# read as an array with R, G, B (or R, G, B, A) components
mask = plt.imread('imagemask.png')
# set zero weights for pixels with blue channel (2) > red channel (0)
# and unit weights for other pixels
weights = 1.0 - (mask[..., 2] > mask[..., 0])
```

(for other image colormaps and mask colors, adapt the comparison logic accordingly). These weights then can be used in the transform of the original data, as well as any other data having the same mask geometry.

4.9.5 Citation

This method has not yet been published elsewhere, so please cite it as the "rBasex method from the PyAbel package", using the current Zenodo DOI: 10.5281/zenodo.7438595.

- G. A. Garcia, L. Nahon, I. Powis, "Two-dimensional charged particle image inversion using a polar basis function expansion", Rev. Sci. Instrum. 75, 4989–4996 (2004).
- M. Ryazanov, "Development and implementation of methods for sliced velocity map imaging. Studies of overtone-induced dissociation and isomerization dynamics of hydroxymethyl radical (CH_2OH and CD_2OH)", Ph.D. dissertation, University of Southern California, 2012. (ProQuest, USC).

rBasex: mathematical details

Coordinates

The coordinate systems used here are defined such that the image is in the xy plane, with the y axis being the (vertical) axis of symmetry. Thus the image-space polar coordinates are

$$r = \sqrt{x^2 + y^2},$$
$$\theta = \arctan \frac{y}{x},$$

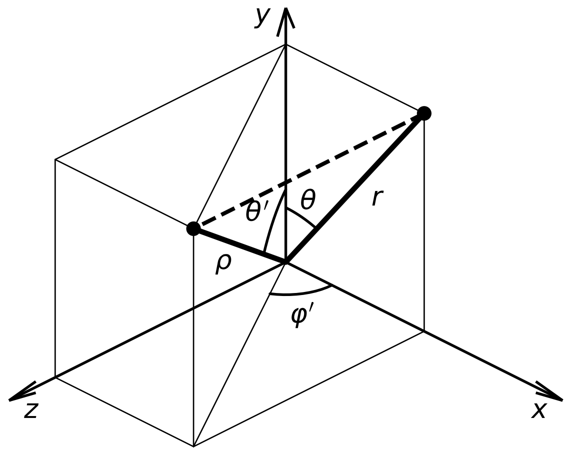
with the polar angle θ measured from the y axis. The corresponding coordinate system for the underlying 3D distribution has the same x and y axes, plus a perpendicular z axis, so the distribution-space spherical coordinates are

$$\rho = \sqrt{x^2 + y^2 + z^2},$$
$$\theta' = \arctan \frac{y}{\sqrt{x^2 + z^2}},$$
$$\phi' = \arctan \frac{z}{x},$$

with the polar angle θ also measured from the y axis (the symmetry axis). The Abel transform performs a projection along the z axis:

$$(x, y, z) \mapsto (x, y),$$

as shown by the dashed line:



This figure also illustrates important relations between the 3D and 2D radii:

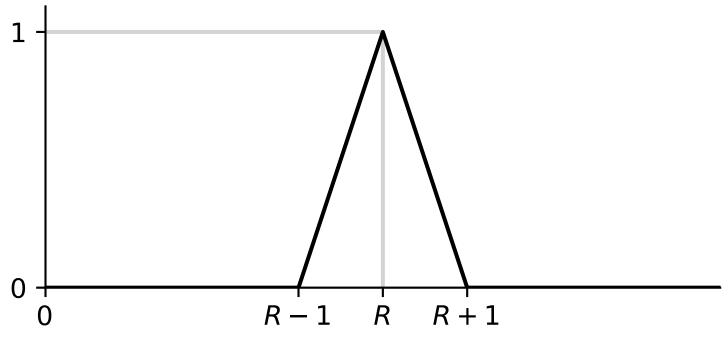
$$\rho = \sqrt{r^2 + z^2}$$

and polar angles:

$$\left.\begin{aligned}\cos\theta &= y/r\\ \cos\theta' &= y/\rho\end{aligned}\right\} \quad\Rightarrow\quad \cos\theta' = \frac{r}{\rho}\cos\theta.$$

Basis functions

The 3D distribution basis consists of direct products of radial and angular basis functions. The radial functions are triangular functions centered at integer radii (whole pixels), spanning ±1 pixel:



$$b_R(\rho) = \begin{cases} \rho - (R-1), & R-1 < \rho < R, \\ (R+1) - \rho, & R \leqslant \rho < R+1, \\ 0 & \text{otherwise} \end{cases}$$

(and $b_0(\rho)$ does not have the inner part, $R-1 < \rho < R$, since $\rho \geqslant 0$). These functions form a basis of continuous piecewise linear approximations with nodes at each pixel. In other words, linear combinations of these functions represent any radial distribution as pixel values connected by straight lines.

The angular basis functions are just integer powers of $\cos\theta'$ from 0 up to the highest order expected in the distribution. Hence the overall 3D distribution basis functions are

$$b_{R,n}(\rho, \theta', \varphi') = b_R(\rho) \cos^n \theta'$$

(due to cylindrical symmetry, there is no dependence on the azimuthal angle φ').

The 2D image basis functions are, correspondingly, the projections of these distribution basis functions along the z axis:

$$p_{R,n}(r, \theta) = \int_{-\infty}^{\infty} b_{R,n}(\rho, \theta', \varphi')\, dz = 2\int_0^{\infty} b_{R,n}(\rho, \theta', \varphi')\, dz.$$

Basis projections

As mentioned earlier, $\cos\theta' = \frac{r}{\rho}\cos\theta$, so we can write

$$\begin{aligned} p_{R,n}(r, \theta) = 2\int_0^{\infty} b_R(\rho) \cos^n \theta'\, dz = \\ = 2\int_0^{\infty} b_R(\rho) \left(\frac{r}{\rho}\right)^n \cos^n \theta\, dz = \\ = 2\int_0^{\infty} b_R(\rho) \left(\frac{r}{\rho}\right)^n dz \cdot \cos^n \theta. \end{aligned}$$

In other words, basis projection are also separable into radial and angular parts:

$$p_{R,n}(r, \theta) = p_{r;n}(r) \cos^n \theta,$$

with the same angular dependence, but their radial parts are different for different angular orders (thus projections of functions with angular dependence different from a singe cosine power, for example, $\sin^2\theta'$ or Legendre polynomials, would be *not* separable).

Since $b_R(\rho)$ consists of two segments that are linear functions of ρ, the integrals above can be expressed in terms of the integrals

$$\int\limits_{z_{\text{min}}}^{z_{\text{max}}} \rho \left(\frac{r}{\rho}\right)^n dz = r \int\limits_{z_{\text{min}}}^{z_{\text{max}}} \left(\frac{r}{\rho}\right)^{n-1} dz$$

and

$$\int\limits_{z_{\text{min}}}^{z_{\text{max}}} R \left(\frac{r}{\rho}\right)^n dz = R \int\limits_{z_{\text{min}}}^{z_{\text{max}}} \left(\frac{r}{\rho}\right)^n dz$$

with appropriate lower and upper limits. That is, only the antiderivatives of the form

$$F_n(r, z) = \int \left(\frac{r}{\rho}\right)^n dz$$

with integer n from -1 to the highest angular order are needed. They all can be computed analytically and are listed in the following table (as a reminder, $\rho = \sqrt{r^2 + z^2}$):

n	$F_n(r, z)$
-1	$\frac{1}{2}z\left(\frac{r}{\rho}\right)^{-1} + \frac{1}{2}r\ln(z+\rho)$
0	z
1	$r\ln(z+\rho)$
2	$r\arctan\frac{z}{r}$
3	$z\left(\frac{r}{\rho}\right)$
4	$\frac{1}{2}z\left(\frac{r}{\rho}\right)^2 + \frac{1}{2}r\arctan\frac{z}{r}$
5	$\frac{1}{3}z\left(\frac{r}{\rho}\right)^3 + \frac{2}{3}z\left(\frac{r}{\rho}\right)$
6	$\frac{1}{4}z\left(\frac{r}{\rho}\right)^4 + \frac{3}{8}z\left(\frac{r}{\rho}\right)^2 + \frac{3}{8}r\arctan\frac{z}{r}$
$\vdots$	$\vdots$
$2m \geqslant 2$	$z\sum\limits_{k=1}^{m-1} a_k\left(\frac{r}{\rho}\right)^{2k} + a_1 r\arctan\frac{z}{r}, \quad a_k = \dfrac{\prod_{l=k+1}^{m-1}(2l-1)}{\prod_{l=k}^{m-1}(2l)}$
$2m+1 \geqslant 3$	$z\sum\limits_{k=0}^{m-1} a_k\left(\frac{r}{\rho}\right)^{2k+1}, \quad a_k = \dfrac{\prod_{l=k+1}^{m-1}(2l)}{\prod_{l=k}^{m-1}(2l+1)}$

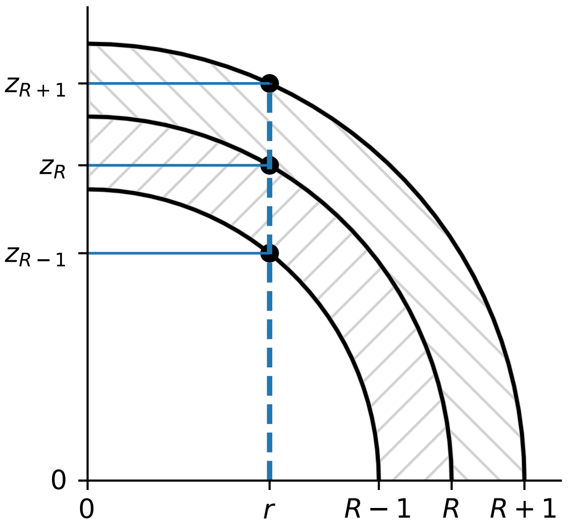
(The general expression assume the usual convention that an empty product equals 1, and an empty sum equals 0.) A simple recurrence relation exists for $n \neq 0$:

$$F_{n+2}(r, z) = \frac{1}{n}z\left(\frac{r}{\rho}\right)^n + \frac{n-1}{n}F_n(r, z).$$

The integration limits have the form

$$z_R = \begin{cases} \sqrt{R^2 - r^2}, & r < R, \\ 0 & \text{otherwise} \end{cases}$$

and are $[z_{R-1}, z_R]$ for the inner part $b_R\big(\rho \in [R-1, R]\big) = \rho - (R-1)$ and $[z_R, z_{R+1}]$ for the outer part $b_R\big(\rho \in [R, R+1]\big) = (R+1) - \rho$:



The ρ values corresponding to the integration limits (for substitution in the antiderivatives F_n) have an even simpler form:

$$\rho|_{z=z_R} = \sqrt{r^2 + z_R^2} = \max(r, R),$$

and hence

$$\left.\left(\frac{r}{\rho}\right)\right|_{z=z_R} = \min\left(\frac{r}{R}, 1\right).$$

The $\arctan\frac{z_R}{r}$ terms can also be "simplified" to $\arccos\left.\frac{r}{\rho}\right|_{z=z_R} = \arccos\frac{r}{R}$ for $r < R$ and 0 otherwise, or $\arccos\left[\min\left(\frac{r}{R}, 1\right)\right]$. This seems to be more computationally efficient on modern systems, although previously it was the other way around, since $\arccos$ was implemented in libraries through $\arctan 2$ (FPATAN), square root (FSQRT) and arithmetic operations.

Collecting all the pieces together, we get the following expression for the radial parts of the projections:

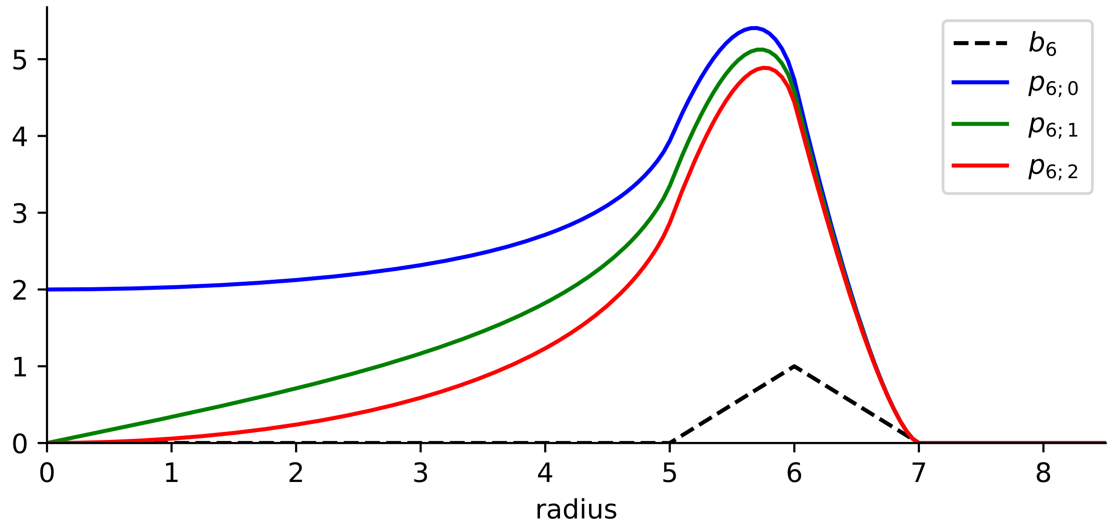
$$\begin{aligned} p_{R;n}(r) = 4[rF_{n-1}(r, z_R) - RF_n(r, z_R)] - \\ - 2[rF_{n-1}(r, z_{R-1}) - (R-1)F_n(r, z_{R-1})] - \\ - 2[rF_{n-1}(r, z_{R+1}) - (R+1)F_n(r, z_{R+1})]. \end{aligned}$$

Like $b_0(\rho)$, the $p_{0;n}(r)$ functions do not have the inner part, so for them ($R = 0$, $z_R = 0$, $R + 1 = 1$) the expression is

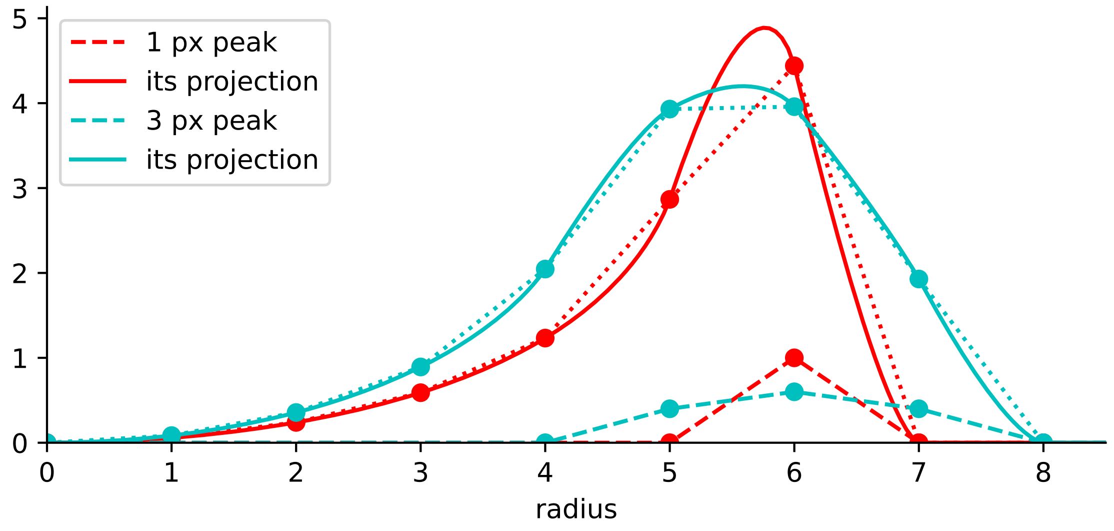
$$\begin{aligned} p_{0;n}(r) &= 2[rF_{n-1}(r, 0) - F_n(r, 0)] - 2[rF_{n-1}(r, z_1) - F_n(r, z_1)] = \\ &= 2[F_n(r, z_1) - F_n(r, 0)] - 2r[F_{n-1}(r, z_1) - F_{n-1}(r, 0)]. \end{aligned}$$

However, in practice $R = 0$ corresponds to the single central pixel, and at the integer grid we have $p_{0;0}(r) = \delta_{r,0}$ and $p_{0;n>0}(r) = 0$, that is the intensity at $r = 0$ must be assumed isotropic.

Here are examples of $p_{R;n}(r)$ plotted for $R = 6$ and $n = 0, 1, 2$, together with the radial part of the distribution basis function $b_R(r)$:



The projection functions have a large curvature near $r \approx R$ and thus are not well represented by piecewise linear approximations at the integer grid, as illustrated below (the solid red line is the same $p_{6;2}(r)$ as above):



This was not a problem for the reconstruction method developed in[1], since it samples these functions at each pixel, with their $r = \sqrt{x^2 + y^2}$ values not limited to integers. But expanding piecewise linear radial distributions over the basis of these curved $p_{R;n}$ might be problematic. However, as the cyan curves illustrate, even for a peak with just 3 nonzero points, its projection is represented by linear segments significantly better. Therefore, for real experimental data with adequate sampling (peak widths > 2 pixels), the piecewise linear approximation should work reasonably well.

[1] M. Ryazanov, "Development and implementation of methods for sliced velocity map imaging. Studies of overtone-induced dissociation and isomerization dynamics of hydroxymethyl radical (CH_2OH and CD_2OH)", Ph.D. dissertation, University of Southern California, 2012. (ProQuest, USC).

