openAbel Documentation

Release 0.2

Oliver Sebastian Haas

Aug 12, 2020

Contents

Start by having a look at the *README*, at the *examples* or at a little bit more details on the *transform methods*.

Contents:

CHAPTER 1

openAbel README

Note: It's best to view this readme in the **openAbel** documentation.

1.1 Introduction

The main goal of **openAbel** is to provide fast and efficients Abel transforms of equispaced data in Python with all actual calculations done in Cython. The most useful methods implemented in this module for that purpose use the Fast Multipole Method combined with arbitrary order end correction of the trapezoidal rule to achieve small errors and fast convergence, as well as linear computational complexity. A couple of other methods are implemented for comparisons. Abel transform function can be used from Python and with numpy arrays or from Cython using pointers.

1.2 Quick Start

In most cases this should be pretty simple:

- Clone the repository: `git clone https://github.com/oliverhaas/openAbel.git`
- Install: `sudo python setup.py install`
- Run example: `python example000_simpleForward.py`

This assumes dependencies are already met and you run a more or less similar system to mine (see *Dependencies*).

1.3 Dependencies

The code was run on several Ubuntu systems without problems. More specific I'm running Ubuntu 16.04 and the following libraries and Python modules, which were all installed the standard way with either `sudo apt install`

`libName` or `sudo pip install moduleName`.

- Python 3.5.2
- Numpy 1.18.1
- Scipy 1.4.1
- Cython 0.29.14
- Matplotlib 3.0.3

As usual newer versions of the libraries should work as well, and many older versions will too. I'm sure it's possible to get **openAbel** to run on vastly different systems, like e.g. Windows systems, but obviously I haven't extensively tested majorly different setups.

1.4 Issues

If there are any issues, bugs or feature request just let me know. As of now there are some gaps in the implementation, e.g. not all transform types are available in all methods, but since the default method vastly outperforms every other method anyway it's not really a pressing issue.

1.5 Transform Methods

For the default and most important method of **openAbel** we adapted the Chebyshev interpolation Fast Multipole Method (FMM) as described by Tausch and calculated end corrections specifically for the Abel transform similar to Kapur. If data points outside of the integration interval can be provided these end corrections are arbitrary order stable and we provide coefficients up to 20th order, otherwise it's recommended to use at most 5th order. The FMM leads to a linear *O(N)* computational complexity algorithm.

In both error and computational complexity there is no better existing method for the intended purpose to my knowledge. I should really stress that there are dozens of publications and methods out there which claim to be fast and/or accurate, but don't get anywhere close to **openAbel** in those aspects.

For more information see the **openAbel** documentation on the transform methods and examples.

1.6 Copyright and License

The code **openAbel** is published under the GNU GPL version 3. This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

For more information see the GNU General Public License copy provided in this repository LICENSE.

CHAPTER 2

Transform Types

In **openAbel** due to the equispaced discretization all methods truncate the Abel transform integral, e.g. for the forward Abel transform

$$F(y) = 2\int_y^\infty \frac{f(r)r}{\sqrt{r^2-y^2}}dr \approx 2\int_y^R \frac{f(r)r}{\sqrt{r^2-y^2}}dr \ .$$

This is sometimes called finite Abel transform. Since $f(r)$ often has compact support or decays very quickly (and R can be chosen very large with a fast transform method) this introduces an arbitrarily small error.

It should be noted that often one can use variable transformations or other discretizations to simplify the calculation of the above integrals. However, often one is interested in exactly the in **openAbel** implemented case on equispaced discretization. This is often due to the relation of the Abel transform with the Fourier and Hankel transforms and the desire to use the same discretization as the FFT or a discrete convolution, or just by the given data (e.g. from experiments).

The type of transform can be chosen by setting the `forwardBackward` parameter:

```
import openAbel
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize)
```

The parameter `stepSize` is the grid spacing of the equidistant grid, `nData` the length of the data input array, and `shift` is an offset of the samples to the symmetry axis and can usually be only 0 or 0.5 (input in units of `stepSize`).

2.1 Forward Abel Transform

The Forward Abel transform is defined as

$$F(y) = 2\int_y^\infty \frac{f(r)r}{\sqrt{r^2-y^2}}dr \approx 2\int_y^R \frac{f(r)r}{\sqrt{r^2-y^2}}dr \ .$$

The Forward Abel Transform is chosen by setting `forwardBackward=-1`.

2.2 Backward (or Inverse) Abel Transform

The Backward (or Inverse) Abel transform is defined as

$$f(r) = -\frac{1}{\pi}\int_r^\infty \frac{F'(y)}{\sqrt{y^2-r^2}}dy \approx -\frac{1}{\pi}\int_r^R \frac{F'(y)}{\sqrt{y^2-r^2}}dy \ .$$

openAbel takes care of taking the derivate of the input data supplied by the user. The Backward Abel Transform is chosen by setting `forwardBackward=1`.

2.3 Backward (or Inverse) Abel Transform with Derivative Input

The Backward (or Inverse) Abel Transform with derivative input is defined as

$$f(r) = -\frac{1}{\pi}\int_r^\infty \frac{g(y)}{\sqrt{y^2-r^2}}dy \approx -\frac{1}{\pi}\int_r^R \frac{g(y)}{\sqrt{y^2-r^2}}dy \ .$$

In contrast to the normal Backward Abel Transform, **openAbel** expects to get the derivative as input by the user.

The Backward Abel Transform with derivative input is chosen by setting `forwardBackward=2`.

2.4 Modified Forward Abel Transform

What we call the Modified Forward Abel Transform in **openAbel** is defined as the integral

$$H(y) = 2\int_y^\infty \frac{h(r)y^2}{r^2\sqrt{r^2-y^2}}dr \approx 2\int_y^R \frac{h(r)y^2}{r^2\sqrt{r^2-y^2}}dr \ .$$

I encountered this integral when a radial electric field of an atom (which has a $1/r^2$ singularity we want to integrate properly) is projected instead of a simpler function like with the normal Forward Abel Transform. One could just use the parameter `shift = 0.5` instead to avoid the singularity of the electric field, but if one incorporates the singularity in the actual integral the convergence is much better. I recommend writing similar methods if one encounters other types of singularities in the Abel Transform.

The Modified Forward Abel Transform is chosen by setting `forwardBackward=-2`.

CHAPTER 3

Transform Methods

In **openAbel** there are a couple of different algorithms for the calculation of the Abel transforms implemented, although most of them are just for comparisons and is it recommended to only use the default method.

The main two obstacles when calculating the transforms numerically are the singularity at $r = y$ and the dependence of the result on y, meaning computational complexity is quadratic $O(N^2)$ if one naively integrates. The main difference between the implemented transforms is how those two issues are treated.

When creating the Abel transform object the `method` keyword argument can be provided to chose different transform methods:

```
import openAbel
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 3, order =␣
↪2)
```

The methods with end corrections can do the transformation in different orders of accuracy by setting the `order` keyword argument; all other methods ignore `order`. Note when we talk about n order accuracy we usually mean $(n+1/2)$ order accuracy due to the square root in the Abel transform kernel. For higher order methods the transformed function has to be sufficiently smooth to achieve the full order of convergence, and in very extreme cases the transform become unstable if high order is used on non-smooth functions. The length of the data vector `nData` we denote as N in the math formulas.

Overall cases where a user should use anything other than `method = 3` (default) and `order = 2` (default) to `order = 5` will be very rare. For a detailed comparison of the methods it is recommended to look at example004_fullComparison.

3.1 Desingularized Trapezoidal Rule

```
# order keyword argument is ignored (only first order implemented)
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 0)
```

The desingularized trapezoidal rule is probably the simplest practicable algorithm. It subtracts the singularity and integrates it analytically, and numerically integrates the remaining desingularized term with the trapezoidal rule. In

the implementation this is done to first order, i.e. for the forward Abel transform this leads to

$$F(y) = 2\int_y^R \frac{(f(r) - f(y))r}{\sqrt{r^2 - y^2}} dr + f(y)\sqrt{R^2 - y^2}\ .$$

Now the singularity seems to be removed, but a closer look and one can see that the singularity is still there in the derivative of the integrand, so the convergence is first order in N instead of second order expected when using trapezoidal rule. One can analytically remove the singularity in higher order with more terms, but this gets kinda complicated (and possibly unstable, plus there are other practical issues). The trapezoidal rule portion of the method leads to quadratic $O(N^2)$ computational complexity of the method.

3.2 Hansen-Law Method

```
# order keyword argument is ignored (only somewhat first order implemented)
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 1)
```

The Hansen-Law method by Hansen and Law is a space state model approximation of the Abel transform kernel. With that method recursively transforms a piecewise linear approximation of the input functions to integrate analytically piece by piece. In principle this results in an 2nd order accurate transform, but the approximation of the Abel transform kernel as a sum of exponentials is quite difficult. In other words the approximation

$$\frac{1}{\sqrt{1 - \exp(-2t)}} \approx \sum_{k=1}^{K} \exp(-\lambda_k t)$$

is in practice not possible to achieve with high accuracy and reasonable K. This is the main limitation of the method, and the original space state model approxmation has a typical relative error of 10^{-3} at best – then it just stops converging with increasing N. If one ignores several details that makes the method apparently linear $O(N)$ computational complexity, so it is implemented here for comparisons. More comments in the remarks.

3.3 Trapezoidal Rule with End Corrections

```
# 0 < order < 20
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 2, order =␣
↪2)
```

The trapezoidal rule with end correction improves on the desingularized trapezoidal rule. It doesn't require analytical integration because it uses precalculated end correction coefficients of arbitrary order. As described in Kapur one can contruct α_i and β_i such that the approxmation

$$\int_a^b f(x)dx \approx h \cdot \sum_{i=1}^{N-2} f(x_i) + h \cdot \sum_{i=0}^{M-1} \alpha_i f(x_{i-p}) + h \cdot \sum_{i=0}^{M-1} \beta_i f(x_{N-1-q})$$

is accurate to order M. Note that p and q should be chosen such that the correction is centered around the end points: Similar to central finite differences this leads to an arbitrary order stable scheme, and thus incredibly fast convergence and small errors. Otherwise it's not recommended to go higher than $M = 5$, again similar to forward and backward finite differences. The trapezoidal rule portion of the method leads to quadratic $O(N^2)$ computational complexity of the method.

Since the calculation of the end correction coefficients requires some analytical calculations, is quite troublesome and time consuming, they have been precalculated in *Mathematica* and stored in binary **.npy* , so they are only loaded by the **openAbel** code when needed and don't have to be calculated. The *Mathematica* notebook which was used to calculate these end correction coefficients can be found in this repository as well.

3.4 Fast Multipole Method with End Corrections

```
# 0 < order < 20
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 3, order =
↪2)
```

The default and recommended method is the Fast Multipole Method (FMM) with end corrections. This method provides a fast linear $O(N)$ computational complexity transform of arbitrary order. The specific FMM used is based on Chebyshev interpolation and nicely described and applied by Tausch on a similar problem. In principle the FMM uses a hierarchic decomposition to combine a linear amount of direct short-range contributions and smooth approximations of long-range contributions with efficient reuse of intermediate results to get in total a linear $O(N)$ computational complexity algorithm. This method thus provides extremely fast convergence and fast computation, and is optimal in that sense for the intended purpose.

3.5 Remarks on Transforms of Noisy Data

For specifically the inverse Abel transform of noisy data (e.g. experimental data) there are a lot of algorithms described in literature which might perform better in some aspects, since they either incorporate some assumptions about the data or some kind of smoothing/filtering of the noise. A nice starting point for people interested in those methods is the Python module PyAbel.

However, there is no reason not to combine the methods provided in **openAbel** with some kind of filering for nicer results. I've had good results with maximally flat filters, as seen in example003_noisyBackward, and with additional material in the Mathematica notebook.

Overall even in this special case there are no algorithms to my knowledge which perform inherently better than the default algorithms of **openAbel** by default.