Transform

The initial 3D distribution has the form

$$I(\rho, \theta') = \sum_n I_n(\rho) \cos^n \theta',$$

where $I_n(\rho)$ are the radial distributions for each angular order. They are represented as a linear combination of the radial basis functions:

$$I_n(\rho) = \sum_R c_{R,n} b_R(\rho).$$

The forward Abel transform of this 3D distribution (in other words, its projection, or the experimentally recorded image) then has the form

$$P(r, \theta) = \sum_n P_n(\rho) \cos^n \theta,$$

where $P_n(r)$ are its radial distributions for each angular order (not to be confused with Legendre polynomials) and are represented as linear combinations of the radial projected basis functions:

$$P_n(r) = \sum_R c_{R,n} p_{R;n}(r)$$

with the same coefficients $c_{R,n}$.

If the radial distributions of both the initial distribution and its projection are sampled at integer radii, these linear combinations can be written in vector-matrix notation as

$$\begin{aligned} I_n(\boldsymbol{\rho}) &= \mathbf{B}^\mathrm{T} \mathbf{c}_n, & \mathbf{B}_{ij} &= b_{R=i}(\rho = j), \\ P_n(\mathbf{r}) &= \mathbf{P}_n^\mathrm{T} \mathbf{c}_n, & (\mathbf{P}_n)_{ij} &= p_{R=i;n}(r = j) \end{aligned}$$

for each angular order n.

It is obvious from the definition of $b_R(\rho)$ that $\mathbf{B}$ is an identity matrix, so the expansion coefficients are simply $\mathbf{c}_n = I_n(\boldsymbol{\rho})$. Thus the forward and inverse Abel transforms can be computed as

$$\begin{aligned} P_n(\mathbf{r}) &= \mathbf{P}_n^\mathrm{T} I_n(\boldsymbol{\rho}), \\ I_n(\boldsymbol{\rho}) &= \left(\mathbf{P}_n^\mathrm{T}\right)^{-1} P_n(\mathbf{r}) \end{aligned}$$

for each angular order separately. Since all projected basis functions satisfy $p_{R;n}(r \geqslant R + 1) = 0$ (see the plots above), the matrices $\mathbf{P}_n^\mathrm{T}$ are upper triangular, and their inversions $\left(\mathbf{P}_n^\mathrm{T}\right)^{-1}$ are also upper triangular for all n, which additionally facilitates the computations. (This triangularity makes the inverse Abel transform similar to the *"onion peeling"* procedure written in a matrix form, but based on linear interpolation for spherical shells instead of midpoint rectangular approximation for cylindrical rings.)

Overall, the transforms proceed as follows:

1. Radial distributions for each angular order are extracted from the input data using `abel.tools.vmi.Distributions`. This takes $O(N\,R_\text{max}^2)$ time, where N is the number of angular terms, and R_max is the largest analyzed radius (assuming $N \ll R_\text{max}$).
2. Radial projected basis functions are computed to construct the $\mathbf{P}_n$ matrices, also in $O(N\,R_\text{max}^2)$ total time.
3. For the inverse Abel transform, the $\mathbf{P}_n^\mathrm{T}$ matrices are inverted, in $O(N\,R_\text{max}^3)$ total time. This step is not needed for the forward Abel transform.
4. The radial distributions from step 1 are multiplied by the transform matrices $\mathbf{P}_n^\mathrm{T}$ or $\left(\mathbf{P}_n^\mathrm{T}\right)^{-1}$ to obtain the reconstructed radial distributions, in $O(N\,R_\text{max}^2)$ total time.

5. If the transformed image is needed, it is constructed from its radial distributions obtained in step 4 using the first formula in this section. This takes $O(N\,R_{\text{max}}^2)$ time.

That is, only step 3 has time complexity that scales cubically with the image size, and all other steps have quadratic complexity. However, for the forward Abel transform, step 3 is not needed at all, and for the inverse Abel transform, its results can be cached. Thus processing a sequence of images takes time linearly proportional to the total number of processed pixels. In other words, the pixel throughput is independent of the image size.

Regularizations

The matrix equation

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

(in our case the vector $\mathbf{x}$ represents the radial part of the sought 3D distribution, the matrix $\mathbf{A}$ represents the forward Abel transform, and the vector $\mathbf{y}$ represents the radial part of the recorded projection) can be solved as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$$

if $\mathbf{A}$ is invertible. However, if the problem is ill-conditioned, computing $\mathbf{A}^{-1}$ might be problematic, and the solution might have undesirably amplified noise.

Regularization methods try to replace the ill-conditioned problem with a related better-conditioned one and use its solution as a well-behaved approximation to the solution of the original problem.

Tikhonov

Instead of inverting $\mathbf{A}$ explicitly, the solution of $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be found as

$$\mathbf{x} = \arg\min_{\mathbf{x}} (\mathbf{y} - \mathbf{A}\mathbf{x})^2,$$

from a quadratic minimization ("least-squares") problem, which is equivalent to the original problem, but makes evident that for ill-conditioned problems the minimum is very "flat", and many different vectors $\mathbf{x}$ can be accepted as a solution.

The idea of Tikhonov regularization is to add some small "regularizing" term to this minimization problem:

$$\tilde{\mathbf{x}} = \arg\min_{\mathbf{x}} \left[(\mathbf{y} - \mathbf{A}\mathbf{x})^2 + g[\mathbf{x}]\right]$$

that will help to select the "best" solution by imposing larger penalty on undesirable solutions. If this term is also a quadratic form

$$g[\mathbf{x}] = (\mathbf{\Gamma}\mathbf{x})^2$$

with some matrix $\mathbf{\Gamma}$ (not necessarily square), then the quadratic minimization problem is reduced back to a linear matrix equation and has the solution

$$\tilde{\mathbf{x}} = \mathbf{A}^{\mathrm{T}}\left(\mathbf{A}\mathbf{A}^{\mathrm{T}} + \mathbf{\Gamma}\mathbf{\Gamma}^{\mathrm{T}}\right)^{-1}\mathbf{y}.$$

In practice, it is convenient to define $\mathbf{\Gamma} = \varepsilon\mathbf{\Gamma}_0$ with some fixed $\mathbf{\Gamma}_0$ and change the "Tikhonov factor" ε to adjust the regularization "strength". The form of $\mathbf{\Gamma}_0$ selects the regularization type:

L_2 norm

This is the simplest case, with $\mathbf{\Gamma}_0 = \mathbf{I}$, the identity matrix. That is, the penalty functional $g[\mathbf{x}] = \varepsilon^2\mathbf{x}^2$ is the quadratic norm of the solution scaled by the regularization parameter.

The idea is that, in a continuous limit, if we have a well-behaved function $f(r)$ and some random noise $\delta(r)$, then

$$\begin{aligned} g[f(r)+\delta(r)] &= \int [f(r)+\delta(r)]^2\,dr = \\ &= \underbrace{\int [f(r)]^2\,dr}_{g[f(r)]} + \underbrace{2\int f(r)\delta(r)\,dr}_{\approx 0} + \underbrace{\int [\delta(r)]^2\,dr}_{>0}. \end{aligned}$$

In other words, a noisy solution will have a larger penalty than a smooth solution, unless the noise is correlated, and a smooth solution will be preferred as long as the noise forward transforms is close to zero ($\|\mathcal{A}\delta(r)\| < \|\delta(r)\|$).

Notice, however, that for very large Tikhonov factors the regularization term starts to dominate in the minimization problem, which tends to

$$\tilde{\mathbf{x}} = \varepsilon^2 \arg\min_{\mathbf{x}} \mathbf{x}^2$$

and thus has the solution $\tilde{\mathbf{x}} \to 0$. For reasonable regularization strengths this intensity suppression effect is small, but the solution is always biased towards zero.

Finite differences

Here the first-order finite difference operator is used as the Tikhonov matrix:

$$\mathbf{\Gamma}_0 = \begin{pmatrix} -1 & 1 & 0 & 0 & \cdots \\ 0 & -1 & 1 & 0 & \cdots \\ 0 & 0 & -1 & 1 & \ddots \\ \vdots & \vdots & \ddots & \ddots & \ddots \end{pmatrix}.$$

It is the discrete analog of the differentiation operator, so in a continuous limit this regularization corresponds to using the penalty functional of the form

$$g[f(r)] = \int \left(\frac{df(r)}{dr}\right)^2\,dr.$$

Noisy functions obviously have larger RMS derivatives that smooth functions and thus are penalized more.

Unlike the L_2-norm regularization, which tends to avoid sign-changing functions and oscillating functions in general, this regularization can produce noticeably overshoots (including negative) around sharp features in the distribution. However, it tends to preserve the overall intensity.

Truncated SVD

This is the method mentioned in the pBasex article[2] (but not actually used in the original pBasex implementation). The idea is that since the condition number of a matrix equals the ratio of its maximal and minimal singular values, performing the singular value decomposition (SVD),

$$\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} = \mathbf{A},$$

inverting $\mathbf{\Sigma}$ (which is diagonal), then excluding its largest values values and assembling the pseudoinverse

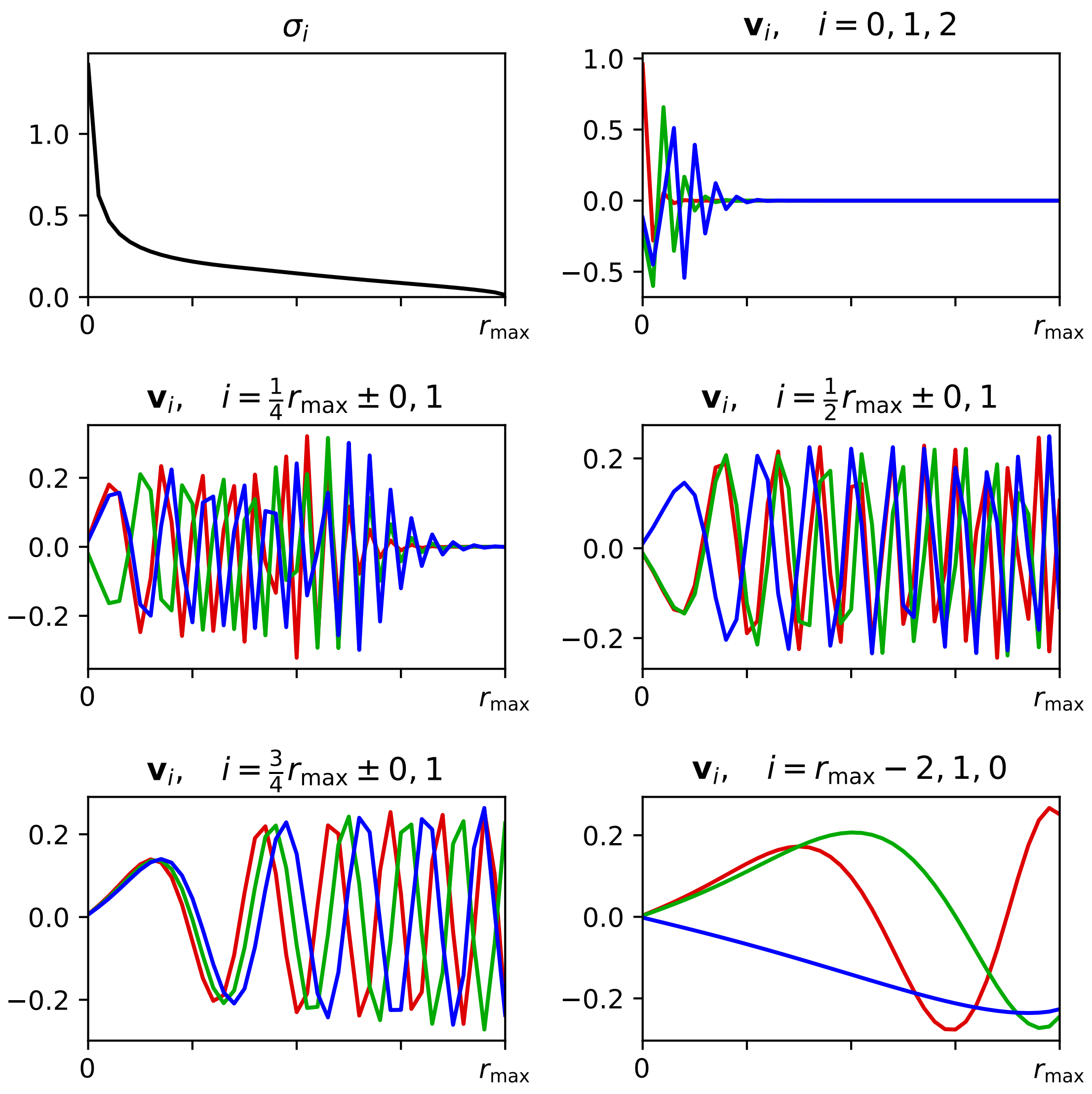
$$\tilde{\mathbf{A}}^{-1} = \mathbf{V}\tilde{\mathbf{\Sigma}}^{-1}\mathbf{U}^{\mathrm{T}}$$

gives a better-conditioned matrix approximation of $\mathbf{A}^{-1}$, which is then used to obtain the approximate solution

$$\tilde{\mathbf{x}} = \tilde{\mathbf{A}}^{-1}\mathbf{y}.$$

This approach can be helpful when the left singular vectors (columns of $\mathbf{V}$, which become linear contributions to $\mathbf{x}$) are physically meaningful and different for the useful signal and the undesirable noise. Then removing the singular values corresponding to the undesirable vectors excludes them from the solution, while retaining the useful contributions. However, this is not the case for our problem. Here are the singular values σ_i of $\mathbf{A}^{-1}$ plotted together with some representative left singular vectors $\mathbf{v}_i$:

[2] G. A. Garcia, L. Nahon, I. Powis, "Two-dimensional charged particle image inversion using a polar basis function expansion", Rev. Sci. Instrum. 75, 4989–4996 (2004).



As can be seen, all these vectors are oscillatory, with negative values, and most of them are delocalized over the whole radial range. That is, they do not have a clear physical meaning for practical applications of the Abel transform.

The only potentially useful observation is that the first vectors, corresponding to the largest singular values, have the highest spacial frequencies and contribute mostly to the lower r range. Thus excluding them might reduce the high-frequency noise near the center of the transformed image. It should be noted, however, that a simple SVD truncation leads to the same problems with delocalized oscillations and the Gibbs phenomenon, as in truncated Fourier series. (From this perspective, soft attenuation, like in the Tikhonov regularization, is a more appropriate approach.)

So this method is not recommended for practical applications and is provided here mostly for completeness.

Non-negative components

This is the simplest *nonlinear* regularization method proposed in[Page 142, 1]. The idea is that the linear matrix equation

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

is replaced by the minimization problem

$$\tilde{\mathbf{x}} = \mathop{\arg\min}_{\mathbf{x} \geqslant 0} (\mathbf{y} - \mathbf{A}\mathbf{x})^2$$

with a physically meaningful constraint that the solution (the intensity distribution) must be non-negative everywhere. If the linear solution happens to be non-negative, this modified problem has exactly the same solution. Otherwise the minimization problem gives the closest (in the least-squares sense) non-negative approximation to the original problem.

Unfortunately, applying non-negativity constraint to trigonometric polynomials,

$$I(\theta) = \sum a_n \cos^n \theta \geqslant 0 \text{ for all } \theta,$$

generally leads to a system of nonlinear inequalities for their coefficients, which cannot be solved efficiently.

However, if the polynomial has no more that two terms, that is its order is 0, 1, or 2 with even powers only, the constraints are linear and can be linearly transformed into nonnegativity constraints on the coefficients:

$$\begin{aligned}
I(\theta) &= c_0 \cos^0 \theta \geqslant 0 & \Leftrightarrow \quad & c_0 \geqslant 0; \\
I(\theta) &= c_0 \cos^0 \theta + c_1 \cos^1 \theta = \\
&= a_0(\cos^0 \theta + \cos^1 \theta) + \\
&+ a_1(\cos^0 \theta - \cos^1 \theta) \geqslant 0 & \Leftrightarrow \quad & a_i \geqslant 0; \\
I(\theta) &= c_0 \cos^0 \theta + c_2 \cos^2 \theta = \\
&= a_0(\cos^0 \theta - \cos^2 \theta) + \\
&+ a_2 \cos^2 \theta \geqslant 0 & \Leftrightarrow \quad & a_i \geqslant 0.
\end{aligned}$$

Notice that in the last case the term $a_0(\cos^0 \theta - \cos^2 \theta) = a_0 \sin^2 \theta$ corresponds to perpendicular transitions, whereas $a_2 \cos^2 \theta$ corresponds to parallel transitions, so the inequalities $a_i \geqslant 0$ have a direct physical meaning that both transition components must be non-negative.

The quadratic minimization problem with linear constraints reduces to a sequence of linear problems and is soluble exactly in a finite number of steps.

In some cases the non-negative solution can be positively biased, since it does not allow negative noise, but can have some positive noise. Nevertheless, this bias is smaller than the positive bias introduced by zeroing negative values in solutions obtained by linear methods (*never do this*!).

The idea of non-negative transition components can be extended to multiphoton processes *without interference between different channels*, so that

$$\begin{aligned}
I(\theta) = &\left(a_0^{(1)} \sin^2 \theta + a_2^{(1)} \cos^2 \theta\right) \times \\
&\times \left(a_0^{(2)} \sin^2 \theta + a_2^{(2)} \cos^2 \theta\right) \times \\
&\times \cdots \times \\
&\times \left(a_0^{(m)} \sin^2 \theta + a_2^{(m)} \cos^2 \theta\right), \quad a_i^{(j)} \geqslant 0,
\end{aligned}$$

which also leads to a linear combination with non-negative coefficients:

$$I(\theta) = \sum b_n \sin^m \theta \cdot \cos^n \theta, \quad b_n \geqslant 0.$$

These constraints, however, are stronger than the intensity non-negativity: for example, the angular distribution

$$\sin^4\theta - 2\sin^2\theta\cdot\cos^2\theta + \cos^4\theta = \left(\sin^2\theta - \cos^2\theta\right)^2$$

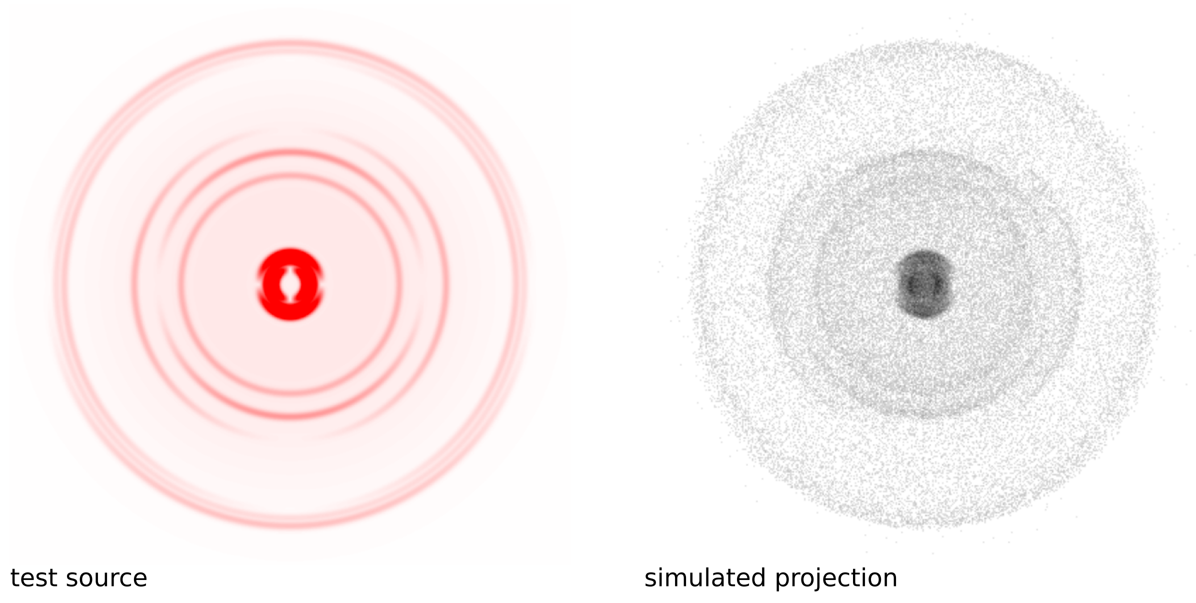
is non-negative everywhere, but contains a negative coefficient for the $\sin^2\theta\cdot\cos^2\theta$ term. So even though this regularization is not always valid for multiphoton processes, it can be useful in some cases and is easy to implement. To remind that it is not "truly non-negative", this regularization is called "positive" here.

A general advice applicable to all regularization methods is that when a relevant model is available, it is better to fit it directly to non-regularized results, thus avoiding additional assumptions and biases introduced by regularizations.

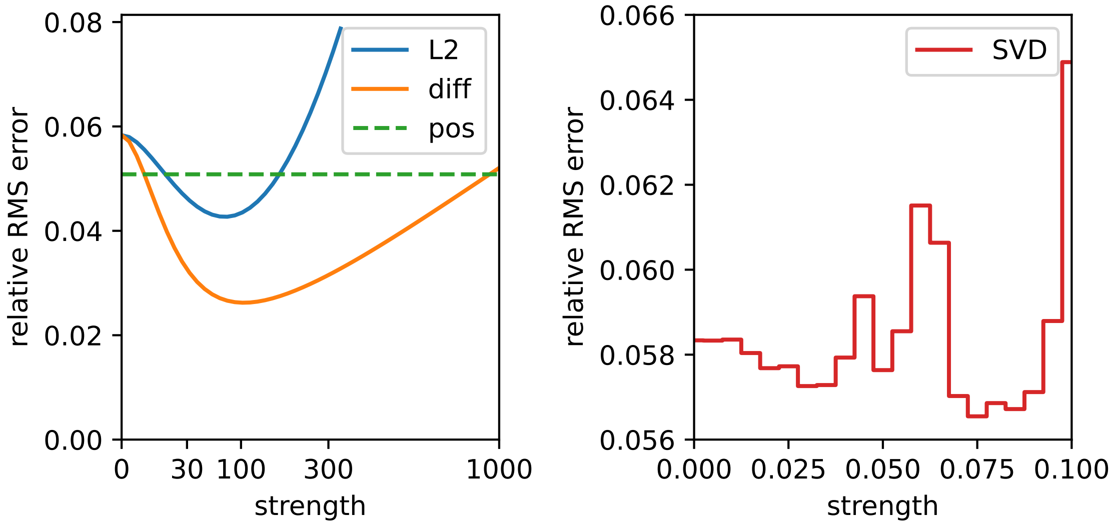
Examples

Warning: Absolute and relative efficiencies of these regularization methods and their optimal parameters depend on the image size, the amount of noise and the distribution itself. Therefore *do not assume* that the examples shown here are directly relevant to *your* data.

Some properties of the regularization methods described above are demonstrated here by applying them to a synthetic example. The test distribution from the BASEX article is forward Abel-transformed to obtain its projection, and then Poissonian noise is added to it to simulate experimental VMI data with relatively low signal levels (such that the noise is prominent):



In order to characterize the regularization performance, all the methods are applied at various strengths to this simulated projection, and the relative root-mean-square error $\|\tilde{I}(r)-I_{\text{src}}(r)\|/\|I_{\text{src}}(r)\|$, where $I_{\text{src}}(r)$ is the "true" radial intensity distribution, and $\tilde{I}(r)$ is the reconstructed distribution, is calculated in each case. The following plot shows how this reconstruction error changes with the regularization strength (the non-parameterized "pos" method is shown by a dashed line):



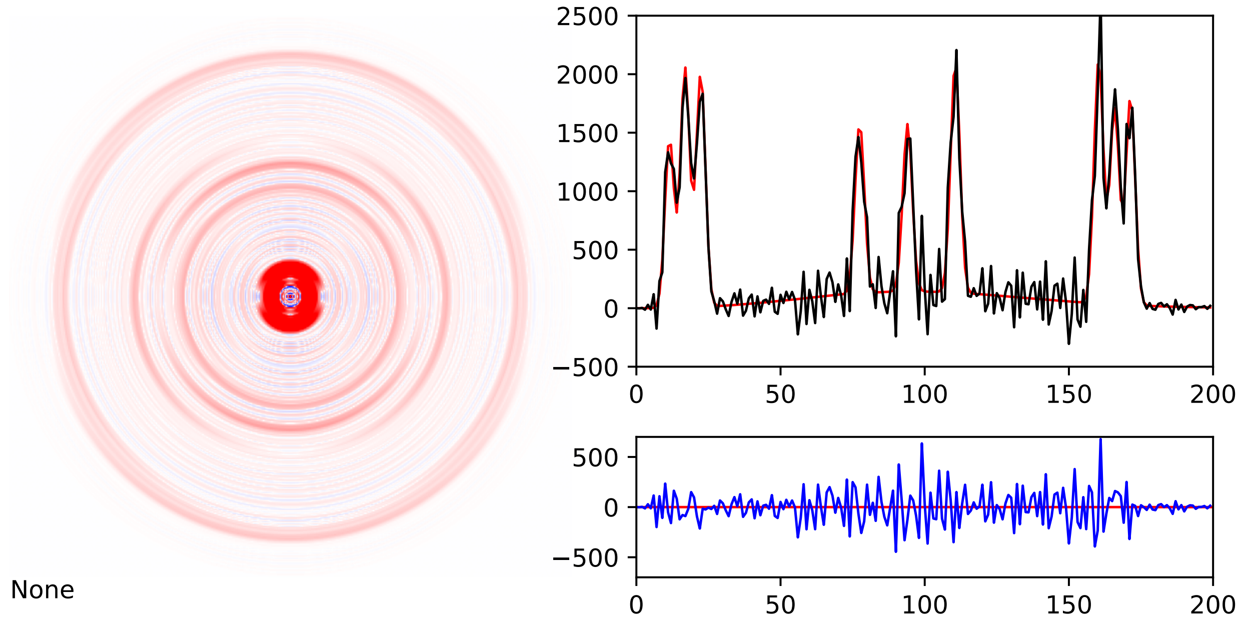
(Note that the horizontal axis in the left plot is nonlinear, and that the vertical axis in the right plot does not start at zero and actually spans a very small range.)

These plots demonstrate that the Tikhonov methods have some optimal strength value, at which the reconstruction error is minimized. At smaller values the noise is not suppressed enough (zero strength corresponds to the non-regularized transform), and at larger values the reconstructed distribution is smoothed too much.

The SVD plot has steps corresponding to successive removal of singular values. The reconstruction error does not decrease monotonically, but exhibits several local minima before starting to grow. Notice that even the global minimum is only slightly better than no regularization.

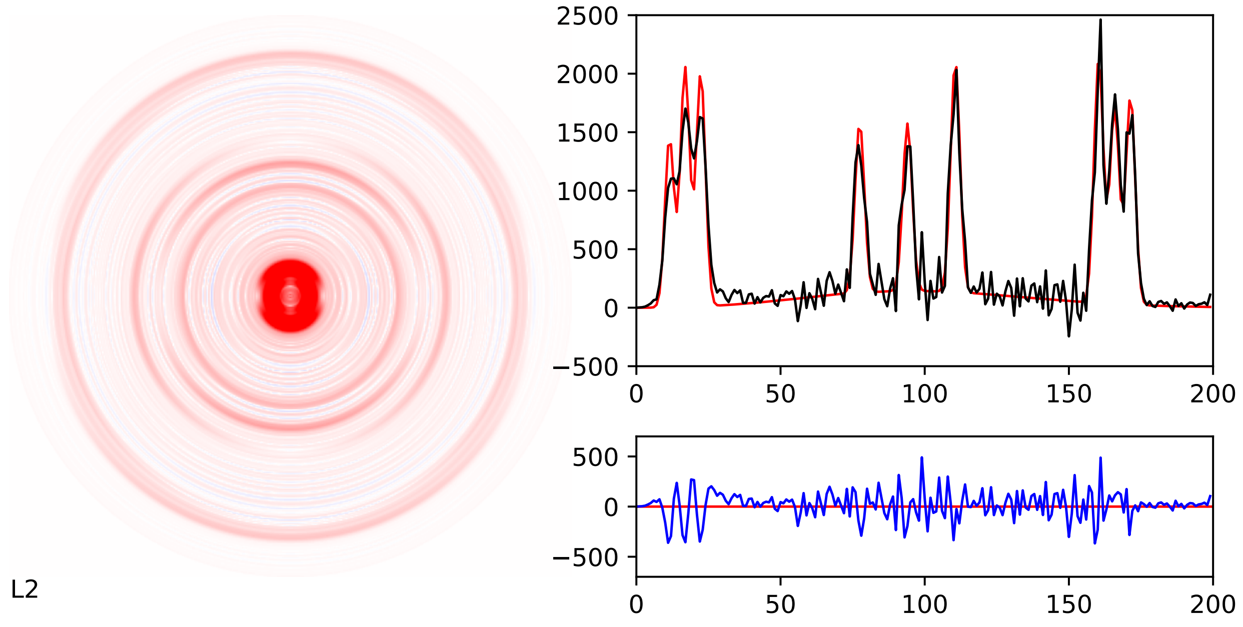
The actual reconstructed images for each regularization method at its optimal strength are shown below with their radial intensity distributions:

Using `reg=None`



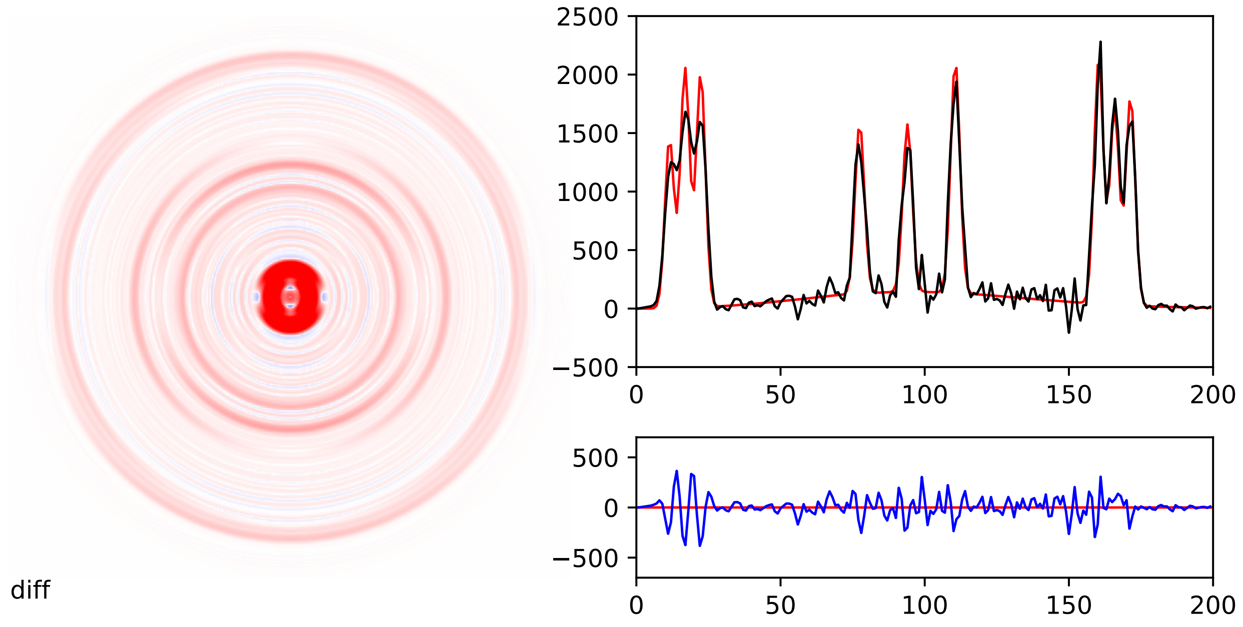
The non-regularized transform results are shows as a reference. The image has red colors for positive intensities and blue colors for negative intensities. The upper plot shows the reconstructed radial intensity distribution in black and the "true" distribution in red behind it. The lower plot shows the the difference between these two distributions in blue (red is the zero line).

Using `reg=('L2', 75)`



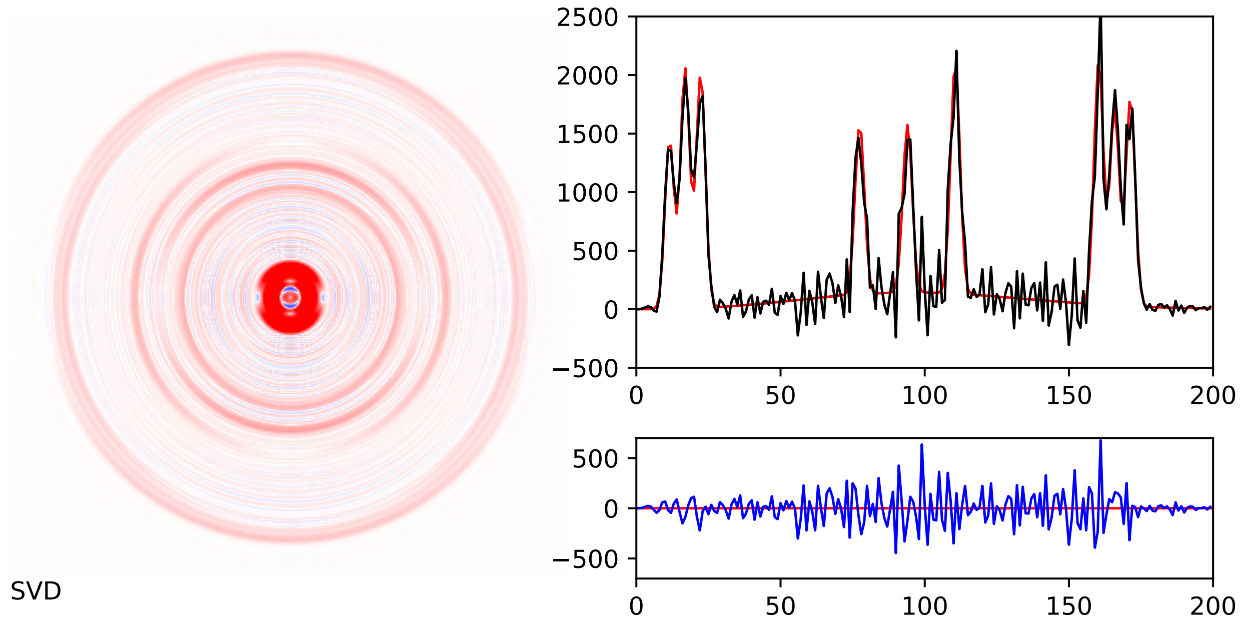
The noise level is generally reduced, but the peaks near the origin are noticeably broadened, which actually increases deviations in this region.

Using `reg=('diff', 100)`



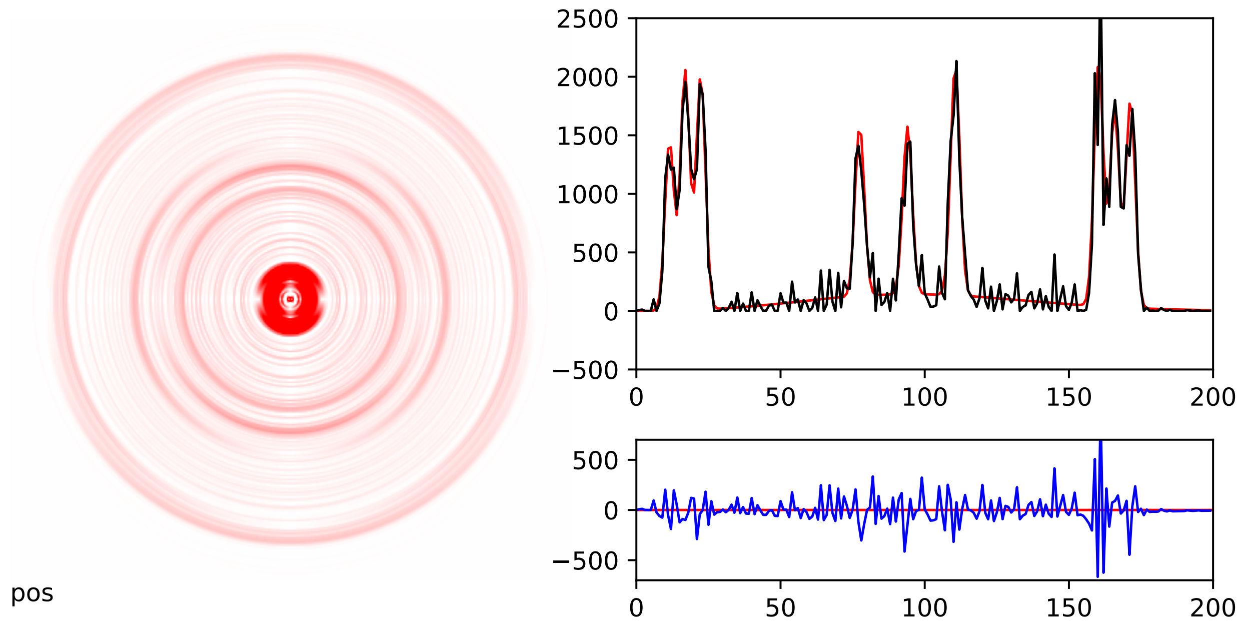
The noise is reduced even more, especially its high-frequency components. The peaks near the origin also suffer, but somewhat differently.

Using `reg=('SVD', 0.075)`



The only noticeable difference from no regularization is some noise reduction near the origin.

Using `reg='pos'`



The most prominent feature is the absence of negative intensities. The noise is reduced significantly in the areas of low intensity, where it is constrained from attaining negative values, which also reduces its positive amplitudes, as the distribution should be reproduced on average. The peaks, being strongly positive, do not have noticeable noise reduction. However, in contrast to other methods, the peaks near the origin are not broadened, while the off-peak noise near them is reduced.

4.10 Three Point

4.10.1 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].

[1] C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

4.10.2 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 $\mathbf{F}$.

4.10.3 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.gaussian_filter.)

4.10.4 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()`*.

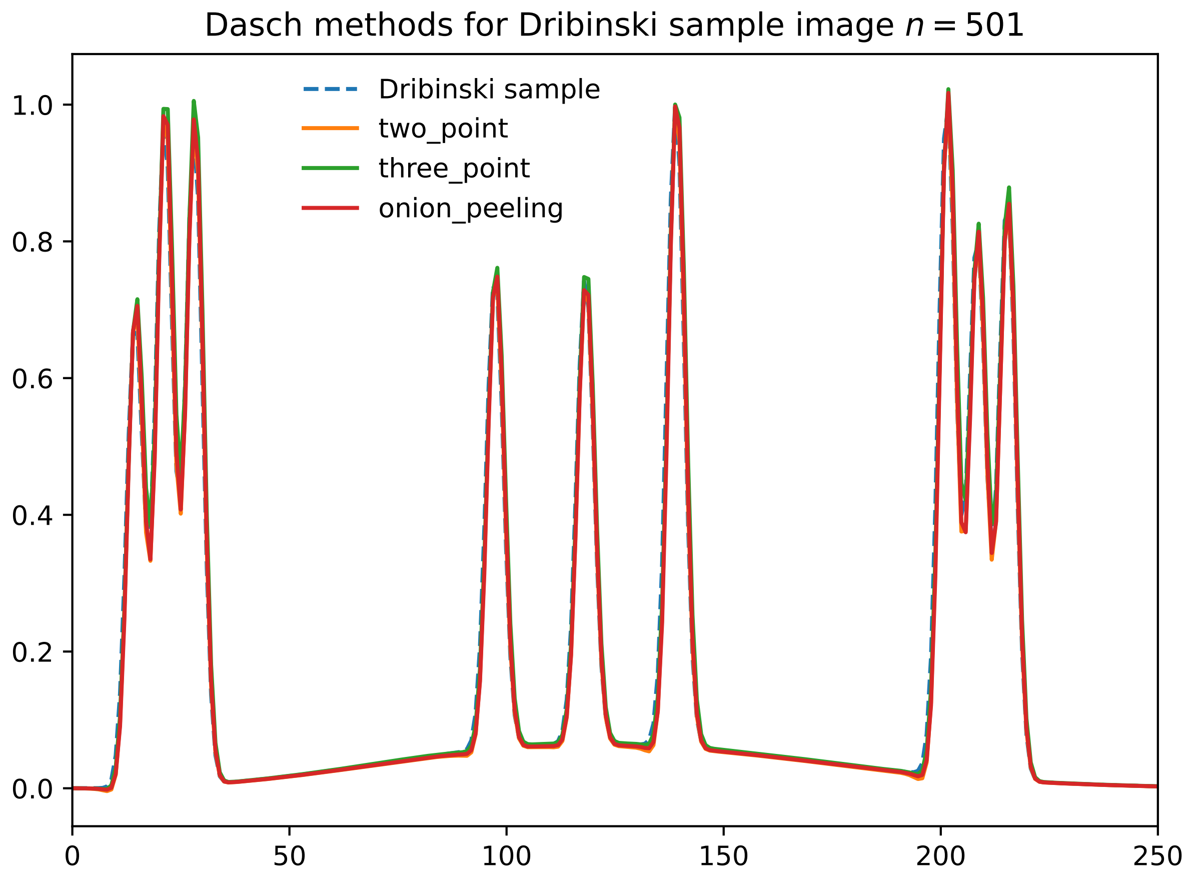
4.10.5 Example