CHAPTER 4

Examples

4.1 example000_simpleForward

This example is just a simple forward transform of a Gaussian. Aside from showing how to do a simple forward transform, this example shows how for non truncated domain an error is introduced.

```
1  ##############################################################################################################
2  # Simple example which calculates forward Abel transform of a Gaussian.
3  # Results are compared with the analytical solution. Mostly default parameters are used.
4  ##############################################################################################################
5
6
7  import openAbel
8  import numpy as np
9  from scipy.special import erf
10 import matplotlib.pyplot as mpl
11
12 ##############################################################################################################
13 # Plotting setup
14 # This block can be ignored, it's just for nicer plots.
15
16 params = {
17    'axes.labelsize': 8,
18    'font.size': 8,
19    'legend.fontsize': 10,
20    'xtick.labelsize': 10,
21    'ytick.labelsize': 10,
22    'text.usetex': False,
23    'figure.figsize': [5., 5.]
24    }
```

(continues on next page)

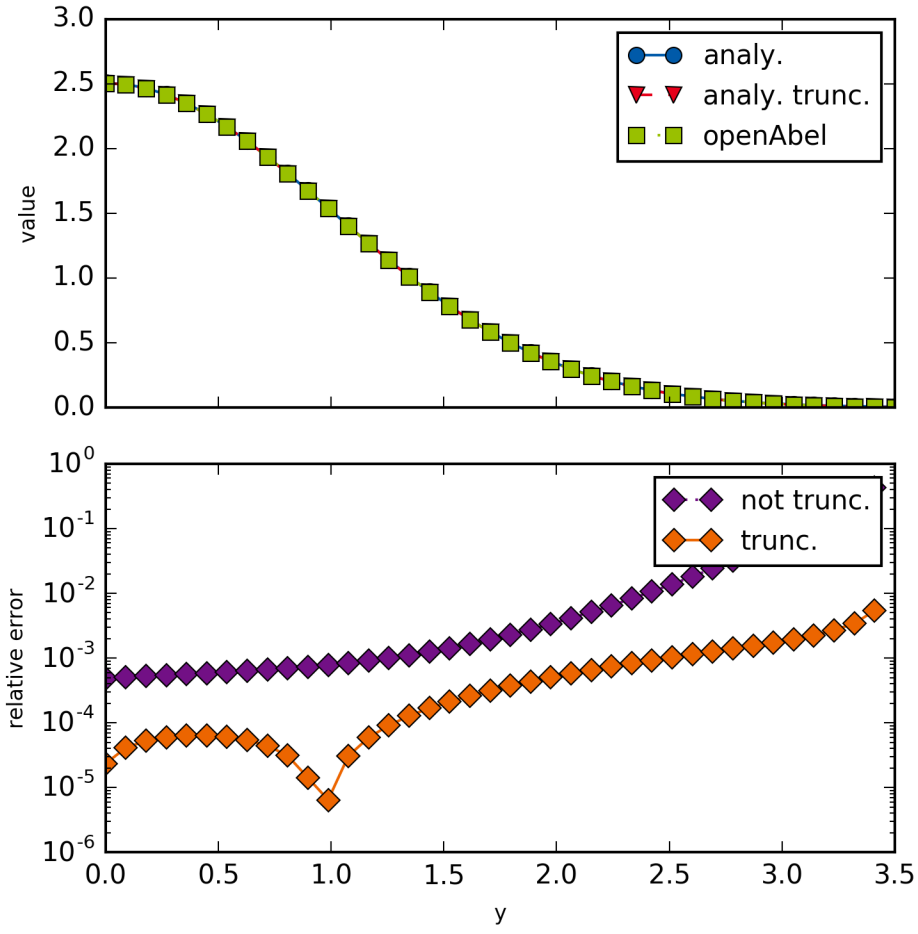

Fig. 1: Simple forward transform of a Gaussian.

(continued from previous page)

```
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

################################################################################################################################

# Parameters
nData = 40
shift = 0.
xMax = 3.5
sig = 1.
stepSize = xMax/(nData-1)
forwardBackward = -1    # Forward transform, similar definition ('-1' = forward) as in FFT libraries.

# Create Abel transform object, which does all precomputation possible without knowing the exact data.
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize)

# Input data
xx = np.linspace(shift*stepSize, xMax, nData)
dataIn = np.exp(-0.5*xx**2/sig**2)

# Forward Abel transform and analytical result.
# We show both the analytical result of a truncated Gaussian and a standard Gaussian to show
# that some error is due to truncation.
dataOut = abelObj.execute(dataIn)
dataOutAna = dataIn*np.sqrt(2*np.pi)*sig
dataOutAnaTrunc = dataIn*np.sqrt(2*np.pi)*sig*erf(np.sqrt((xMax**2-xx**2)/2)/sig)


# Plotting
fig, axarr = mpl.subplots(2, 1, sharex=True)

axarr[0].plot(xx, dataOutAna, color = colors[0], marker = markers[0], linestyle = linestyles[0], label='analy.')
axarr[0].plot(xx, dataOutAnaTrunc, color = colors[1], marker = markers[1], linestyle = linestyles[1], label='analy. trunc.')
axarr[0].plot(xx, dataOut, color = colors[2], marker = markers[2], linestyle = linestyles[2], label='openAbel')
axarr[0].set_ylabel('value')
axarr[0].legend()

axarr[1].semilogy(xx[:-1], np.abs((dataOut[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
                  color = colors[3], marker = markers[3], linestyle = linestyles[3], label = 'not trunc.')
axarr[1].semilogy(xx[:-1], np.abs((dataOut[:-1]-dataOutAnaTrunc[:-1])/dataOutAnaTrunc[:-1]),
                  color = colors[4], marker = markers[3], linestyle = linestyles[4], label='trunc.')
```

(continues on next page)

(continued from previous page)

```
axarr[1].set_ylabel('relative error')
axarr[1].set_xlabel('y')
axarr[1].legend()

mpl.tight_layout()
mpl.savefig('example000_simpleForward.png', dpi=300)

mpl.show()
```

4.2 example001_simpleBackward

This example is just a simple backward transform of a Gaussian. Aside from showing how to do a simple backward transform, this example shows how for non truncated domain an error is introduced, and how taking the derivative analytically of the data (if possible) improves the error. Taking numerical derivatives always amplifies noise and increases the resulting error.