```
# -*- 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
import matplotlib.pyplot as plt

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

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

(continues on next page)

(continued from previous page)

```python
# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration_3D(origQ[0], origin=(-1, 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)
# speed distribution
    speed = abel.tools.vmi.angular_integration_3D(AQ0, origin=(-1, 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.xlim((0, 250))
plt.legend(loc='upper center', bbox_to_anchor=(0.35, 1), frameon=False)
plt.tight_layout()
# plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```



4.10.6 Notes

The algorithm contained two typos in Eq. (7) in the original citation[Page 153, 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.

4.10.7 Citation

- C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).
- K. Martin, PhD Thesis: "Acoustic Modification of Sooting Combustion", University of Texas at Austin (2002) (record, PDF).

[2] K. Martin, PhD Thesis: "Acoustic Modification of Sooting Combustion", University of Texas at Austin (2002) (record, PDF).

4.11 Two Point (Dasch)

4.11.1 Introduction

The "Dasch two-point" deconvolution algorithm is one of several described in the Dasch paper[1]. See also the *"three-point"* and *"onion peeling"* descriptions.

4.11.2 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}.

4.11.3 When to use it

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

4.11.4 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()*.

4.11.5 Example

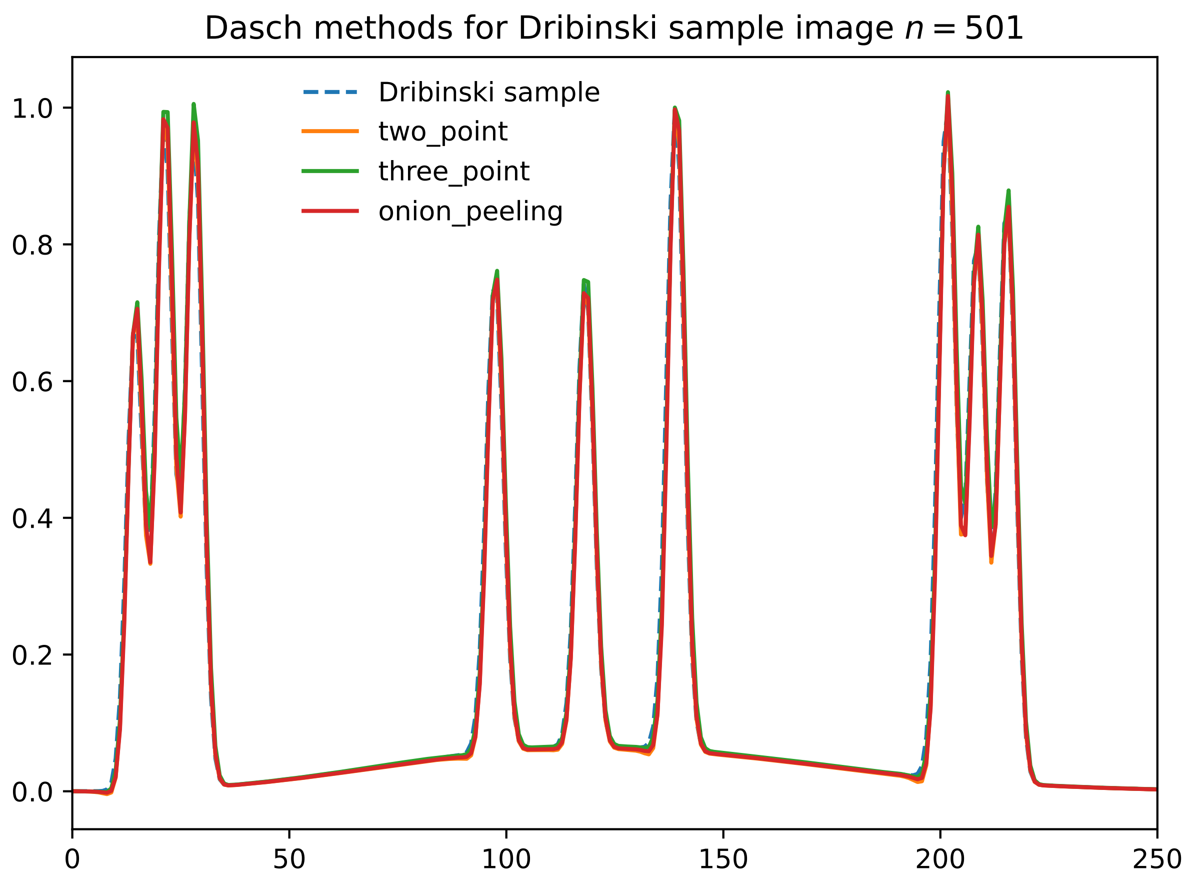
```
# -*- coding: utf-8 -*-
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

"""example_dasch_methods.py.
"""

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

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

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

(continues on next page)

[1] C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

(continued from previous page)

```
# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration_3D(origQ[0], origin=(-1, 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)
# speed distribution
    speed = abel.tools.vmi.angular_integration_3D(AQ0, origin=(-1, 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.xlim((0, 250))
plt.legend(loc='upper center', bbox_to_anchor=(0.35, 1), frameon=False)
plt.tight_layout()
# plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```



For more information on the PyAbel implementation of the `two_point` algorithm, please see PR #155.

4.11.6 Citation

- C. J. Dasch, "One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods", Appl. Opt. 31, 1146–1152 (1992).

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 (ϵ) dependent anisotropy parameter, which varies between -1 and +2, and $P_2(\cos\theta)$ is the 2nd-order Legendre polynomial in $\cos\theta$. σ_{total} is the total photodetachment cross section. The anisotropy parameter provides phase information about the dynamics of the photon process[1].

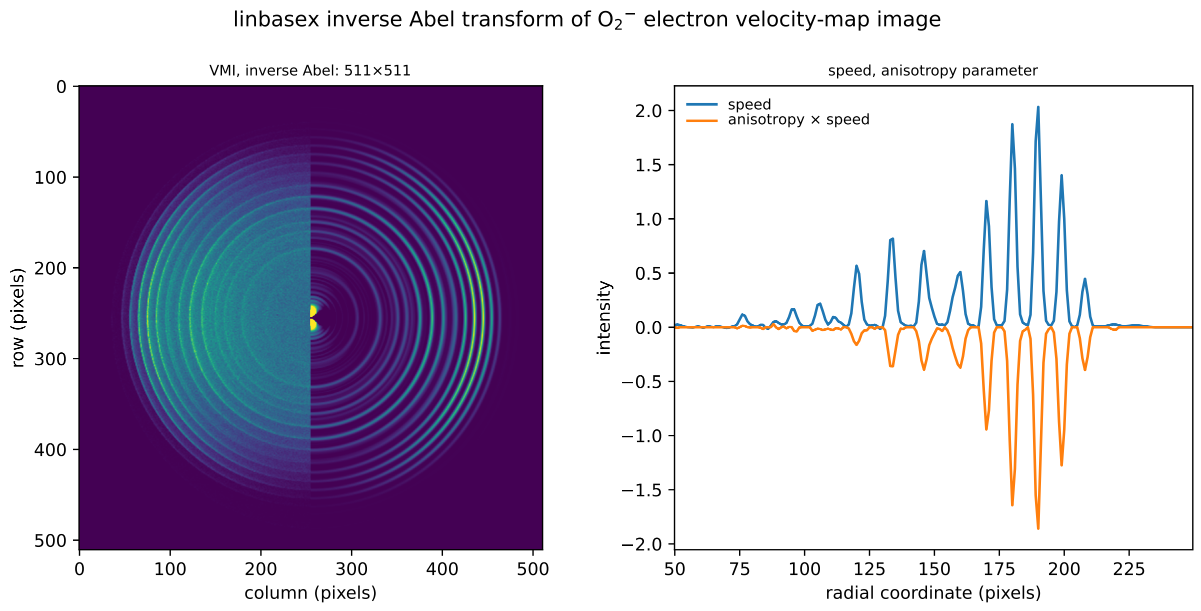
5.1 Methods

`PyAbel` provides several methods to determine the anisotropy parameter β:

Method 1: *linbasex* evaluates β 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 *Example: Linbasex*.

[1] J. Cooper, R. N. Zare, "Angular Distribution of Photoelectrons", J. Chem. Phys. 48, 942–943 (1968).



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).

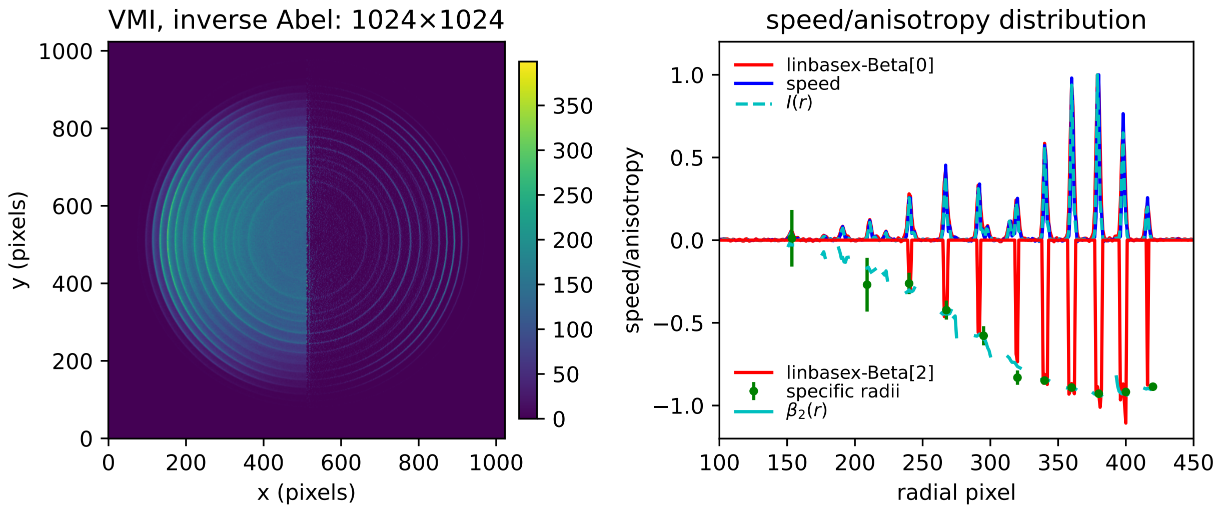
Method 3: using *abel.tools.vmi.Distributions*.

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.



A demonstration of using *Distributions* for incomplete images is also included in *Example: rBasex beam block*.

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 $R_i(\text{actual}) = R_i \times \text{scalefactor}_i$. 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, S. T. Gibson, 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, θ) 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 `tol`.

6.6 Example

```
import numpy as np
import matplotlib.pyplot as plt
import abel
from abel.tools.circularize import circularize, circularize_image
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 -----------
```

(continues on next page)

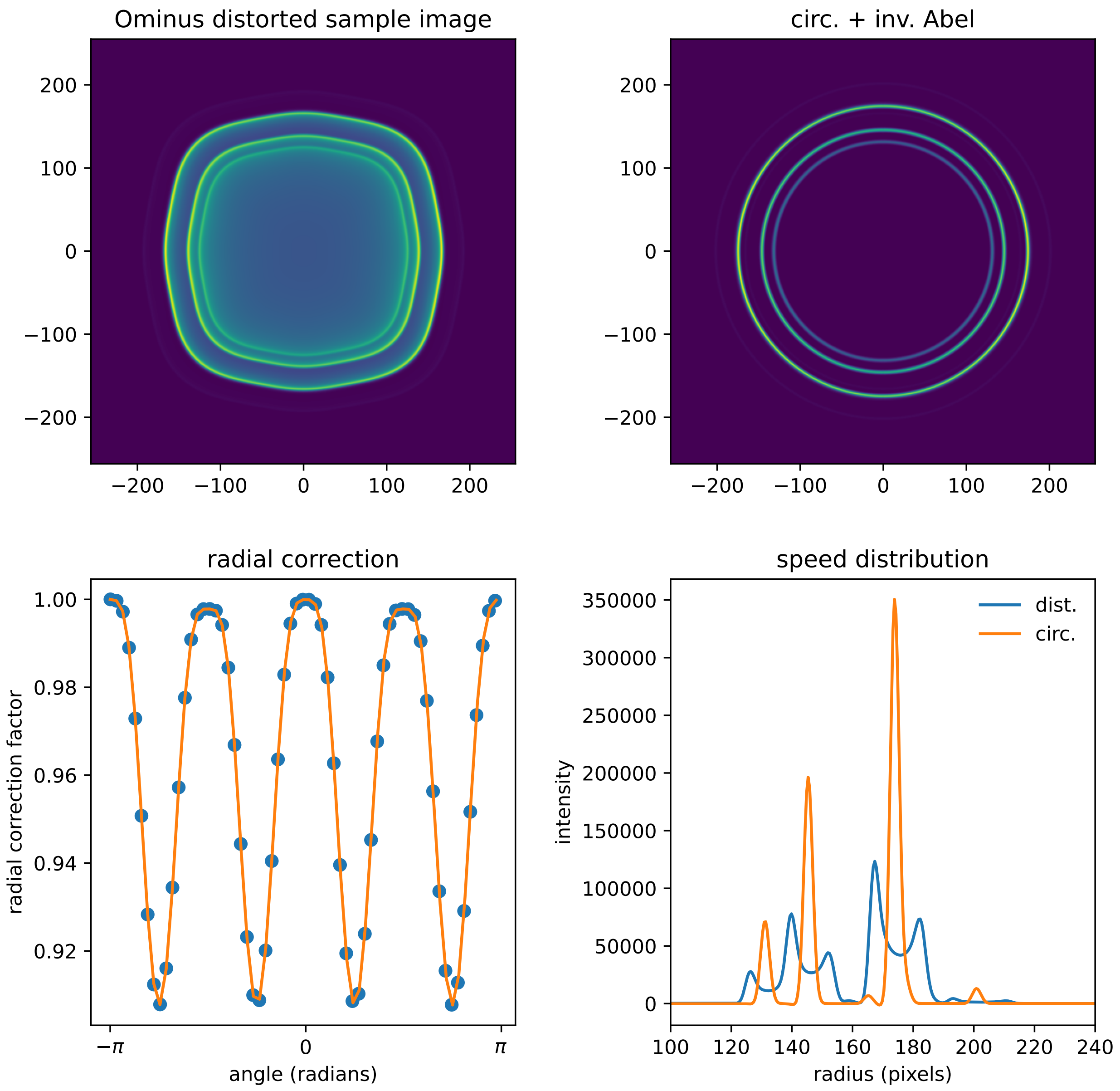
(continued from previous page)

```
IM = abel.tools.analytical.SampleImage(n=511, name='Ominus', sigma=2).func

# 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 = circularize(IMf, radial_correction_function=flower_scaling)

# circularize ------------
IMcirc, sla, sc, scspl = circularize_image(IMdist,
                                           method='lsq', dr=0.5, dt=0.1,
                                           tol=0, return_correction=True)

# inverse Abel transform for distored and circularized images ---------
AIMdist = abel.Transform(IMdist, method="three_point").transform
AIMcirc = abel.Transform(IMcirc, method="three_point").transform

# respective speed distributions
rdist, speeddist = abel.tools.vmi.angular_integration_3D(AIMdist, dr=0.5)
rcirc, speedcirc = abel.tools.vmi.angular_integration_3D(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, origin='lower', extent=extent)
axs[0, 0].set_title("Ominus distorted sample image")

axs[0, 1].imshow(AIMcirc, vmin=0, 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.tight_layout(h_pad=2, w_pad=2)
# plt.savefig("plot_example_circularize_image.png", dpi=75)
plt.show()
```



Chapter 7

Examples

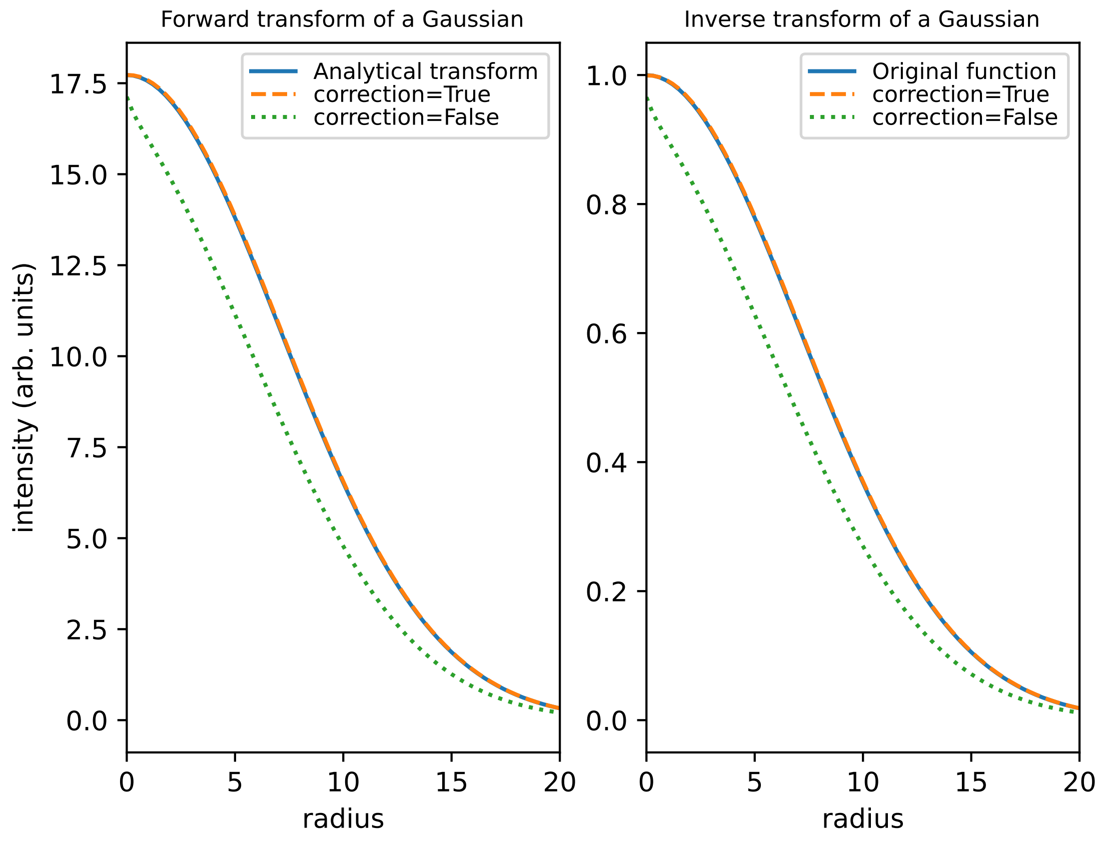
7.1 Example: Direct Gaussian

```
# -*- 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)

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)')
```

(continues on next page)

(continued from previous page)

```
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: 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, method="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_3D(AIM)

# 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
```

(continues on next page)

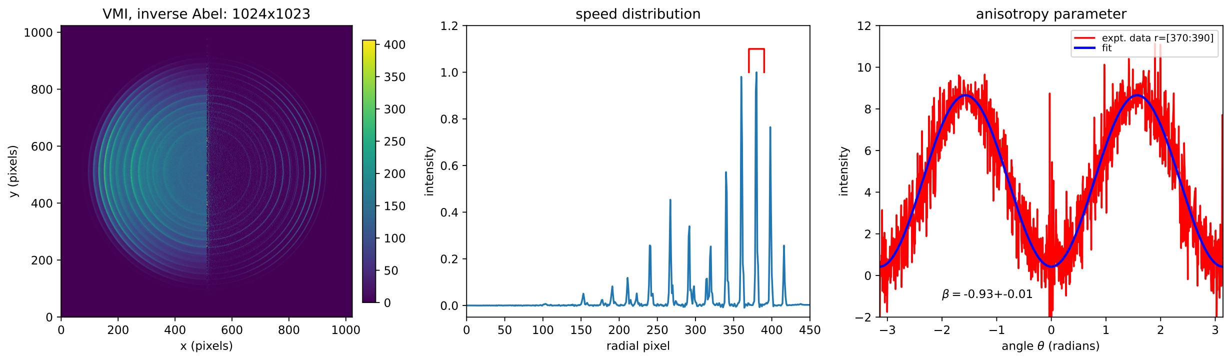
(continued from previous page)

```
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, 4.5))
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', 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(xmin=0, 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
    return (3*x*x-1)/2
```

(continues on next page)

(continued from previous page)

```
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(xmin=-np.pi, xmax=np.pi, ymin=-2, ymax=12)
ax3.set_xlabel("angle $\\theta$ (radians)")
ax3.set_ylabel("intensity")
ax3.set_title("anisotropy parameter")


# Plot the angular distribution
plt.tight_layout()

# Save a image of the plot
# plt.savefig("plot_example_O2_PES_PAD.png", dpi=100)

# Show the plots
plt.show()
```



7.3 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
```

(continues on next page)

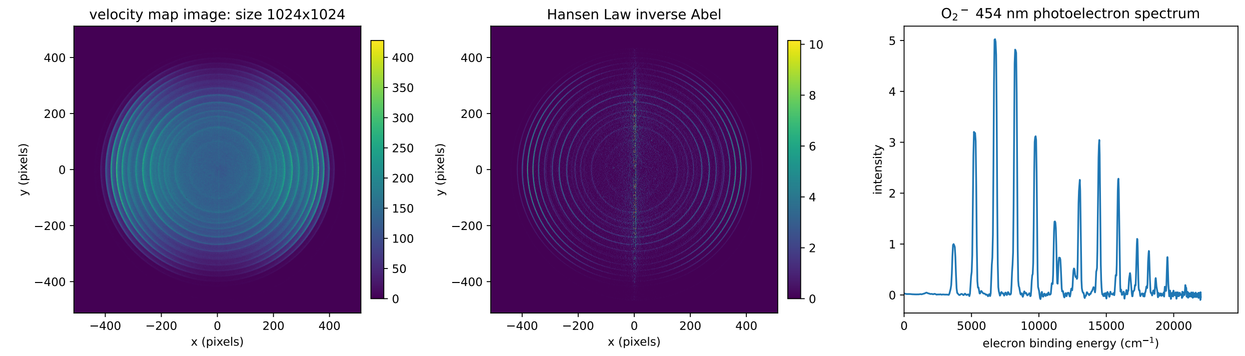
(continued from previous page)

```python
# 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

# center image returning odd size
IMc = abel.tools.center.center_image(IM, method='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, 4.5))
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, 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, 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)')
```

(continues on next page)

(continued from previous page)

```
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')

# Prettify the plot a little bit:
plt.tight_layout()

# save copy of the plot
# plt.savefig('plot_example_hansenlaw.png', dpi=100)

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
```

(continues on next page)