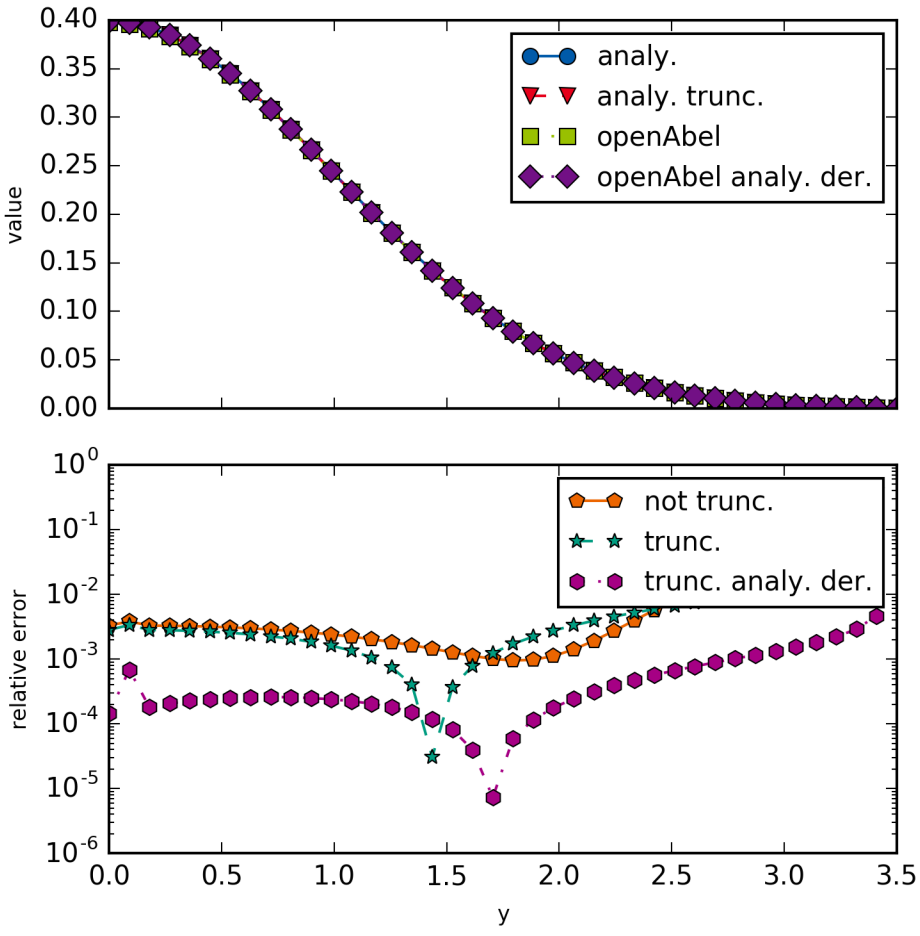

Fig. 2: Simple backward transform of a Gaussian.

```
################################################################################
################################################################
# Simple example which calculates backward Abel transform of a Gaussian.
# Results are compared with the analytical solution. Mostly default parameters are␣
↪used.
################################################################################
################################################################


import openAbel
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as mpl

################################################################################
################################################################
# Plotting setup
# This block can be ignored, it's just for nicer plots.

params = {
   'axes.labelsize': 8,
   'font.size': 8,
   'legend.fontsize': 10,
   'xtick.labelsize': 10,
   'ytick.labelsize': 10,
   'text.usetex': False,
   'figure.figsize': [5., 5.]
   }
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','
↪#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

################################################################################
################################################################

# Parameters
nData = 40
shift = 0.
xMax = 3.5
sig = 1.
stepSize = xMax/(nData-1)
forwardBackward = 1    # Backward transform, similar definition ('1' = backward) as␣
↪in FFT libraries.

# Create Abel transform object, which does all precomputation possible without␣
↪knowing the exact data.
abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize)

# Input data
xx = np.linspace(shift*stepSize, xMax, nData)
dataIn = np.exp(-0.5*xx**2/sig**2)
```

(continues on next page)

(continued from previous page)

```
50
51  # Backward transform and analytical result.
52  # We show both the analytical result of a truncated Gaussian and a standard Gaussian␣
    ↪to show
53  # that some error is due to truncation.
54  dataOut = abelObj.execute(dataIn)
55  dataOutAna = dataIn/np.sqrt(2*np.pi)/sig
56  dataOutAnaTrunc = dataIn/np.sqrt(2*np.pi)/sig*erf(np.sqrt((xMax**2-xx**2)/2)/sig)
57
58  # There is the option for the user to provide the derivative in the backward Abel␣
    ↪transform directly.
59  # This is useful and can decrease the error, e.g. if the derivative can be taken␣
    ↪analytically.
60  forwardBackward = 2
61  abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize)
62  dataIn = -xx/sig**2*np.exp(-0.5*xx**2/sig**2)
63  dataOut2 = abelObj.execute(dataIn)
64
65
66  # Plotting
67  fig, axarr = mpl.subplots(2, 1, sharex=True)
68
69  axarr[0].plot(xx, dataOutAna, color = colors[0], marker = markers[0], linestyle =␣
    ↪linestyles[0], label='analy.')
70  axarr[0].plot(xx, dataOutAnaTrunc, color = colors[1], marker = markers[1], linestyle␣
    ↪= linestyles[1], label='analy. trunc.')
71  axarr[0].plot(xx, dataOut, color = colors[2], marker = markers[2], linestyle =␣
    ↪linestyles[2], label='openAbel')
72  axarr[0].plot(xx, dataOut2, color = colors[3], marker = markers[3], linestyle =␣
    ↪linestyles[3], label='openAbel analy. der.')
73  axarr[0].set_ylabel('value')
74  axarr[0].legend()
75
76  axarr[1].semilogy(xx[:-1], np.abs((dataOut[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
77                    color = colors[4], marker = markers[4], linestyle = linestyles[4],␣
    ↪label = 'not trunc.')
78  axarr[1].semilogy(xx[:-1], np.abs((dataOut[:-1]-dataOutAnaTrunc[:-1])/
    ↪dataOutAnaTrunc[:-1]),
79                    color = colors[5], marker = markers[5], linestyle = linestyles[5], ␣
    ↪label='trunc.')
80  axarr[1].semilogy(xx[:-1], np.abs((dataOut2[:-1]-dataOutAnaTrunc[:-1])/
    ↪dataOutAnaTrunc[:-1]),
81                    color = colors[6], marker = markers[6], linestyle = linestyles[6], ␣
    ↪label='trunc. analy. der.')
82  axarr[1].set_ylabel('relative error')
83  axarr[1].set_xlabel('y')
84  axarr[1].legend()
85
86  mpl.tight_layout()
87  mpl.savefig('example001_simpleBackward.png', dpi=300)
88
89  mpl.show()
90
91
92
```

4.3 example002_methodOrder

This example shows how to switch to other transform methods and orders. It illustrates how quickly the errors of high order methods converge to machine precision, even for very small data sets. It is of course important that the input data is sufficiently smooth and other errors (e.g. truncation errors) are small enough as well.

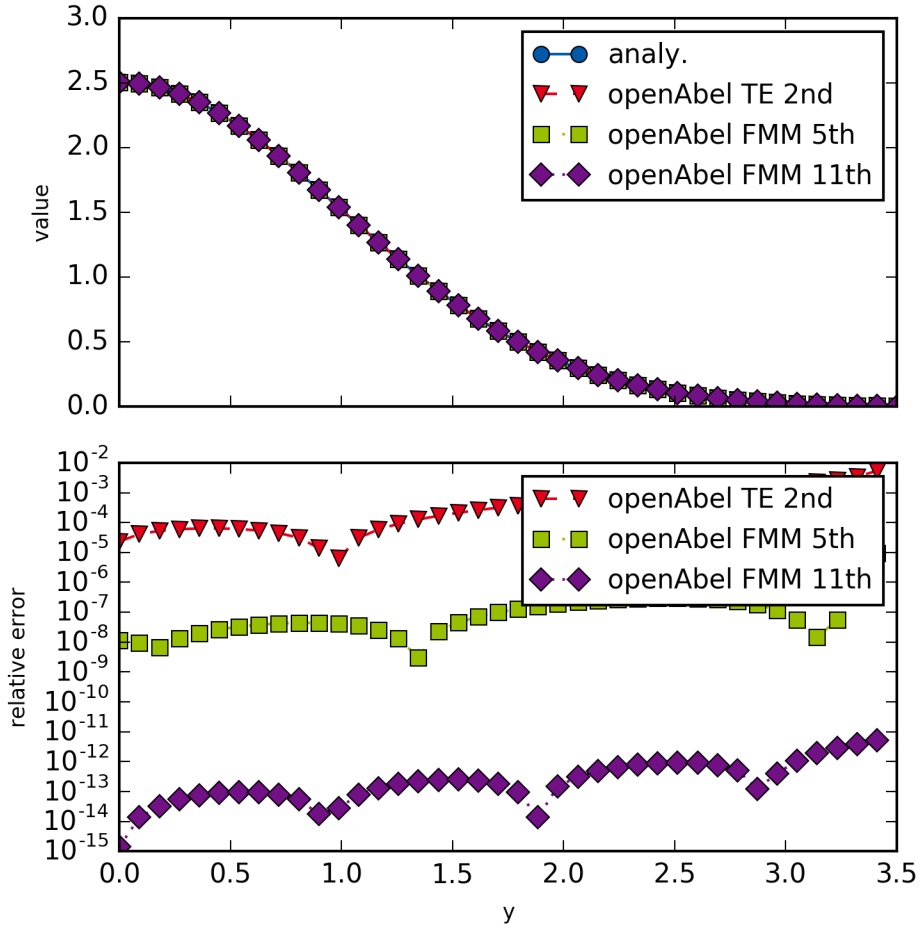

Fig. 3: Different methods and orders.

```
##############################################################################
↪#####################################################
# Simple example which shows how to select different methods and orders.
# Results are compared with the analytical solution.
##############################################################################
↪#####################################################


import openAbel
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as mpl

##############################################################################
↪#####################################################
```

(continues on next page)

(continued from previous page)

```
# Plotting setup
# This block can be ignored, it's just for nicer plots.

params = {
   'axes.labelsize': 8,
   'font.size': 8,
   'legend.fontsize': 10,
   'xtick.labelsize': 10,
   'ytick.labelsize': 10,
   'text.usetex': False,
   'figure.figsize': [5., 5.]
   }
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','
↪#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

##############################################################################
↪############################################################

# Parameters
nData = 40
shift = 0.
xMax = 3.5
sig = 1.
stepSize = xMax/(nData-1)

forwardBackward = -1    # Forward transform, similar definition ('1' = backward) as␣
↪in FFT libraries.

# Create Abel transform object for three different methods and orders.
# Some methods ignore the order keyword argument, and for the normal user
# only method = 3 and order = 2 to order = 5 are recommended.
# Higher orders require data outside the integration domain to be stable.
# For more information see the documentation.
abelObj0 = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 2, order =␣
↪2)
abelObj1 = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 3, order =␣
↪5)
abelObj2 = openAbel.Abel(nData, forwardBackward, shift, stepSize, method = 3, order =␣
↪11)

# Input data
xx = np.linspace(shift*stepSize, xMax, nData)
dataIn = np.exp(-0.5*xx**2/sig**2)
xxExt = np.linspace(shift*stepSize, xMax+5*stepSize, nData+5)    # floor((order-1)/2)␣
↪extra points at right end
dataInExt = np.exp(-0.5*xxExt**2/sig**2)

# Backward transform and analytical result

dataOut0 = abelObj0.execute(dataIn)
```

(continues on next page)

(continued from previous page)

```
dataOut1 = abelObj1.execute(dataIn)
dataOut2 = abelObj2.execute(dataInExt, leftBoundary = 2, rightBoundary = 3)  # 2 means use even symmetry, 3 means input extra points.
dataOutAna = dataIn*np.sqrt(2*np.pi)*sig*erf(np.sqrt((xMax**2-xx**2)/2)/sig)

# Plotting
fig, axarr = mpl.subplots(2, 1, sharex=True)

axarr[0].plot(xx, dataOutAna, color = colors[0], marker = markers[0], linestyle = linestyles[0], label='analy.')
axarr[0].plot(xx, dataOut0, color = colors[1], marker = markers[1], linestyle = linestyles[1], label='openAbel TE 2nd')
axarr[0].plot(xx, dataOut1, color = colors[2], marker = markers[2], linestyle = linestyles[2], label='openAbel FMM 5th')
axarr[0].plot(xx, dataOut2, color = colors[3], marker = markers[3], linestyle = linestyles[3], label='openAbel FMM 11th')
axarr[0].set_ylabel('value')
axarr[0].legend()

axarr[1].semilogy(xx[:-1]/sig, np.abs((dataOut0[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
                  color = colors[1], marker = markers[1], linestyle = linestyles[1], label = 'openAbel TE 2nd')
axarr[1].semilogy(xx[:-1]/sig, np.abs((dataOut1[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
                  color = colors[2], marker = markers[2], linestyle = linestyles[2], label='openAbel FMM 5th')
axarr[1].semilogy(xx[:-1]/sig, np.abs((dataOut2[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
                  color = colors[3], marker = markers[3], linestyle = linestyles[3], label='openAbel FMM 11th')
axarr[1].set_ylabel('relative error')
axarr[1].set_xlabel('y')
axarr[1].legend()

mpl.tight_layout()
mpl.savefig('example002_methodOrder.png', dpi=300)

mpl.show()
```

4.4 example003_noisyBackward

This example shows how to filter and transform noisy data. It illustrates how noisy input data can lead to large errors of the the backward transform result, and how filters can be used to – at elast visually – alleviate those errors.