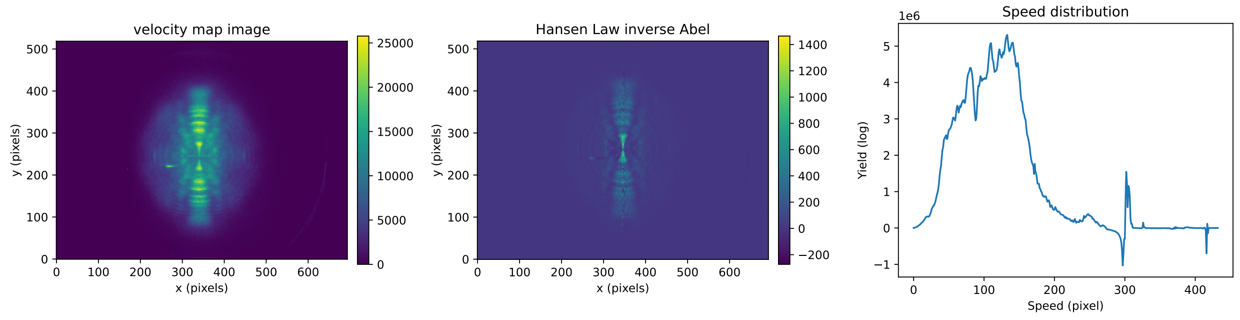
(continued from previous page)

```
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]
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,
                       origin=(240, 340)).transform

r, speeds = abel.tools.vmi.angular_integration_3D(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')
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')
fig.colorbar(im2, ax=ax2, fraction=.1, shrink=0.9, pad=0.03)
ax2.set_xlabel('x (pixels)')
ax2.set_ylabel('y (pixels)')
```

(continues on next page)

(continued from previous page)

```python
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.tight_layout()

# Save a image of the plot
# plt.savefig(output_plot, dpi=100)

# Show the plots
plt.show()
```



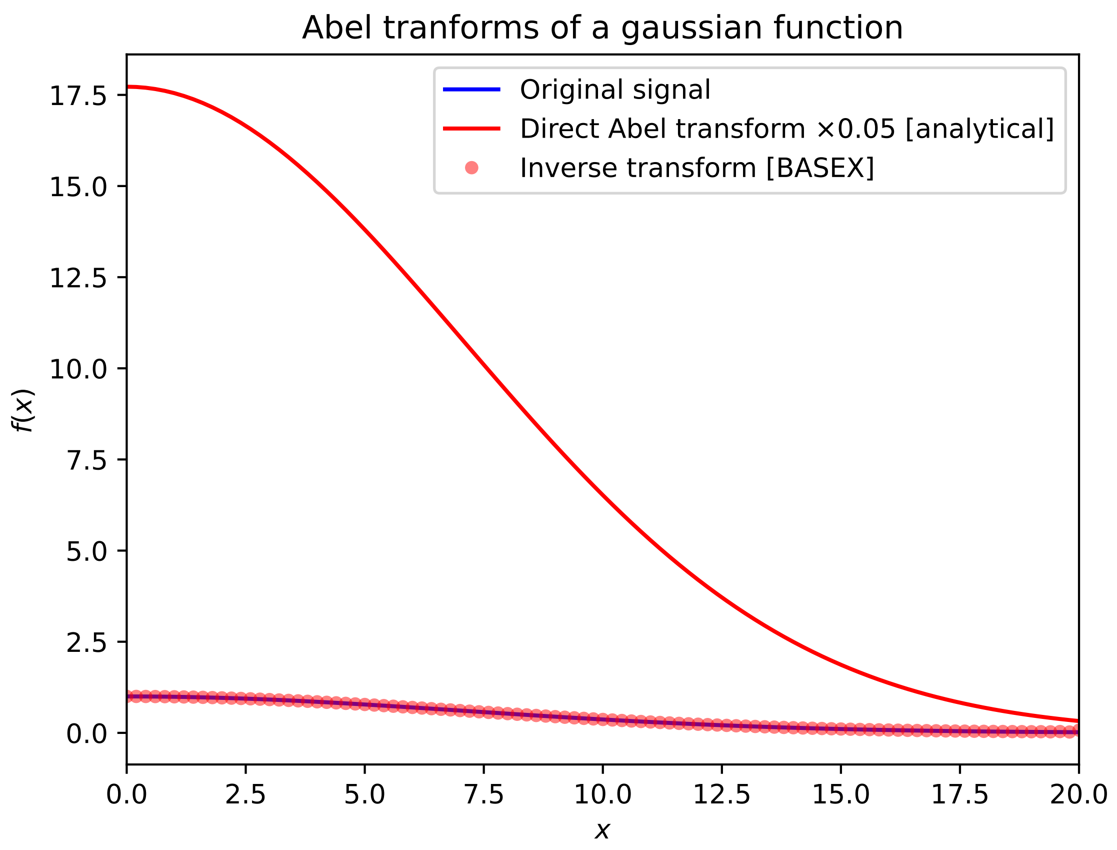
7.5 Example: Basex Gaussian

```python
# -*- coding: utf-8 -*-
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)
```

(continues on next page)

(continued from previous page)

```
ax.plot(ref.r, ref.func, 'b', label='Original signal')
ax.plot(ref.r, ref.abel, 'r', label='Direct Abel transform ×0.05 [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()
```



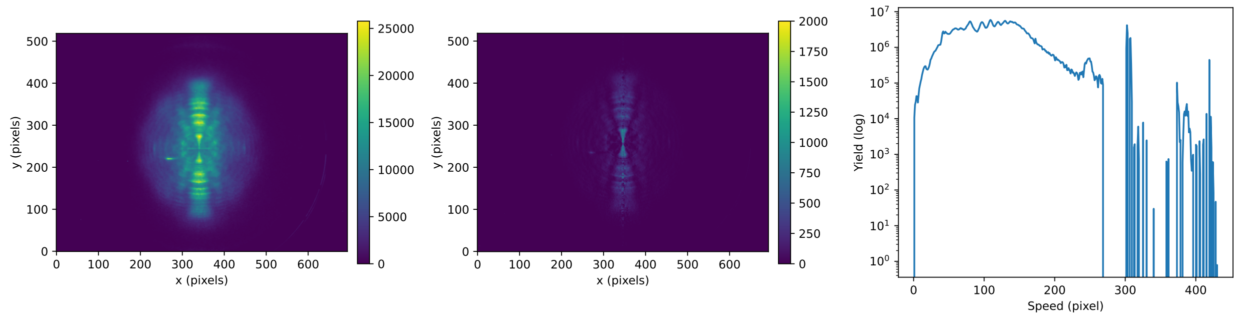
7.6 Example: Basex photoelectron

```python
# -*- 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
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)
# https://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.
# https://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 origin in (row, col) format
origin = (245, 340)
# or, use automatic centering
# origin = 'com'
# origin = 'gaussian'

# Step 3: perform the BASEX transform!
print('Performing the inverse Abel transform:')

recon = abel.Transform(raw_data, direction='inverse', method='basex',
                       origin=origin, verbose=True).transform

speeds = abel.tools.vmi.angular_integration_3D(recon)
```

(continues on next page)

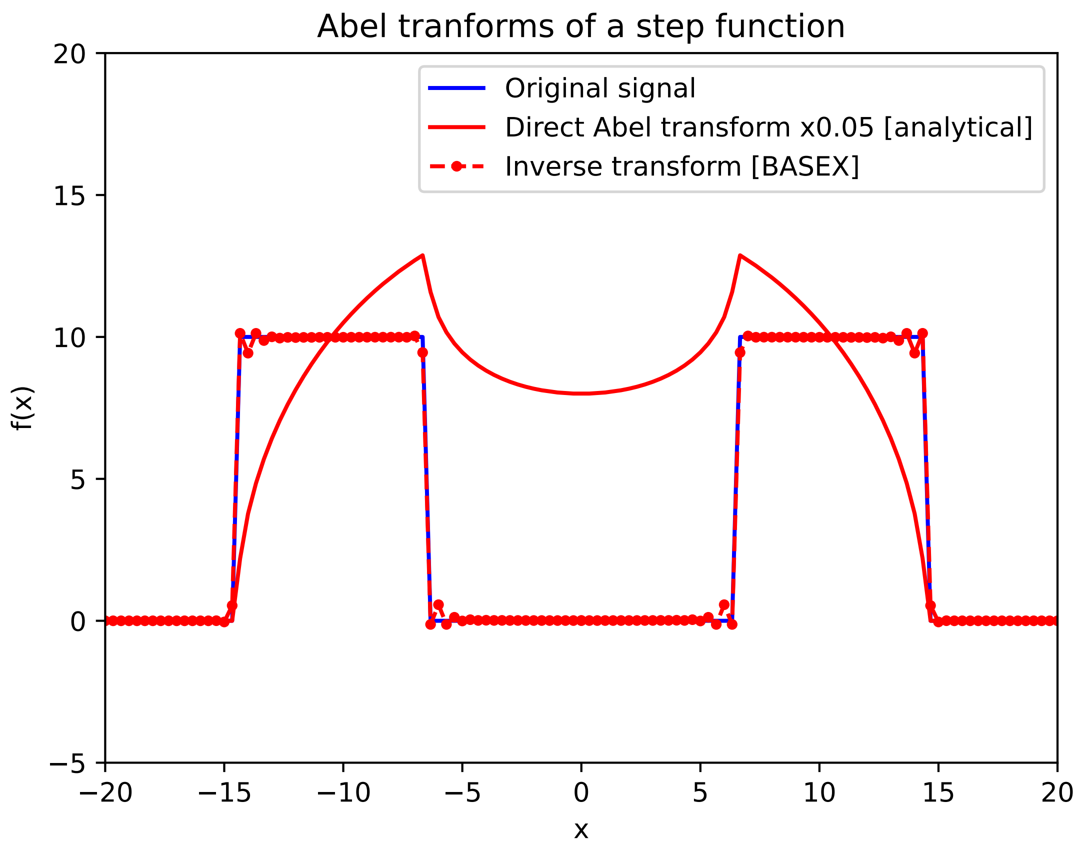
(continued from previous page)

```
# 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')
fig.colorbar(im1, ax=ax1, fraction=0.1, 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', clim=(0, 2000))
fig.colorbar(im2, ax=ax2, fraction=0.1, shrink=0.9, pad=0.03)
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.tight_layout()

# Show the plots
plt.show()
```



7.7 Example: Basex step function

```python
import matplotlib.pyplot as plt
from abel.basex import basex_transform
from abel.tools.analytical import StepAnalytical

# This example calculates the BASEX transform of a step function and
# compares with the analtical result.
fig, ax = plt.subplots(1, 1)
plt.title('Abel tranforms of a step function')

n = 301
r_max = 50
A0 = 10.0
r1 = 6.5
r2 = 14.5

# define a symmetric step function and calculate its analytical Abel transform
st = StepAnalytical(n, r_max, r1, r2, A0)

ax.plot(st.r, st.func, 'b', label='Original signal')

ax.plot(
    st.r, st.abel*0.05, 'r',
    label='Direct Abel transform x0.05 [analytical]')

center = n//2
right_half = st.abel[center:]
left_half = st.abel[:center+1][::-1]
# BASEX Transform:
# Calculate the inverse abel transform for the centered data
recon_right = basex_transform(right_half, dr=st.dr, verbose=True)
recon_left = basex_transform(left_half, dr=st.dr, verbose=False)
plt.plot(
    st.r[center:], recon_right, '--.', c='red',
    label='Inverse transform [BASEX]')

plt.plot(st.r[:center+1], recon_left[::-1], '--.', c='red')

ax.legend()

ax.set_xlim(-20, 20)
ax.set_ylim(-5, 20)
ax.set_xlabel('x')
ax.set_ylabel("f(x)")

plt.legend()
plt.show()
```



7.8 Example: All Dribinski

```
# -*- coding: utf-8 -*-

# This example compares some available inverse Abel transform methods
# for the Dribinski sample image

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

from time import time
import matplotlib.pylab as plt
import numpy as np

import abel

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# inverse Abel transform methods -----------------------------
```

(continues on next page)

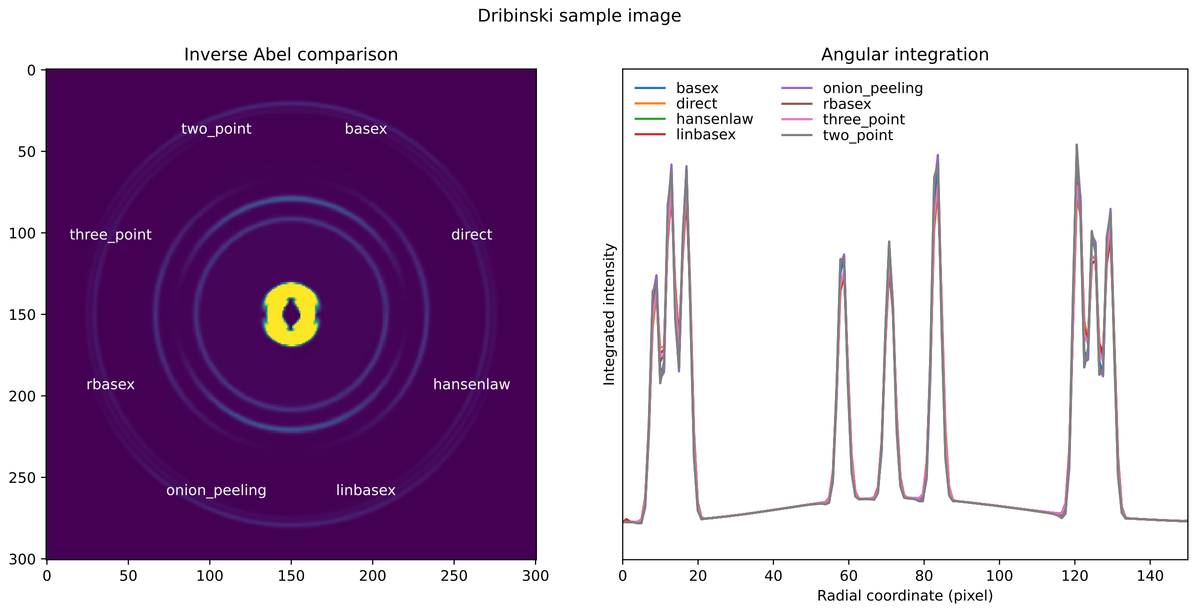
(continued from previous page)

```
transforms = [
    "basex",
    "direct",
    "hansenlaw",
    "linbasex",
    "onion_peeling",
    "rbasex",
    "three_point",
    "two_point",
]
# number of transforms:
ntrans = len(transforms)

# sample image radius in pixels
R = 150

fIM = abel.tools.analytical.SampleImage(n=2 * R + 1, name="Dribinski").abel
# fIM += np.random.normal(0, 5000, fIM.shape)  # try adding some noise

print("image shape {}".format(fIM.shape))

# sectors for combining output images (clockwise from top)
row = np.arange(-R, R + 1)[:, None]
col = np.arange(-R, R + 1)
sector = np.asarray(ntrans * (1 - np.arctan2(col, row) / np.pi) / 2, dtype=int)

# apply each method --------------------

IM = np.zeros_like(fIM)  # for inverse Abel transformed images
ymax = 0  # max. speed distribution

for i, method in enumerate(transforms):
    print("\n------- {:s} inverse ...".format(method))
    t0 = time()

    # inverse Abel transform using 'method'
    recon = abel.Transform(fIM, method=method, direction="inverse",
                           symmetry_axis=(0, 1)).transform

    print("                    {:.4f} s".format(time()-t0))

    # copy sector to combined output image
    idx = sector == i
    IM[idx] = recon[idx]

    # method label for each quadrant
    annot_angle = 2 * np.pi * (0.5 + i) / ntrans
    annot_coord = (R + 0.8 * R * np.sin(annot_angle),
                   R - 0.8 * R * np.cos(annot_angle))
    ax1.annotate(method, annot_coord, color="white", ha="center")

    # polar projection and speed profile
```

(continues on next page)

(continued from previous page)

```
    radial, speed = abel.tools.vmi.angular_integration_3D(recon)

    # plot speed distribution
    ax2.plot(radial, speed, label=method)

    # update limit
    ymax = max(ymax, speed.max())

plt.suptitle('Dribinski sample image')

ax1.set_title('Inverse Abel comparison')
vmax = IM[:, R+2:].max()  # ignoring pixels near center line
ax1.imshow(IM, vmin=0, vmax=0.1 * vmax)

ax2.set_title('Angular integration')
ax2.set_xlabel('Radial coordinate (pixel)')
ax2.set_xlim(0, 150)
ax2.set_ylabel('Integrated intensity')
ax2.set_ylim(-0.1 * ymax, 1.2 * ymax)
ax2.set_yticks([])
ax2.legend(ncol=2, labelspacing=0.1, frameon=False)

plt.tight_layout()
# plt.savefig('plot_example_all_dribinski.png', dpi=100)
plt.show()
```



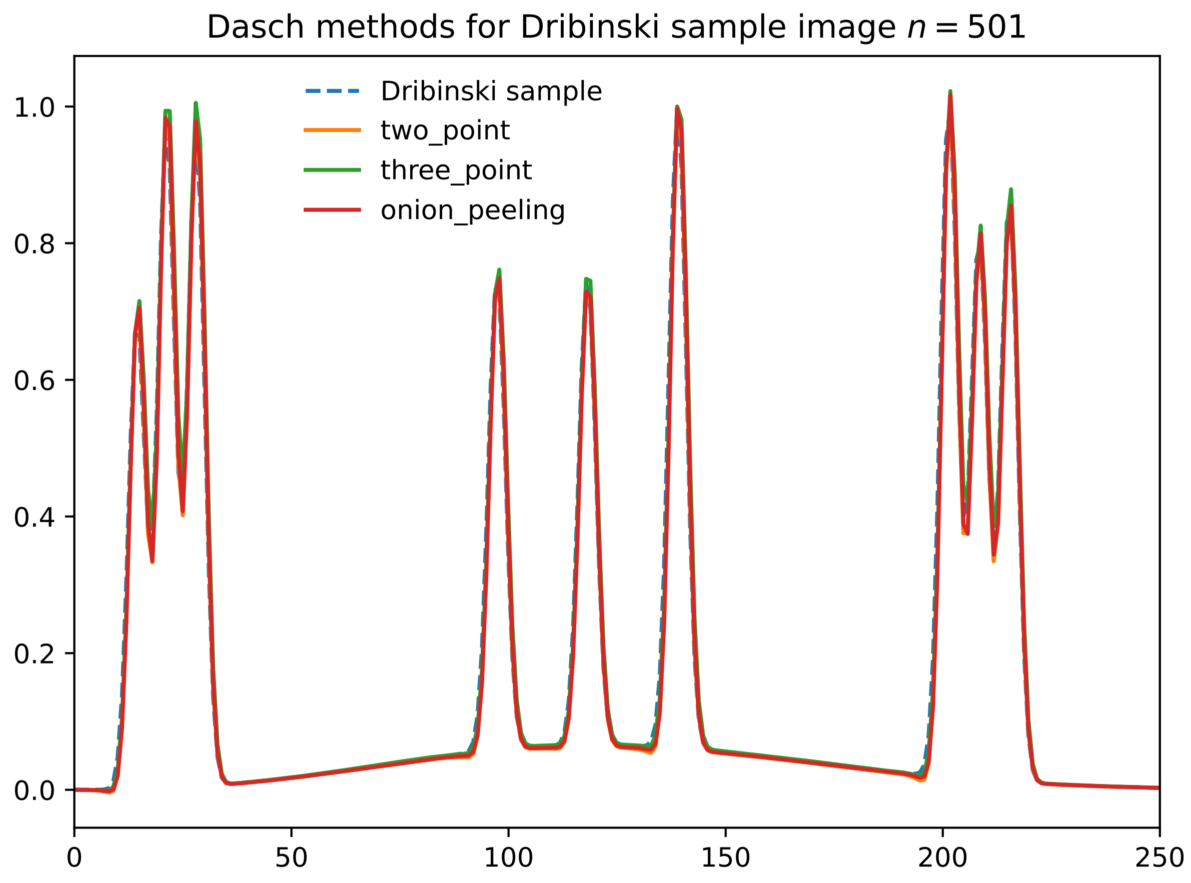
7.9 Example: Dasch methods

```
# -*- 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).func

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

# speed distribution of original image
orig_speed = abel.tools.vmi.angular_integration_3D(origQ[0], origin=(-1, 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)
# speed distribution
    speed = abel.tools.vmi.angular_integration_3D(AQ0, origin=(-1, 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))
```

(continues on next page)

(continued from previous page)

```
plt.xlim((0, 250))
plt.legend(loc='upper center', bbox_to_anchor=(0.35, 1), frameon=False)
plt.tight_layout()
# plt.savefig("plot_example_dasch_methods.png",dpi=100)
plt.show()
```



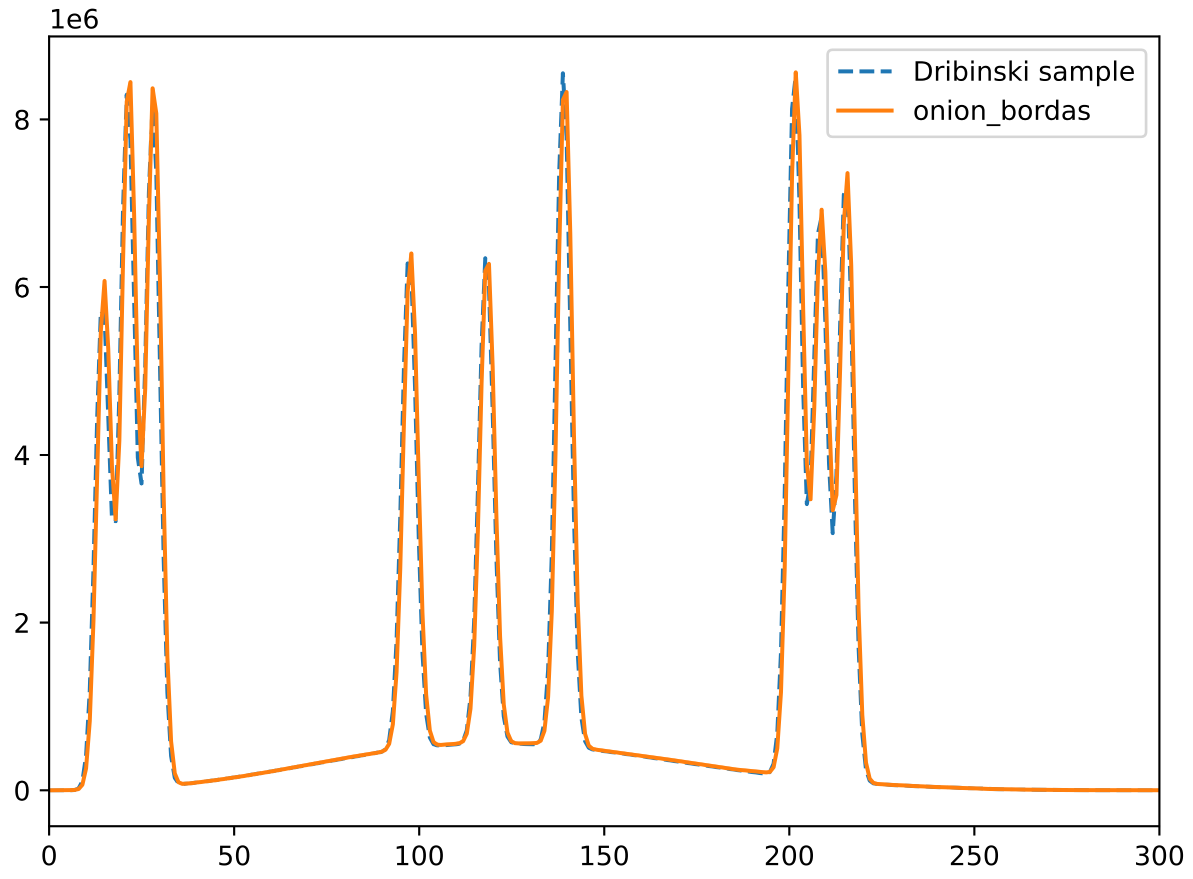
7.10 Example: Onion Bordas

```
# -*- 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).func
```

(continues on next page)

(continued from previous page)

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

# speed distribution
orig_speed = abel.tools.vmi.angular_integration_3D(origQ[0], origin=(-1, 0))

# 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_3D(borQ0, origin=(-1, 0))

plt.plot(*orig_speed, linestyle='dashed', label="Dribinski sample")
plt.plot(bor_speed[0], bor_speed[1], label="onion_bordas")
plt.xlim((0, 300))
plt.legend(loc=0)
plt.tight_layout()
# plt.savefig("plot_example_onion_bordas.png",dpi=100)
plt.show()
```



7.11 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')
```

(continues on next page)

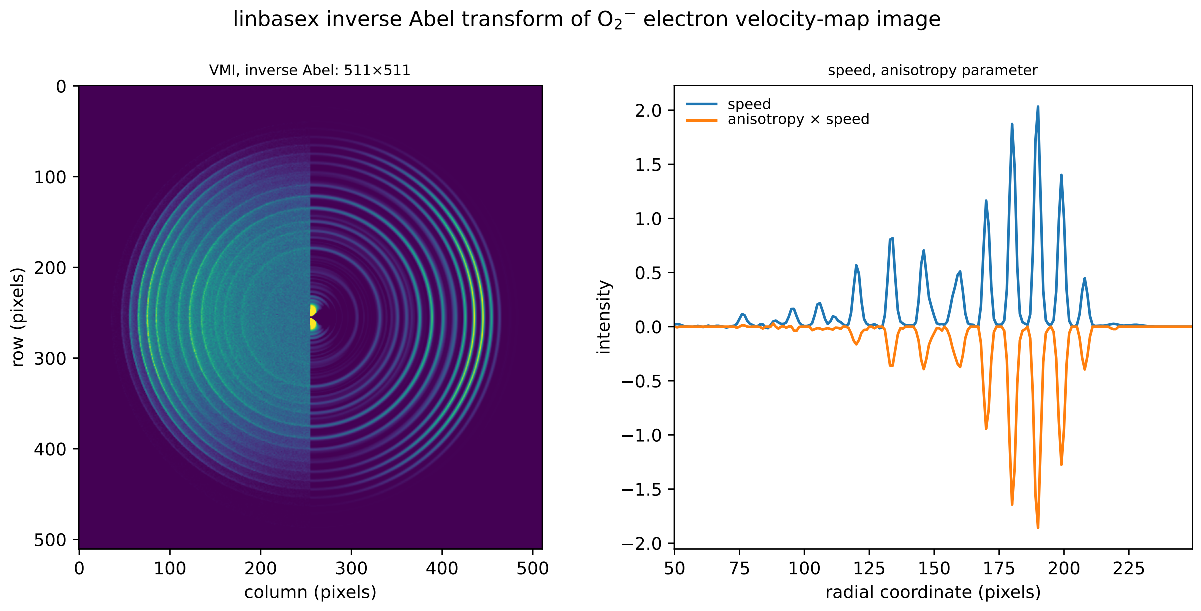
(continued from previous page)

```
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,
# `origin="convolution"` takes care of this
IM = abel.tools.center.center_image(IM, origin="convolution", square=True)

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
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",
                     transform_options=dict(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=(10, 5))
ax1 = plt.subplot(121)
ax2 = plt.subplot(122)

# 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', vmin=0, vmax=vmax)
```

(continues on next page)

(continued from previous page)

```
ax1.set_xlabel('column (pixels)')
ax1.set_ylabel('row (pixels)')
ax1.set_title('VMI, inverse Abel: {:d}×{: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)
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(r'linbasex inverse Abel transform of O$_{2}{}^{-}$ electron '
             'velocity-map image',
             fontsize='larger')

plt.tight_layout()

# Save a image of the plot
# plt.savefig("plot_example_linbasex.png", dpi=100)

# Show the plots
plt.show()
```



7.12 Example: rBasex beam block

```
from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt
from copy import copy

import abel
from abel.tools.analytical import SampleImage
from abel.tools.vmi import rharmonics
from abel.rbasex import rbasex_transform

# This example demonstrates analysis of velocity-map images with "damaged"
# areas, in this case, with some parts obstructed by a beam block (see
# https://aip.scitation.org/doi/10.1063/1.4921990 for a practical example).
# First, a general Abel-transform method is used naively to demonstrate
# artifacts produced in the reconstructed image and the extracted radial
# distributions.
# Second, radial distributions are extracted from the same reconstructed image,
# but with its artifacts masked, showing good agreement with the actual
# distributions.
# Third, the rBasex method is used to transform the initial damaged image with
# the experimental artifacts masked, yielding a correct and cleaner
# reconstructed image and correct reconstructed distributions.

R = 150  # image radius
N = 2 * R + 1  # full image width and height
block_r = 20  # beam-block disk radius
block_w = 5  # beam-block holder width

vlim = 3.6  # intensity limits for images
ylim = (-1.3, 2.7)  # limits for plots

# create source distribution and its profiles for reference
source = SampleImage(N).func / 100
r_src, P0_src, P2_src = rharmonics(source)

# simulate experimental image:
# projection
im, _ = rbasex_transform(source, direction='forward')
# Poissonian noise
im[im < 0] = 0
im = np.random.RandomState(0).poisson(im)
# image coordinates
im_x = np.arange(float(N)) - R
im_y = R - np.arange(float(N))[:, None]
im_r = np.sqrt(im_x**2 + im_y**2)
# simulate beam-block shadow
im = im / (1 + np.exp(-(im_r - block_r)))
im[:R] *= 1 / (1 + np.exp(-(np.abs(im_x) - block_w)))
```

(continues on next page)

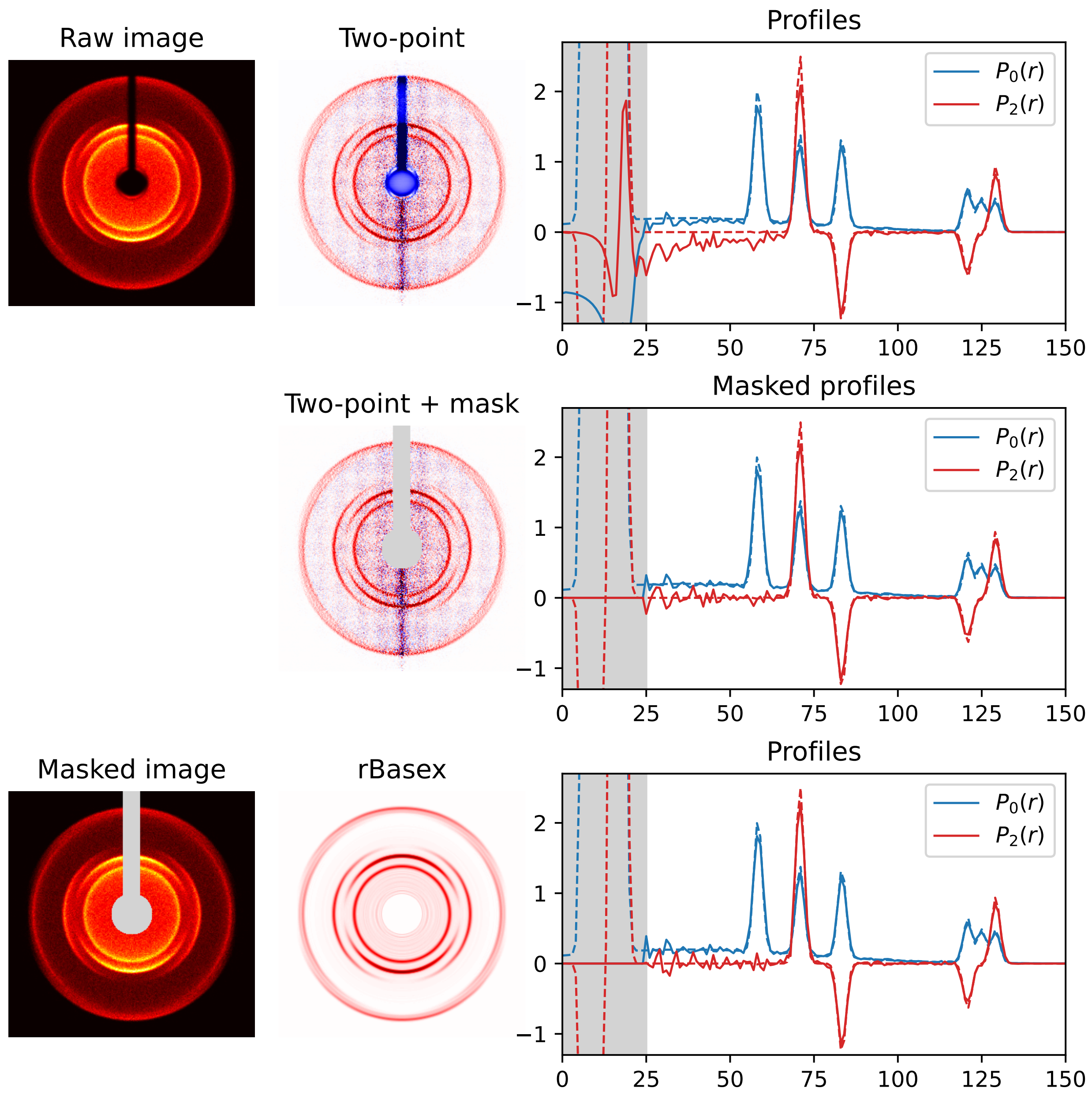
(continued from previous page)

```
# create mask that fully covers beam-block shadow
mask_r = block_r + 5
mask_w = block_w + 5
mask = np.ones_like(im)
mask[im_r < mask_r] = 0
mask[:R, R-mask_w:R+mask_w] = 0

# reconstruct "as is" by a general Abel-transform method
rec_abel = abel.Transform(im, method='two_point').transform
# extract profiles "as is"
r_abel, P0_abel, P2_abel = rharmonics(rec_abel)
# extract profiles from masked reconstruction
r_abel_masked, P0_abel_masked, P2_abel_masked = rharmonics(rec_abel, weights=mask)

# reconstruct masked image with rBasex
rec_rbasex, distr_rbasex = rbasex_transform(im, weights=mask)
r_rbasex, P0_rbasex, P2_rbasex = distr_rbasex.rharmonics()

# plotting...
plt.figure(figsize=(7, 7))

cmap_hot = copy(plt.cm.hot)
cmap_hot.set_bad('lightgray')
cmap_seismic = copy(plt.cm.seismic)
cmap_seismic.set_bad('lightgray')

def plots(row,
          im_title, im, im_mask,
          tr_title, tr, tr_mask,
          pr_title, r, P0, P2):
    # input image
    if im is not None:
        plt.subplot(3, 4, 4 * row + 1)
        plt.title(im_title)
        im_masked = np.ma.masked_where(im_mask == 0, im)
        plt.imshow(im_masked, cmap=cmap_hot)
        plt.axis('off')

    # transformed image
    plt.subplot(3, 4, 4 * row + 2)
    plt.title(tr_title)
    tr_masked = np.ma.masked_where(tr_mask == 0, tr)
    plt.imshow(tr_masked, vmin=-vlim, vmax=vlim, cmap=cmap_seismic)
    plt.axis('off')

    # profiles
    plt.subplot(3, 2, 2 * row + 2)
    plt.title(pr_title)
    plt.axvspan(0, mask_r, color='lightgray')  # shade region without valid data
    plt.plot(r_src, P0_src, 'C0--', lw=1)
    plt.plot(r_src, P2_src, 'C3--', lw=1)
    plt.plot(r, P0, 'C0', lw=1, label='$P_0(r)$')
```

(continues on next page)

(continued from previous page)

```
    plt.plot(r, P2, 'C3', lw=1, label='$P_2(r)$')
    plt.xlim((0, R))
    plt.ylim(ylim)
    plt.legend()

plots(0,
      'Raw image', im, None,
      'Two-point', rec_abel, None,
      'Profiles', r_abel, P0_abel, P2_abel)

plots(1,
      None, None, None,
      'Two-point + mask', rec_abel, mask,
      'Masked profiles', r_abel_masked, P0_abel_masked, P2_abel_masked)

plots(2,
      'Masked image', im, mask,
      'rBasex', rec_rbasex, None,
      'Profiles', r_rbasex, P0_rbasex, P2_rbasex)

plt.subplots_adjust(left=0.01, right=0.97, wspace=0.1,
                    bottom=0.04, top=0.96, hspace=0.3)
plt.show()
```



7.13 Example: rBasex multimass

```
from __future__ import division, print_function

import numpy as np
import matplotlib.pyplot as plt

import abel
from abel.rbasex import rbasex_transform

# This example demonstrates analysis of partially overlapping velocity-map
```

(continues on next page)

(continued from previous page)

```
# images, which might be useful for the "multimass imaging" method (see
# https://pubs.acs.org/doi/10.1021/jp053143m).
# Reconstruction of each part of the combined image uses zero weights for all
# pixels not belonging to that part or contaminated by signals from other
# parts in order to extract distributions pertaining to that part only.
# Notice that the central image in this example cannot be correctly
# reconstructed by "usual" Abel-transform methods, since all its quadrants are
# contaminated. However, the remaining uncontaminated radial and angular ranges
# are sufficient for rBasex to reconstruct the complete distributions.

# Create an artificial sample image from experimental and synthetic images.
# central image
im2 = np.loadtxt('data/O2-ANU1024.txt.bz2')
from scipy.ndimage import zoom
im2 = zoom(im2, 0.75)  # (resized for better visibility)
h2, w2 = im2.shape
# right image
im3 = np.loadtxt('data/VMI_art1.txt.bz2')
im3 *= 2  # (intensified for better visibility)
h3, w3 = im3.shape
# left image
h1 = w1 = min(h3, w3)
r1 = h1 // 2
x = np.linspace(-r1, r1, w1)
im1 = np.random.RandomState(0).poisson(200 * np.exp(-(x**2 + x[:, None]**2) /
                                       (r1 / 3)**2))
# assemble the whole image
h, w = max(h2, h3), w2 + w3
im = np.zeros((h, w))
row1, col1 = (h - h1) // 2, 0
im[row1:row1+h1, col1:col1+w1] += im1
row2, col2 = (h - h2) // 2, (w - w2) // 2
im[row2:row2+h2, col2:col2+w2] += im2
row3, col3 = (h - h3) // 2, w - w3
im[row3:row3+h3, col3:col3+w3] += im3

# Origins and maximal radii for each part
# (in reality they will need to be determined from the data somehow; also,
# in practice it would be better to cut the whole image into parts and work
# with them separately, which is not done here to simplify the code).
# for left image
origin1 = (h // 2 - 1, w1 // 2)
r1 = min(h1, w1) // 2
# for central image
origin2 = (h // 2, w // 2)
r2 = min(h2, w2) // 2 - 50
# for right image
origin3 = (h // 2 - 1, w - w3 // 2 - 1)
r3 = min(h3, w3) // 2

# Create "masks" for each part with unit weights for "good" pixels and zero
# weights for "bad" pixels.
```

(continues on next page)