Maximally flat filters have been calculated as described in a paper by Hosseini, and a small Mathematica script of the calculation is provided in the additional materials.

```
################################################################################################################################
# Example which calculates backward Abel transform of noisy data.
```

(continues on next page)

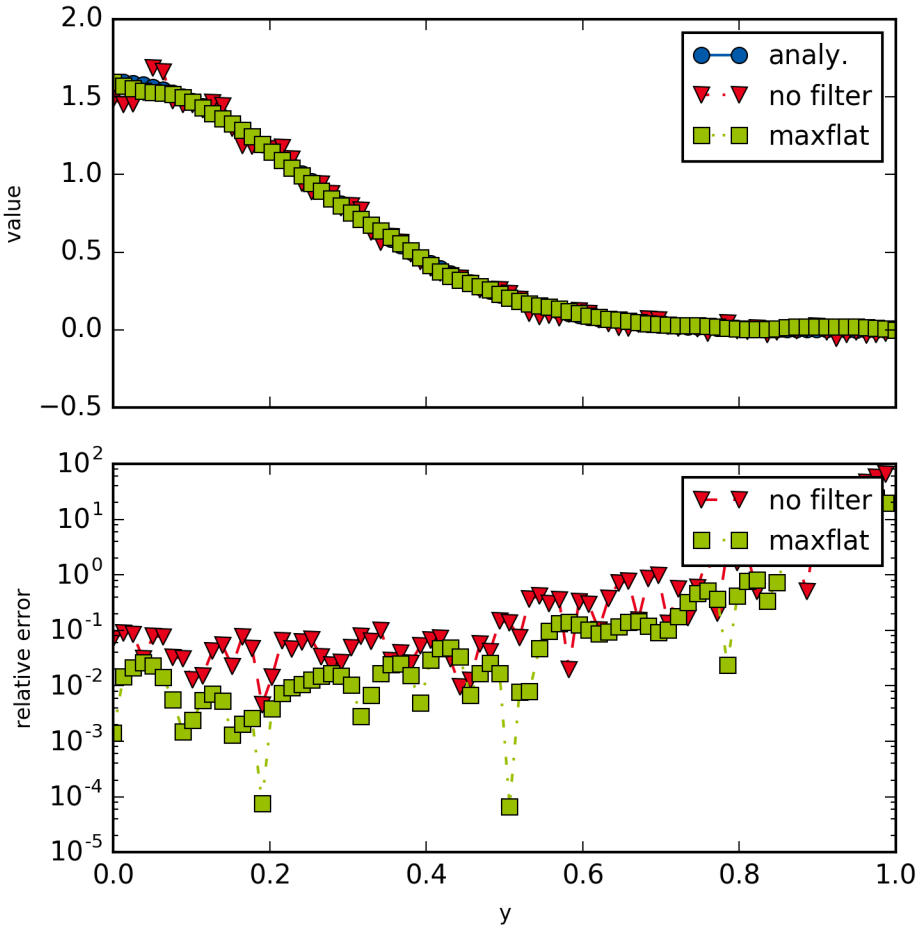

Fig. 4: Backward transform of noisy data.

(continued from previous page)

```
# This is a typical use case for many experimental line-of-sight measurements.
# openAbel doesn't inherently provide any filtering or smoothing, but one
# can achieve good results with manual noise-robust numerical derivatives.
################################################################################
################################################################


import openAbel
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as mpl

################################################################################
################################################################
# Plotting setup
# This block can be ignored, it's just for nicer plots.

params = {
   'axes.labelsize': 8,
   'font.size': 8,
   'legend.fontsize': 10,
   'xtick.labelsize': 10,
   'ytick.labelsize': 10,
   'text.usetex': False,
   'figure.figsize': [5., 5.]
   }
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','
#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

################################################################################
################################################################

# Parameters
nData = 80
xMax = 1.
shift = 0.
sig = 1./4.
stepSize = xMax/(nData-1)
forwardBackward = 2
noiseAmp = 0.01

abelObj = openAbel.Abel(nData, forwardBackward, shift, stepSize)  # Backward Abel␣
↪transform where user inputs derivative


# No filtering
der = np.asarray([0.5, 0., -0.5])/stepSize
xx = np.linspace(-stepSize*(der.shape[0]-1)/2, xMax+stepSize*(der.shape[0]-1)/2,␣
↪nData+(der.shape[0]-1))
dataIn = np.exp(-0.5*xx**2/sig**2)
```

(continues on next page)