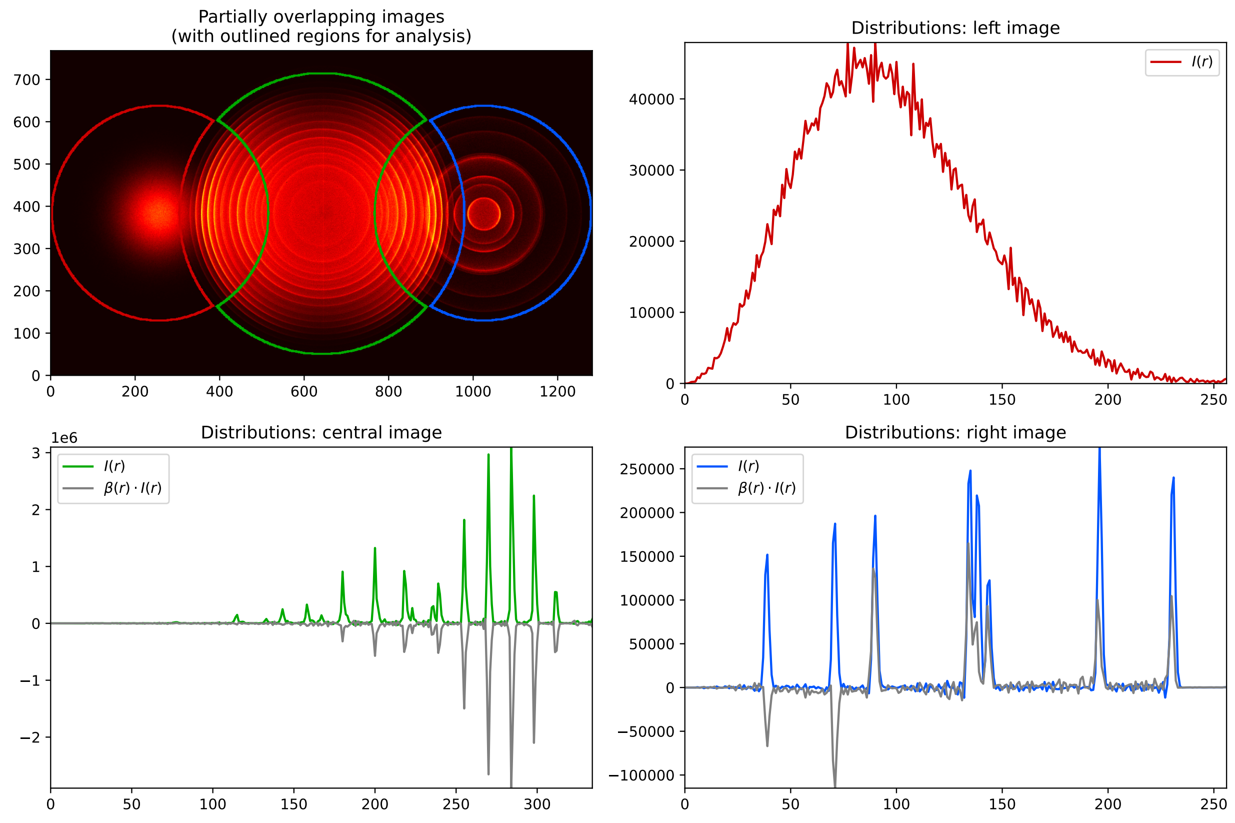
(continued from previous page)

```
# coordinates relative to each origin
x1, y1 = abel.tools.polar.index_coords(im, origin=origin1)
x2, y2 = abel.tools.polar.index_coords(im, origin=origin2)
x3, y3 = abel.tools.polar.index_coords(im, origin=origin3)
# for left image (include left, exclude central)
mask1 = np.array((x1**2 + y1**2 < r1**2) *  # inside radius r1 from origin1 and
                 (x2**2 + y2**2 > r2**2),   # outside radius r2 from origin2
                 dtype=float)
# for central image (include central, exclude left and right)
mask2 = np.array((x2**2 + y2**2 < r2**2) *  # inside radius r2 from origin2 and
                 (x1**2 + y1**2 > r1**2) *  # outside radius r1 from origin1 and
                 (x3**2 + y3**2 > r3**2),   # outside radius r3 from origin3
                 dtype=float)
# for right image (include right, exclude central)
mask3 = np.array((x3**2 + y3**2 < r3**2) *  # inside radius r3 from origin3 and
                 (x2**2 + y2**2 > r2**2),   # outside radius r2 from origin2
                 dtype=float)

fig = plt.figure(figsize=(12, 8))

# Show the whole image
plt.subplot(221)
plt.title('Partially overlapping images\n'
          '(with outlined regions for analysis)')
plt.imshow(im, cmap='hot')
# overlay with the boundaries of each mask (only for demonstration)
from scipy.ndimage import binary_erosion
brush = np.ones((11, 11))
dmask = 1 * (mask1 - binary_erosion(mask1, structure=brush)) + \
        2 * (mask2 - binary_erosion(mask2, structure=brush)) + \
        3 * (mask3 - binary_erosion(mask3, structure=brush))
dmask[dmask == 0] = np.nan
from matplotlib.colors import ListedColormap
rgb = ListedColormap(['#CC0000', '#00AA00', '#0055FF'])
plt.imshow(dmask, extent=(0, w, 0, h), cmap=rgb, interpolation='nearest')

# Analyze the left part and plot results
plt.subplot(222)
plt.title('Distributions: left image')
# the reconstructed image is not used in this example, so it is not created;
# also notice that order=0 is enough for this totally isotropic case
_, distr = rbasex_transform(im, origin=origin1, rmax=r1, order=0,
                            weights=mask1, out=None)
r, I = distr.rIbeta()
plt.plot(r, I, c=rgb(0), label='$I(r)$')
plt.legend()
plt.autoscale(enable=True, tight=True)

# Analyze the central part and plot results
plt.subplot(223)
plt.title('Distributions: central image')
# here the default order=2 is needed and used
```

(continues on next page)

(continued from previous page)

```
_, distr = rbasex_transform(im, origin=origin2, rmax=r2, weights=mask2, out=None)
r, I, beta = distr.rIbeta()
plt.plot(r, I, c=rgb(1), label='$I(r)$')
# beta(r) I(r) is the "speed distribution" of P_2(r)
plt.plot(r, beta * I, c='gray', label='$\\beta(r) \\cdot I(r)$')
plt.legend()
plt.autoscale(enable=True, tight=True)

# Analyze the right part and plot results
plt.subplot(224)
plt.title('Distributions: right image')
_, distr = rbasex_transform(im, origin=origin3, rmax=r3, weights=mask3, out=None)
r, I, beta = distr.rIbeta()
plt.plot(r, I, c=rgb(2), label='$I(r)$')
plt.plot(r, I * beta, c='gray', label='$\\beta(r) \\cdot I(r)$')
plt.legend()
plt.autoscale(enable=True, tight=True)

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



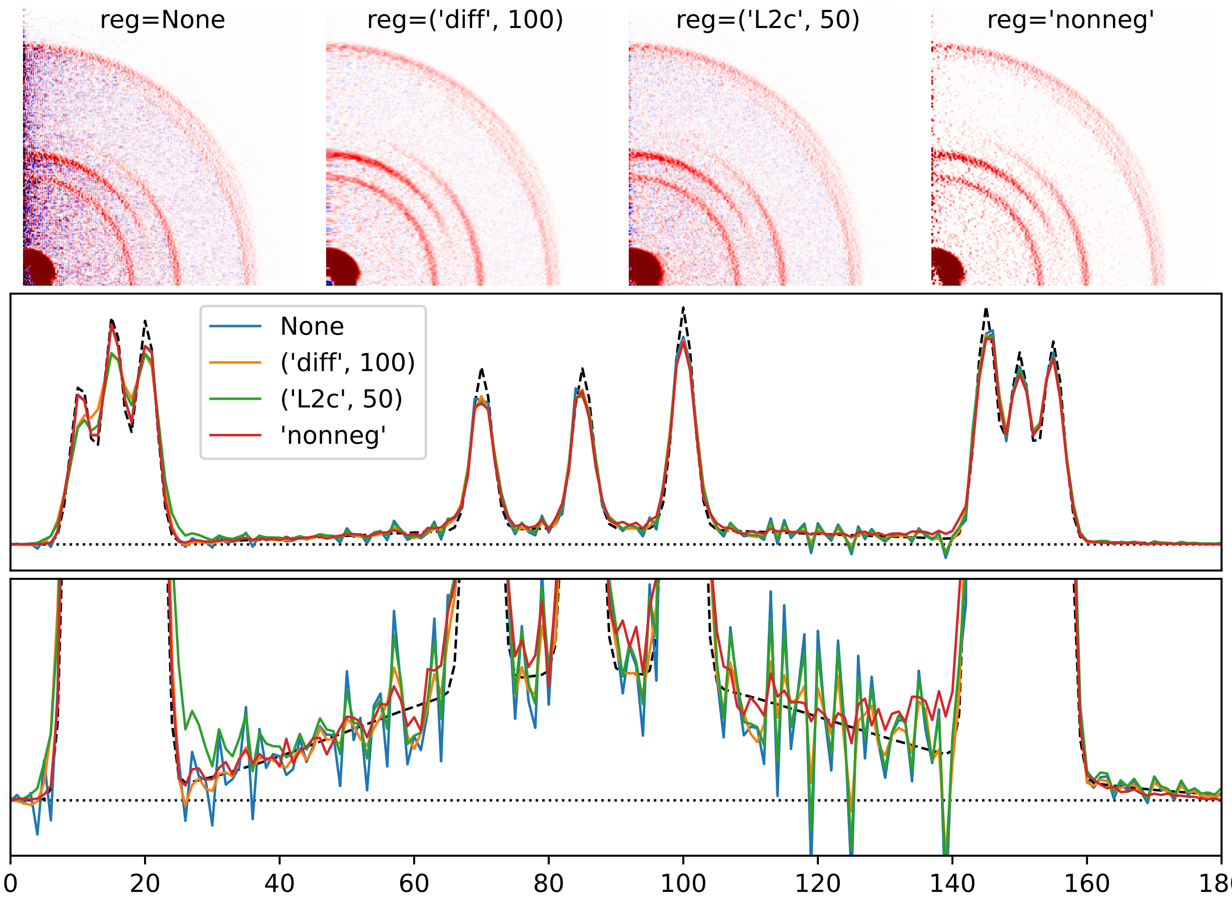
7.14 Example: Daun regularizations

```
# -*- coding: UTF-8 -*-
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import abel

# one quadrant of Dribinski sample image, its size and intensity distribution
im = abel.tools.analytical.SampleImage().func
q0 = abel.tools.symmetry.get_image_quadrants(im)[0]
n = q0.shape[0]
I0, _ = abel.tools.vmi.Ibeta(q0, origin='ll')

# forward-transformed quadrant
Q = abel.rbasex.rbasex_transform(q0, origin='ll', direction='forward')[0]
# rescale intensities to 1000 max
scale = 1000 / Q.max()
q0 *= scale
I0 *= scale
Q *= scale
# add Poissonian noise
Q = np.random.RandomState(0).poisson(Q)

# regularization parameters
regs = [None,
        ('diff', 100),  # RMS optimal is ~50
        ('L2c', 50),    # RMS optimal is ~25
        'nonneg']

# array for corresponding intensity distributions
Is = []

plt.figure(figsize=(7, 5))

# transformed images
for i, reg in enumerate(regs):
    plt.subplot(3, 4, 1 + i)
    plt.axis('off')

    q = abel.daun.daun_transform(Q, degree=1, reg=reg)
    plt.imshow(q, clim=(-3, 3), cmap='seismic')
    plt.text(n / 2, 0, 'reg=' + repr(reg), ha='center', va='top')

    I, _ = abel.tools.vmi.Ibeta(q, origin='ll')
    Is.append(I)

# plots of intensity distributions
for plot in [2, 3]:
    plt.subplot(3, 1, plot)
    plt.axhline(0, c='k', ls=':', lw=1)
    plt.plot(I0, '--k', lw=1)
```

(continues on next page)

(continued from previous page)

```
    for i, reg in enumerate(regs):
        plt.plot(Is[i], lw=1, label=repr(reg))

    plt.xlim((0, n - 1))
    plt.yticks([])
    if plot == 2:  # full y range
        plt.xticks([])
        plt.legend(loc='upper center', bbox_to_anchor=(0.25, 1))
    else:  # magnified
        plt.ylim((-2000, 8000))

plt.subplots_adjust(left=0.01, right=0.98,
                    bottom=0.05, top=1,
                    wspace=0, hspace=0.03)
plt.show()
```



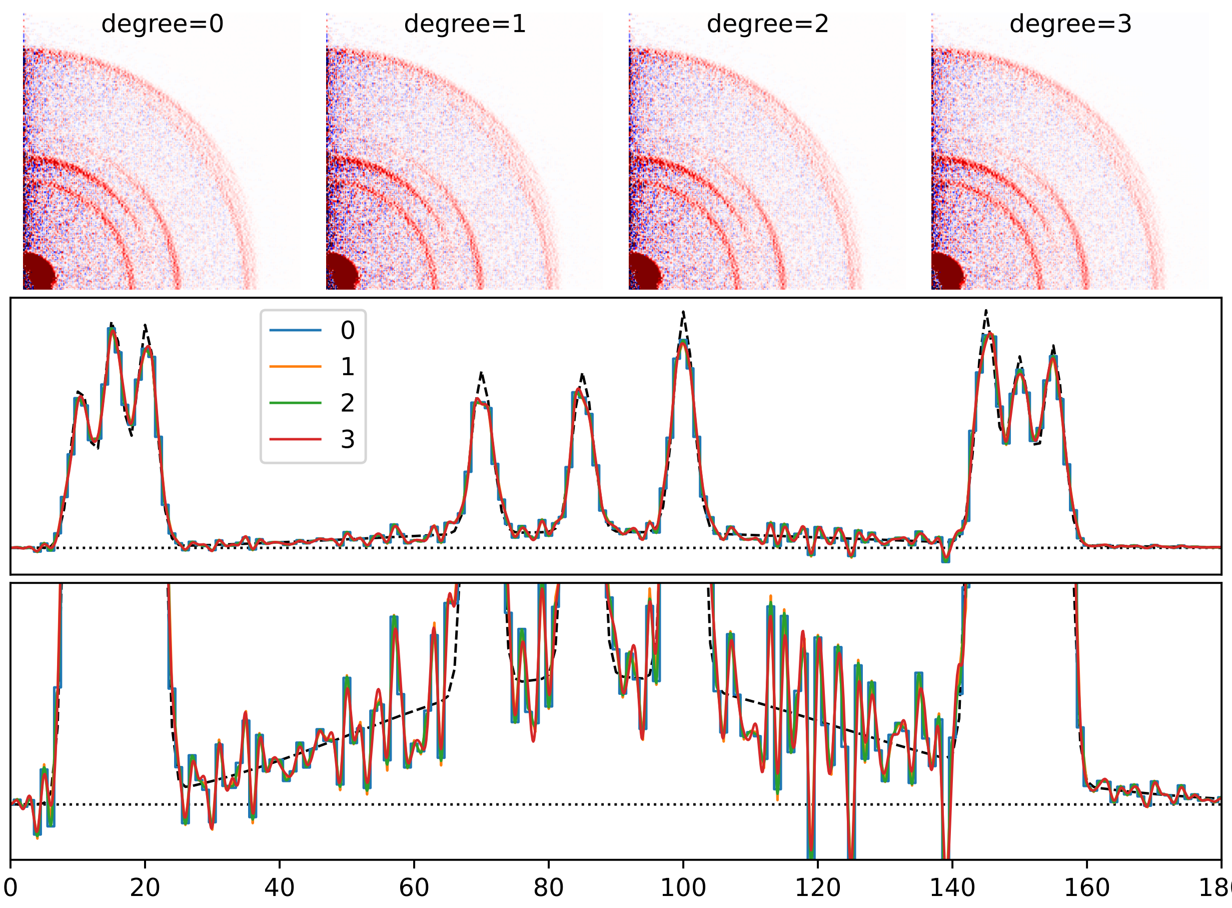
7.15 Example: Daun degree

```
# -*- coding: UTF-8 -*-
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline, splev
import abel

# one quadrant of Dribinski sample image, its size and intensity distribution
im = abel.tools.analytical.SampleImage().func
q0 = abel.tools.symmetry.get_image_quadrants(im)[0]
n = q0.shape[0]
I0, _ = abel.tools.vmi.Ibeta(q0, origin='ll')

# forward-transformed quadrant
Q = abel.rbasex.rbasex_transform(q0, origin='ll', direction='forward')[0]
# rescale intensities to 1000 max
scale = 1000 / Q.max()
q0 *= scale
I0 *= scale
Q *= scale
# add Poissonian noise (comment out to compare the clean transform results)
Q = np.random.RandomState(0).poisson(Q)

# array for corresponding intensity distributions
Is = []

plt.figure(figsize=(7, 5))

# transformed images
for degree in range(4):
    plt.subplot(3, 4, 1 + degree)
    plt.axis('off')

    q = abel.daun.daun_transform(Q, degree=degree)
    plt.imshow(q, clim=(-3, 3), cmap='seismic')
    plt.text(n / 2, 0, 'degree=' + str(degree), ha='center', va='top')

    I, _ = abel.tools.vmi.Ibeta(q, origin='ll')
    Is.append(I)

# pixel subdivisions for smooth plots
sub = 10
rsub = np.linspace(0, n, sub * n + 1)

# plots of intensity distributions
for plot in [2, 3]:
    plt.subplot(3, 1, plot)
    plt.axhline(0, c='k', ls=':', lw=1)
    plt.plot(I0, '--k', lw=1)
```

(continues on next page)

(continued from previous page)

```
    # degree = 0: plot with steps
    plt.step(np.arange(n), Is[0], lw=1, label='0', where='mid')
    # degree = 1: plot with lines
    plt.plot(Is[1], lw=1, label='1')
    # degree = 2: plot using parabolic segments
    r1, r2 = np.arange(sub // 2) / sub, np.arange(sub // 2, 0, -1) / sub
    b = np.concatenate((2 * r1**2, 1 - 2 * r2**2, 1 - 2 * r1**2, 2 * r2**2))
    I = np.zeros_like(rsub)
    for m in range(1, n):
        i0 = sub * m
        I[i0 - sub: i0 + sub] += Is[2][m] * b
    plt.plot(rsub, I, lw=1, label='2')
    # degree = 3: plot with cubic splines
    spl = make_interp_spline(np.arange(n), Is[3], bc_type='clamped')
    plt.plot(rsub, splev(rsub, spl), lw=1, label='3')

    plt.xlim((0, n - 1))
    plt.yticks([])
    if plot == 2:  # full y range
        plt.xticks([])
        plt.legend(loc='upper center', bbox_to_anchor=(0.25, 1))
    else:  # magnified
        plt.ylim((-2000, 8000))

plt.subplots_adjust(left=0.01, right=0.98,
                    bottom=0.05, top=1,
                    wspace=0, hspace=0.03)
plt.show()
```



7.16 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)
```

(continues on next page)

(continued from previous page)

```
# === linbasex transform ===================================
proj_angles = np.arange(0, 180, 10)/180*np.pi  # projection angles in 10° 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', origin='slice',
                     center_options=dict(square=True),
                     transform_options=dict(
                         proj_angles=proj_angles,
                         radial_step=radial_step, smoothing=smoothing,
                         threshold=threshold, clip=clip,
                         verbose=True))


# === Hansen & Law inverse Abel transform ==================
HIM = abel.Transform(IM, origin="slice", method="hansenlaw",
                     symmetry_axis=None, angular_integration=True)

# speed distribution
radial, speed = HIM.angular_integration

# 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, radial_ranges=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
```

(continues on next page)

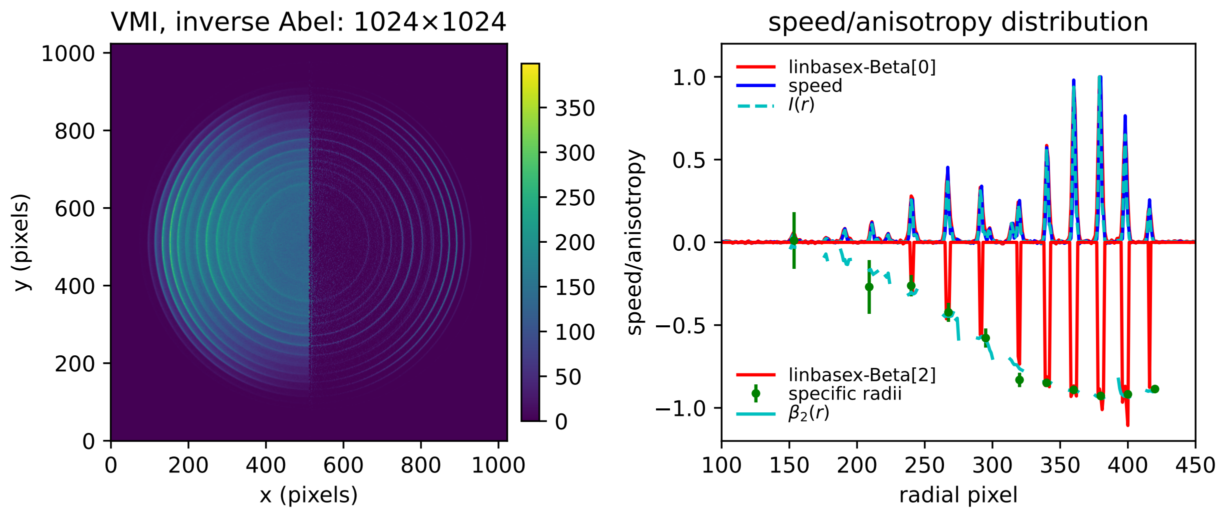
(continued from previous page)

```
# plots of the analysis
fig = plt.figure(figsize=(8, 3.5))
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', 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')
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.tight_layout()

# Save a image of the plot
# plt.savefig("plot_example_PAD.png", dpi=100)

# Show the plots
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



7.17 Example: Circularize image