(continued from previous page)

```
np.random.seed(2202)
dataInWithNoise = dataIn + noiseAmp*np.random.randn(nData+(der.shape[0]-1))

# Take derivatives
dataInD = np.convolve(dataInWithNoise, der, mode = 'valid')

# Backward transform
dataOutNoFilter = abelObj.execute(dataInD)


# Maximally flat filtering
# The length of this filter should be adjusted to the noise.
# For more information see documentation and orignal maxflat paper https://ieeexplore.ieee.org/document/7944698/.
der = np.asarray([4.76837e-7, 9.53674e-6, 0.0000901222, 0.000534058, 0.00221968, 0.00684929,
                  0.0161719, 0.0295715, 0.041585, 0.0431252, 0.0280313, 0., -0.0280313,
                  -0.0431252, -0.041585, -0.0295715, -0.0161719, -0.00684929,
                  -0.00221968, -0.000534058, -0.0000901222, -9.53674e-6, -4.76837e-7])/stepSize
xx = np.linspace(-stepSize*(der.shape[0]-1)/2, xMax+stepSize*(der.shape[0]-1)/2, nData+(der.shape[0]-1))
dataIn = np.exp(-0.5*xx**2/sig**2)
np.random.seed(2202)
dataInWithNoise = dataIn + noiseAmp*np.random.randn(nData+(der.shape[0]-1))

# Take derivatives
dataInD = np.convolve(dataInWithNoise, der, mode = 'valid')

# Backward transform
dataOutMaxFlat = abelObj.execute(dataInD)

# Analytical result
xx = np.linspace(stepSize*shift, xMax, nData)
dataIn = np.exp(-0.5*xx**2/sig**2)
dataOutAna = dataIn/np.sqrt(2*np.pi)/sig*erf(np.sqrt((xMax**2-xx**2)/2)/sig)


# Plotting
fig, axarr = mpl.subplots(2, 1, sharex=True)

axarr[0].plot(xx, dataOutAna, color = colors[0], marker = markers[0], linestyle = linestyles[0], label='analy.')
axarr[0].plot(xx, dataOutNoFilter, color = colors[1], marker = markers[1], linestyle = linestyles[2], label='no filter')
axarr[0].plot(xx, dataOutMaxFlat, color = colors[2], marker = markers[2], linestyle = linestyles[3], label='maxflat')
axarr[0].set_ylabel('value')
axarr[0].legend()

axarr[1].semilogy(xx[:-1], np.abs((dataOutNoFilter[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
                  color = colors[1], marker = markers[1], linestyle = linestyles[1], label = 'no filter')
axarr[1].semilogy(xx[:-1], np.abs((dataOutMaxFlat[:-1]-dataOutAna[:-1])/dataOutAna[:-1]),
```

(continues on next page)

(continued from previous page)

```
                    color = colors[2], marker = markers[2], linestyle = linestyles[2], ␣
↪label='maxflat')
axarr[1].set_ylabel('relative error')
axarr[1].set_xlabel('y')
axarr[1].legend()

mpl.tight_layout()
mpl.savefig('example003_noisyBackward.png', dpi=300)

mpl.show()
```

4.5 example004_fullComparison

This example provides a rather extensive comparison of different **openAbel** methods. It shows how well the main methods perform in every regard, especially if the input data is sufficiently smooth. Most importantly one can see the fast convergence of the higher order methods in the bottom left plot, and the linear computational complexity of the main methods in the bottom middle and right plots.

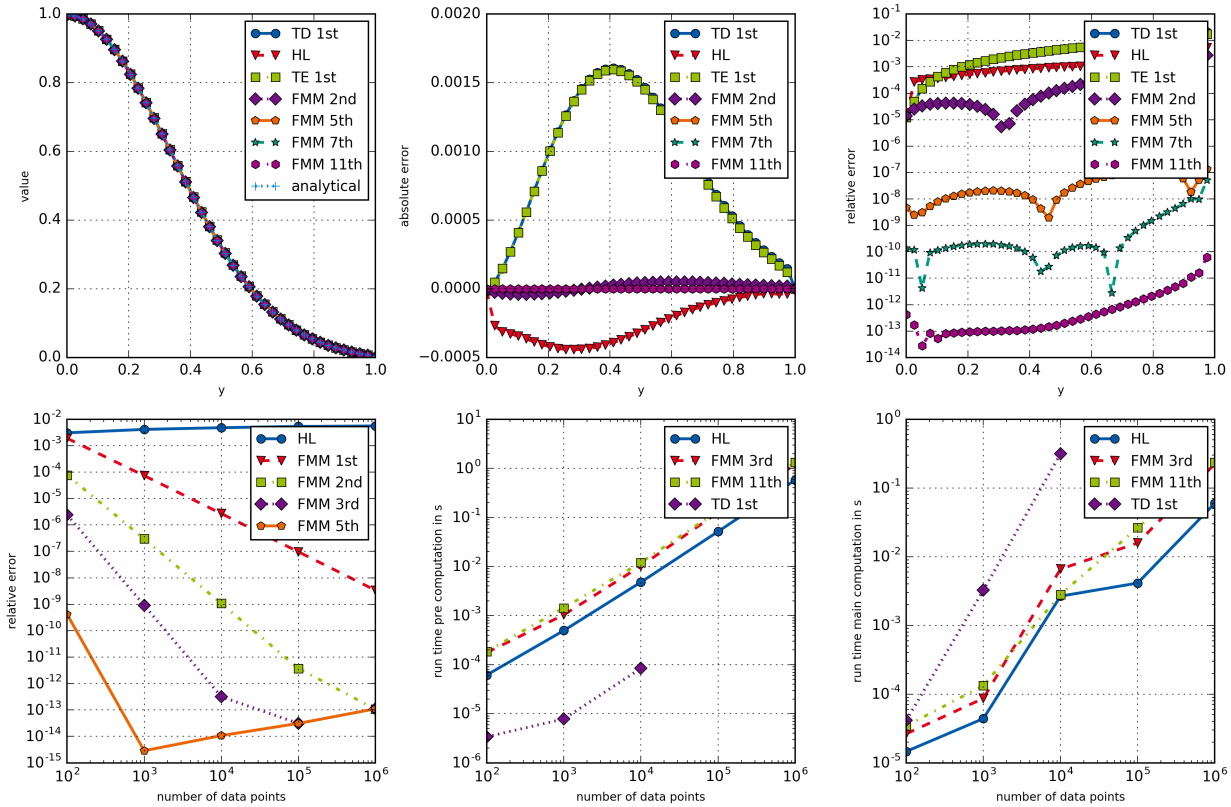

Fig. 5: Comparison of different openAbel methods.

```
################################################################################
↪##################################################
# Example to showcase the different methods and orders, including convergence study␣
↪and run times.
# Forward transform is done here, but similar results apply to the backward transform.
# This example takes possibly minutes to run, so beware
################################################################################
↪##################################################


import openAbel as oa
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as mpl
import datetime as dt
import time as ti

################################################################################
↪##################################################
# Plotting setup

params = {
   'axes.labelsize': 8,
   'font.size': 8,
   'legend.fontsize': 10,
   'xtick.labelsize': 10,
   'ytick.labelsize': 10,
   'text.usetex': False,
   'figure.figsize': [12., 8.]
   }
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','
↪#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = mpl.subplots(2, 3)


################################################################################
↪##################################################
# Error over radius of different methods and orders

def errorAbel(nData, method, order):

    dx = 1./(nData-1);
    xx = np.linspace(0., 1., nData)
    sig = 1./3.

    dataIn = 1./sig/np.sqrt(2*np.pi)*np.exp(-0.5*xx**2/sig**2)

    dataAna = 2.*1./sig/np.sqrt(2*np.pi)*np.exp(-0.5*xx**2/sig**2) * \
              np.sqrt(np.pi/2.)*sig*erf(np.sqrt(1**2-xx**2)/np.sqrt(2.)/sig)
```