```
import numpy as np
import matplotlib.pyplot as plt
import abel
from abel.tools.circularize import circularize, circularize_image
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).func

# 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
```

(continues on next page)

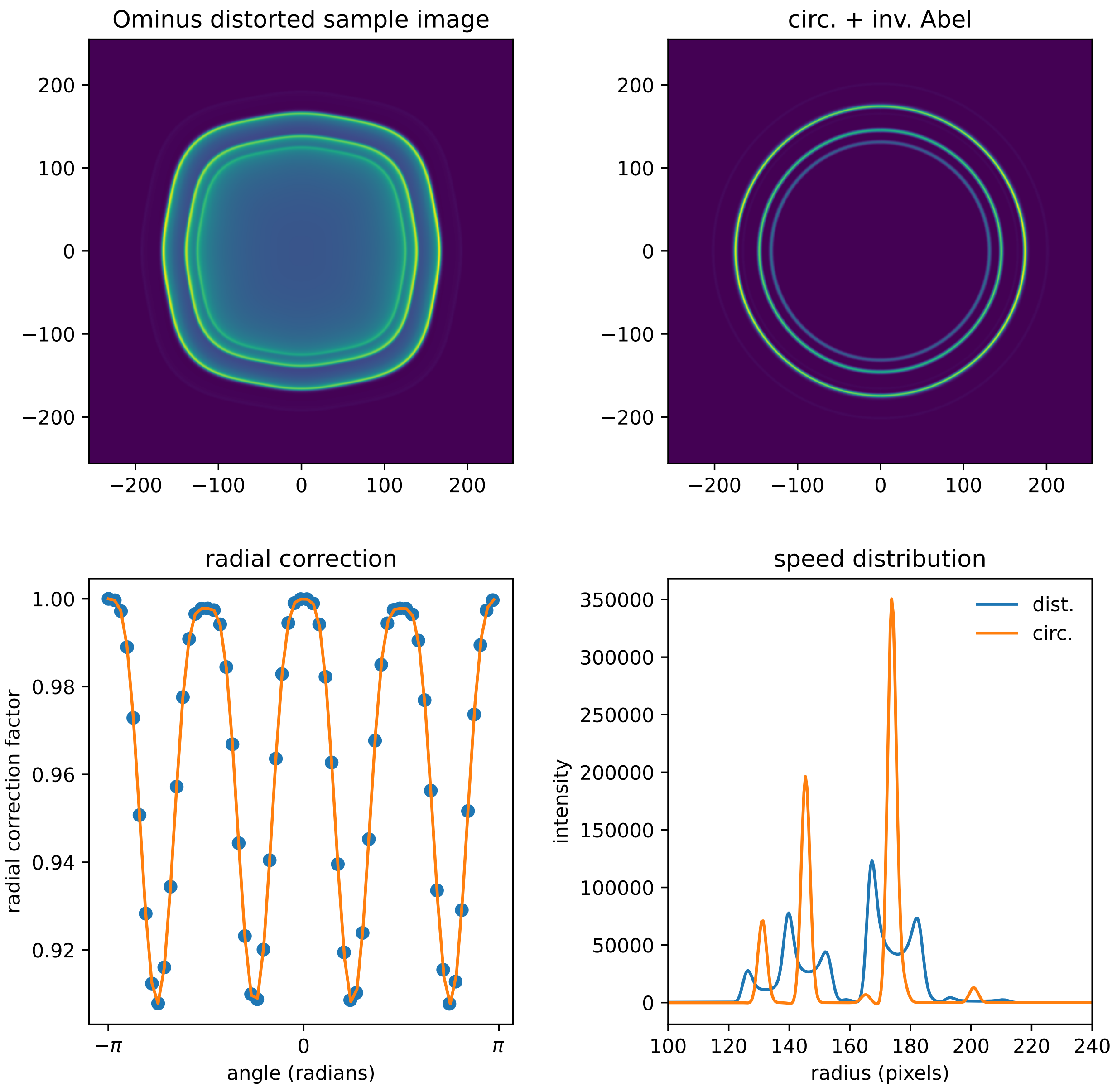
(continued from previous page)

```
IMdist = circularize(IMf, radial_correction_function=flower_scaling)

# circularize ------------
IMcirc, sla, sc, scspl = circularize_image(IMdist,
                                          method='lsq', dr=0.5, dt=0.1,
                                          tol=0, return_correction=True)

# inverse Abel transform for distored and circularized images ---------
AIMdist = abel.Transform(IMdist, method="three_point").transform
AIMcirc = abel.Transform(IMcirc, method="three_point").transform

# respective speed distributions
rdist, speeddist = abel.tools.vmi.angular_integration_3D(AIMdist, dr=0.5)
rcirc, speedcirc = abel.tools.vmi.angular_integration_3D(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, origin='lower', extent=extent)
axs[0, 0].set_title("Ominus distorted sample image")

axs[0, 1].imshow(AIMcirc, vmin=0, 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.tight_layout(h_pad=2, w_pad=2)
# plt.savefig("plot_example_circularize_image.png", dpi=75)
```

(continues on next page)

(continued from previous page)

```
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 Rebasing

If possible, before submitting 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.

8.2 Code style

We hope that the PyAbel code will be understandable, hackable, and maintainable for many years to come. So, please use good coding style, include plenty of comments, use docstrings for functions, and pick informative variable names.

PyAbel attempts to follow PEP8 style whenever possible, since the PEP8 recommendations typically produces code that is easier to read. You can check your code using pycodestyle, which can be called from the command line or incorporated right into most text editors. Also, PyAbel is using automated pycodestyle checking of all pull requests using pep8speaks. However, producing readable code is the primary goal, so please go ahead and break the rules of PEP8 when doing so improves readability. For example, if a section of your code is easier to read with lines slightly longer than 79 characters, then use the longer lines.

8.3 Unit tests

Before submitting a 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
```

8.4 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 and the sphinx_rtd_theme. You can install them using

```
pip install sphinx
pip install sphinx_rtd_theme
```

Once you have these packages installed, 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.

8.5 Changelog

If the change is significant (more than just a typo-fix), please leave a short note about the change in CHANGELOG.rst, at the bottom of the "Unreleased" section (the PR number can be added later).

8.6 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 ensure an overall code quality, please consider the following points in your pull request.

8.6.1 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)

8.6.2 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 with all 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.

8.7 Dependencies

The current list of dependencies can be found in `setup.py`. Please refrain from adding new dependencies, unless it cannot be avoided.

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`.

8.9 For maintainers: Releasing a new version

First, make a pull request that does the following:

- Increment the version number in `abel/_version.py`.
- Update `CHANGELOG.rst` by renaming the "Unreleased" section to the new version and the expected release date.
- Use the changelog to write version release notes that can be included as a comment in the PR and will be used later.
- Update copyright years in `doc/conf.py`.

After the PR is merged:

- Press the "Draft a new release" button on the Releases page and create a new tag, matching the new version number (for example, "v1.2.3" for version "1.2.3").
- Copy and paste the release notes from the PR into the release notes.
- Release it!
- PyAbel should be automatically released on PyPI (see PR #161).
- Check that the new package is on PyPI.
- Check that the new docs are on Read the Docs.
- Check that the new version is on Zenodo.
- A bot should automatically make a PR on the conda-forge repo. This takes a while and needs to be merged manually.
- Check that the new conda packages are on Anaconda.org.

Chapter 9

Changelog

v0.9.0 (2022-12-14)

- Correct behavior of relative basis_dir in basex under Python 2 (PR #336).
- Improvements in tools.analytical.SampleImage class: more consistent and intuitive interface, accurate Abel transform for existing images, additional sample images with exact Abel transform (PR #339, #352).
- Analytical Abel transform of axially symmetric piecewise polynomials in spherical coordinates (PR #339).
- Offline HTML and PDF documentation is now available at Read the Docs and can be built locally (PR #343, #348, #349).
- **Important** changes in the **lin-BASEX** implementation (PR #357):
  - It was erroneously producing even-sized images with offset origin. Now the output is odd-sized and properly centered.
  - Output images for radial_step > 1 were scaled down by the same factor. Now they have the save size as the input.
  - The sign of Beta for odd Legendre orders is reversed to be consistent with PyAbel conventions (zero angle is upwards, towards smaller row indices).
- Transform() with method='linbasex' now always stores the radial, Beta and projection attributes, so there is no need to pass return_Beta=True in transform_options (PR #357).

v0.8.5 (2022-01-21)

- New functions in tools.vmi for angular integration and averaging, replacing angular_integration() and average_radial_intensity(), which had incorrect or nonintuitive behavior (PR #318, PR #319).
- Avoid unnecessary calculations in transform.Transform() for the symmetry_axis=(0, 1) case (PR #324).
- New method by Daun et al. and its extensions (PR #326).
- Basis sets are now by default stored in a single system-specific directory, see get_basis_dir() and set_basis_dir() in abel.transform (PR #327). **Important!** The current working directory is no longer used by default for loading basis sets. It is recommended to execute

  ```
  import abel; print(abel.transform.get_basis_dir(make=True))
  ```

  and move all existing basis sets to the reported directory.

- Cython optimization flags are changed to make conda packages compatible with all CPUs and to improve the direct_C method performance (PR #331). Bitwise reproducibility of direct_C transforms might be affected.

v0.8.4 (2021-04-15)

- Added odd angular orders to tools.vmi.Distributions (PR #266).
- **Important!** Some "center" functions/parameters are renamed to "origin" or "method"; using old names still works but will print deprecation warnings, please update your code accordingly. Image origin is now always in the (row, column) format for consistency within PyAbel and with NumPy/SciPy; this can break some code, so please check carefully and update it if necessary. See PR #267.
- Fixed the GUI examples (example_GUI.py and example_simple_GUI.py) so that they work with the lastest versions of tk (PR #269).
- New method rBasex for velocity-map images, based on pBasex and the work of Mikhail Ryazanov (PR #270).
- Added "orders", "sinpowers" and "valid" to tools.vmi.Distributions results, reordered cossin() powers for consistency (PR #270).
- Improved tools.vmi.Distributions performance on Windows (PR #270).
- More corrections to the GUI example: working without the "bases" directory, loading "from transform", interface enhancements (PR #277).
- Improved documentation (PR #283, PR #288).
- Correctly use quadrants in abel.Transform (PR #287).
- Circularization now uses periodic splines (to avoid discontinuity), with smoothing determined by the RMS tolerance instead of the nonintuitive "smooth" parameter (PR #293).
- Corrected and improved tools.center (PR #302).
- Moved numpy import to try block in setup.py. This allows pip to install PyAbel in situations where numpy is not already installed (PR #310).

v0.8.3 (2019-08-16)

- New tools.vmi.Distributions class for extracting radial intensity and anisotropy distributions (PR #257).
- Dropped PyAbel version from basex cache files (PR #260).

v0.8.2 (2019-07-11)

- Added forward transform to basex method (PR #240).
- Corrected tools.transform_pairs.profile4 (PR #241).
- Removed tools.transform_pairs.profile8, which was the same as profile6 (PR #241).
- Major changes in benchmark.AbelTiming class (PR #244, #252).
- Corrected image shift in onion_bordas method (PR #248).
- Changed gradient calculations in hansenlaw method to first-order finite difference (PR #249).
- Corrected background width for Dribinski sample image in tools.analytical.SampleImage (PR #255).

v0.8.1 (2018-11-29)

- Implemented Tikhonov regularization in basex method (PR #227).
- Improvements and changes in basis caching for methods using basis sets (PR #227, #232, #235).
- Improvements in basis generation for basex method, allow any sigma (PR #235).
- Added intensity correction to basex method (PR #235).
- New tools.polynomial module for analytical Abel transform of piecewise polynomials (PR #235).

(2016-04-20)

- Lin-BASEX method of Thomas Gerber and co-workers available (PR #177)

(2016-03-20)

- Dasch two-point, three-point (updated), and onion-peeling available (PR #155).

(2016-03-15)

- Changed abel.transform to be a class rather than a function. The previous syntax of abel.transform(IM)[‘transform’] has been replaced with abel.Transform(IM).transform.

Bibliography

[bordas1996] C. Bordas, F. Paulig, H. Helm, and D. L. Huestis. Photoelectron imaging spectrometry: Principle and inversion method. Rev. Sci. Instrum., **67**, 2257, 1996. DOI: 10.1063/1.1147044.

[chandler1987] David W. Chandler and Paul L. Houston. Two-dimensional imaging of state-selected photodissociation products detected by multiphoton ionization. J. Chem. Phys., **87**, 1445, 1987. DOI: 10.1063/1.453276.

[cignoli2001] Francesco Cignoli, Silvana De Iuliis, Vittorio Manta, and Giorgio Zizak. Two-dimensional two-wavelength emission technique for soot diagnostics. Appl. Opt., **40**, 5370, 2001. DOI: 10.1364/AO.40.005370.

[coppersmith1990] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. J. Symb. Comput., **9**, 251, 1990. DOI: 10.1016/S0747-7171(08)80013-2.

[dasch1992] Cameron J. Dasch. One-dimensional tomography: a comparison of Abel, onion-peeling, and filtered backprojection methods. Appl. Opt., **31**, 1146, 1992. DOI: 10.1364/AO.31.001146.

[daun2006] Kyle J. Daun, Kevin A. Thomson, Fengshan Liu, Fengshan J. Smallwood, Deconvolution of axisymmetric flame properties using Tikhonov regularization. Appl. Opt., **45**, 4638, 2006. DOI: 10.1364/AO.45.004638.

[demicheli2017] Enrico De Micheli. A fast algorithm for the inversion of Abel's transform. Appl. Math. Comput., **301**, 12, 2017. DOI: 10.1016/j.amc.2016.12.009.

[dick2014] Bernhard Dick. Inverting ion images without Abel inversion: maximum entropy reconstruction of velocity maps. Phys. Chem. Chem. Phys., **16**, 570, 2014. DOI: 10.1039/C3CP53673D.

[deiluliis1998] S. De Iuliis, M. Barbini, S. Benecchi, F. Cignoli, and G. Zizak. Determination of the soot volume fraction in an ethylene diffusion flame by multiwavelength analysis of soot radiation. Combust. Flame, **115**, 253, 1998. DOI: 10.1016/S0010-2180(97)00357-X.

[dribinski2002] Vladimir Dribinski, Alexei Ossadtchi, Vladimir A. Mandelshtam, and Hanna Reisler. Reconstruction of Abel-transformable images: The Gaussian basis-set expansion Abel transform method. Rev. Sci. Instrum., *73*, 2634, 2002. DOI: 10.1063/1.1482156.

[das2017] Dhrubajyoti D. Das, William J. Cannella, Charles S. McEnally, Charles J. Mueller, and Lisa D. Pfefferle. Two-dimensional soot volume fraction measurements in flames doped with large hydrocarbons. Proc. Combust. Inst., **36**, 871, 2017. DOI: 10.1016/j.proci.2016.06.047.

[garcia2004] Gustavo A. Garcia, Laurent Nahon, and Ivan Powis. Two-dimensional charged particle image inversion using a polar basis function expansion. Rev. Sci. Instrum., **75**, 4989, 2004. DOI: 10.1063/1.1807578.

[gascooke2000] Jason R. Gascooke. Energy Transfer in Polyatomic-Rare Gas Collisions and Van Der Waals Molecule Dissociation. PhD thesis, Flinders University, SA 5001, Australia, 2000. Available at github.com/PyAbel/abel_info/blob/master/Gascooke_Thesis.pdf.

[gascooke2017] Jason R. Gascooke, Stephen T. Gibson, and Warren D. Lawrance. A "circularisation" method to repair deformations and determine the centre of velocity map images. J. Chem. Phys., **147**, 013924, 2017. DOI: 10.1063/1.4981024.

[gerber2013] Thomas Gerber, 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**, 033101, 2013. DOI: 10.1063/1.4793404.

[gladstone2016] Par G. Randall Gladstone, S. Alan Stern, Kimberly Ennico, Catherine B. Olkin, Harold A. Weaver, Leslie A. Young, Michael E. Summers, Darrell F. Strobel, David P. Hinson, Joshua A. Kammer, Alex H. Parker, Andrew J. Steffl, Ivan R. Linscott, Joel Wm. Parker, Andrew F. Cheng, David C. Slater, Maarten H. Versteeg, Thomas K. Greathouse, Kurt D. Retherford, Henry Throop, Nathaniel J. Cunningham, William W. Woods, Kelsi N. Singer, Constantine C. C. Tsang, Eric Schindhelm, Carey M. Lisse, Michael L. Wong, Yuk L. Yung, Xun Zhu, Werner Curdt, Panayotis Lavvas, Eliot F. Young, G. Leonard Tyler, and The New Horizons Science Team. The atmosphere of Pluto as observed by New Horizons. Science, **351**, 6279, 2016. DOI: 10.1126/science.aad8866.

[glasser1978] J. Glasser, J. Chapelle, and J. C. Boettner. Abel inversion applied to plasma spectroscopy: a new interactive method. Appl. Opt., **17**, 3750, 1978. DOI: 10.1364/AO.17.003750.

[hansen1985] Eric W. Hansen and Phaih-Lan Law. Recursive methods for computing the abel transform and its inverse. J. Opt. Soc. Am. A, **2**, 510, Apr 1985. DOI: 10.1364/JOSAA.2.000510.

[hansen1985b] E. Hansen. Fast hankel transform algorithm. IEEE Trans. Acoust., **33**, 666–671, 1985. DOI: 10.1109/tassp.1985.1164579.

[harrison2018] G. R. Harrison, J. C. Vaughan, B. Hidle, and G. M. Laurent. DAVIS: a direct algorithm for velocity-map imaging system. J of Chem. Phys., **148**, 194101, 2018. DOI: 10.1063/1.5025057.

[hickstein2019] Daniel D. Hickstein, Stephen T. Gibson, Roman Yurchak, Dhrubajyoti D. Das, Mikhail Ryazanov. A direct comparison of high-speed methods for the numerical Abel transform. Rev. Sci. Instrum., **90**, 065115, 2019. DOI: 10.1063/1.5092635.

[lumpe2007] J. D. Lumpe, L. E. Floyd, L. C. Herring, S. T. Gibson, and B. R. Lewis. Measurements of thermospheric molecular oxygen from the solar ultraviolet spectral irradiance monitor. J. Geophys. Res. Atmos., **112**, D16308, 2007. DOI: 10.1029/2006JD008076.

[rallis2014] C. E. Rallis, T. G. Burwitz, P. R. Andrews, M. Zohrabi, R. Averin, S. De, B. Bergues, Bethany Jochim,A. V. Voznyuk, Neal Gregerson, B. Gaire, I. Znakovskaya, J. McKenna, K. D. Carnes, M. F. Kling, I. Ben-Itzhak, and E. Wells. Incorporating real time velocity map image reconstruction into closed-loop coherent control. Rev. Sci. Instrum., **85**, 113105, 2014. DOI: 10.1063/1.4899267.

[ryazanov2012] Mikhail Ryazanov. Development and implementation of methods for sliced velocity map imaging. Studies of overtone-induced dissociation and isomerization dynamics of hydroxymethyl radical (CH_2OH and CD_2OH). PhD thesis, University of Southern California, 2012. www.proquest.com/docview/1289069738

[snelling1999] David R. Snelling, Kevin A. Thomson, Gregory J. Smallwood, and Ömer L. Gülder. Two-dimensional imaging of soot volume fraction in laminar diffusion flames. Appl. Opt., **38**, 2478, 1999. DOI: 10.1364/AO.38.002478.

[vanduzor2010] Matthew Van Duzor, Foster Mbaiwa, Jie Wei, Tulsi Singh, Richard Mabbs, Andrei Sanov, Steven J. Cavanagh, Stephen T. Gibson, Brenton R. Lewis, and Jason R. Gascooke. Vibronic coupling in the superoxide anion: The vibrational dependence of the photoelectron angular distribution. J. Chem. Phys., **133**, 174311, 2010. DOI: 10.1063/1.3493349.

[whitaker2003] B. J. Whitaker. Imaging in Molecular Dynamics: Technology and Applications. Cambridge University Press, 2003. ISBN 9781139437905. books.google.com/books?id=m8AYdeM3aRYC.

[yurchak2015] Roman Yurchak. Experimental and numerical study of accretion-ejection mechanisms in laboratory astrophysics. Thesis, Ecole Polytechnique (EDX), 2015. theses.hal.science/tel-01338614.

Python Module Index

Index