(continues on next page)

(continued from previous page)

```
    abelObj = oa.Abel(nData, -1, 0., dx, method = method, order = order)
    dataOut = abelObj.execute(dataIn)

    abserr = dataOut-dataAna
    relerr = np.abs(abserr/np.clip(dataAna, 1.e-300, None))

    return (xx, abserr, relerr, dataOut, dataAna)


# Loop over several methods and orders
names = ['TD 1st', 'HL', 'TE 1st', 'FMM 2nd', 'FMM 5th', 'FMM 7th', 'FMM 11th']
orders = [-1, -1, 1, 2, 5, 7, 11]
methods = [0, 1, 2, 3, 3, 3, 3]

for ii in range(len(orders)):

    (xx, abserr, relerr, dataOut, dataAna) = errorAbel(40, methods[ii], orders[ii])
    ax1.plot(xx, dataOut, label = str(names[ii]), color = colors[ii],
             linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax2.plot(xx, abserr, label = str(names[ii]), color = colors[ii],
                 linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax3.semilogy(xx[:-1], relerr[:-1], label=str(names[ii]), color = colors[ii],
                 linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

ii += 1
ax1.plot(xx, dataAna, label = 'analytical', color = colors[ii],
         linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

ax1.legend()
ax1.set_xlabel('y')
ax1.set_ylabel('value')
ax1.grid(True)

ax2.legend()
ax2.set_xlabel('y')
ax2.set_ylabel('absolute error')
ax2.grid(True)

ax3.legend()
ax3.set_xlabel('y')
ax3.set_ylabel('relative error')
ax3.grid(True)


################################################################################
↪#####################################################
# Convergence of different methods and orders

def convergenceAbel(nArray, method, order):

    conv = np.empty(nArray.shape[0])
    for ii in range(nArray.shape[0]):

        nData = nArray[ii]
        dx = 1./(nData-1);
        xx = np.linspace(0., 1., nData)
```

(continues on next page)

(continued from previous page)

```
        sig = 1./3.

        dataIn = 1./sig/np.sqrt(2*np.pi)*np.exp(-0.5*xx**2/sig**2)

        abelObj = oa.Abel(nData, -1, 0., dx, method = method, order = order)
        dataOut = abelObj.execute(dataIn)

        dataAna = 2./sig/np.sqrt(2*np.pi)*np.exp(-0.5*xx**2/sig**2) * \
                    np.sqrt(np.pi/2.)*sig*erf(np.sqrt(1**2-xx**2)/np.sqrt(2.)/sig)
        conv[ii] = np.sqrt(np.sum(((dataOut-dataAna)/np.clip(dataAna, 1.e-300,␣
↪None))**2)/nData)

    return conv


# Loop over several methods and orders
names = ['HL', 'FMM 1st', 'FMM 2nd', 'FMM 3rd', 'FMM 5th']
orders = [-1, 1, 2, 3, 5]
methods = [1, 3, 3, 3, 3]
nArray = 10**(np.arange(5)+2)

for ii in range(len(orders)):

    conv = convergenceAbel(nArray, methods[ii], orders[ii])
    ax4.loglog(nArray, conv, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

ax4.legend()
ax4.set_xlabel('number of data points')
ax4.set_ylabel('relative error')
ax4.grid(True)


################################################################################
↪#####################################################
# Run times of different methods and orders

def runtimesAbel(nArray, nMeasure, method, order):

    runtimes = np.zeros(nArray.shape[0])
    runtimesPre = np.zeros(nArray.shape[0])

    for ii in range(nArray.shape[0]):

        dataIn = np.ones(nArray[ii])
        T = np.empty(nMeasure)
        for jj in range(nMeasure):
            t0 = ti.time()
            abelObj = oa.Abel(nArray[ii], -1, 0., 1., method = method, order = order)
            t1 = ti.time()
            T[jj] = t1-t0
        runtimesPre[ii] = np.sum(T)/nMeasure

        abelObj = oa.Abel(nArray[ii], -1, 0., 1., method = method, order = order)
        t0 = ti.time()
        for jj in range(nMeasure):
            dataOut = abelObj.execute(dataIn)
```

(continues on next page)

(continued from previous page)

```
        t1 = ti.time()

        runtimes[ii] = (t1-t0)/nMeasure

    return (runtimesPre, runtimes)


# Loop over several methods and orders
names = ['HL', 'FMM 3rd', 'FMM 11th', 'TD 1st']
orders = [-1, 3, 11, -1]
methods = [1, 3, 3, 0]
nArray = 10**(np.arange(5)+2)
nArraySmall = 10**(np.arange(3)+2)

for ii in range(3):

    (runtimesPre, runtimes) = runtimesAbel(nArray, 5, methods[ii], orders[ii])
    ax5.loglog(nArray, runtimesPre, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax6.loglog(nArray, runtimes, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

for ii in range(3,len(names)):

    (runtimesPre, runtimes) = runtimesAbel(nArraySmall, 5, methods[ii], orders[ii])
    ax5.loglog(nArraySmall, runtimesPre, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax6.loglog(nArraySmall, runtimes, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

ax5.legend()
ax5.set_xlabel('number of data points')
ax5.set_ylabel('run time pre computation in s')
ax5.grid(True)

ax6.legend()
ax6.set_xlabel('number of data points')
ax6.set_ylabel('run time main computation in s')
ax6.grid(True)


mpl.tight_layout()
mpl.savefig('example004_fullComparison.png', dpi=300)

mpl.show()
```

4.6 example005_comparisonPyAbel

This example provides a rather extensive comparison of different **openAbel** methods with PyAbel methods. It shows how well the main methods of **openAbel** perform in every regard, especially if the input data is sufficiently smooth.

Since **PyAbel**'s focus is on the backward (or inverse) transform, this example does it as well.

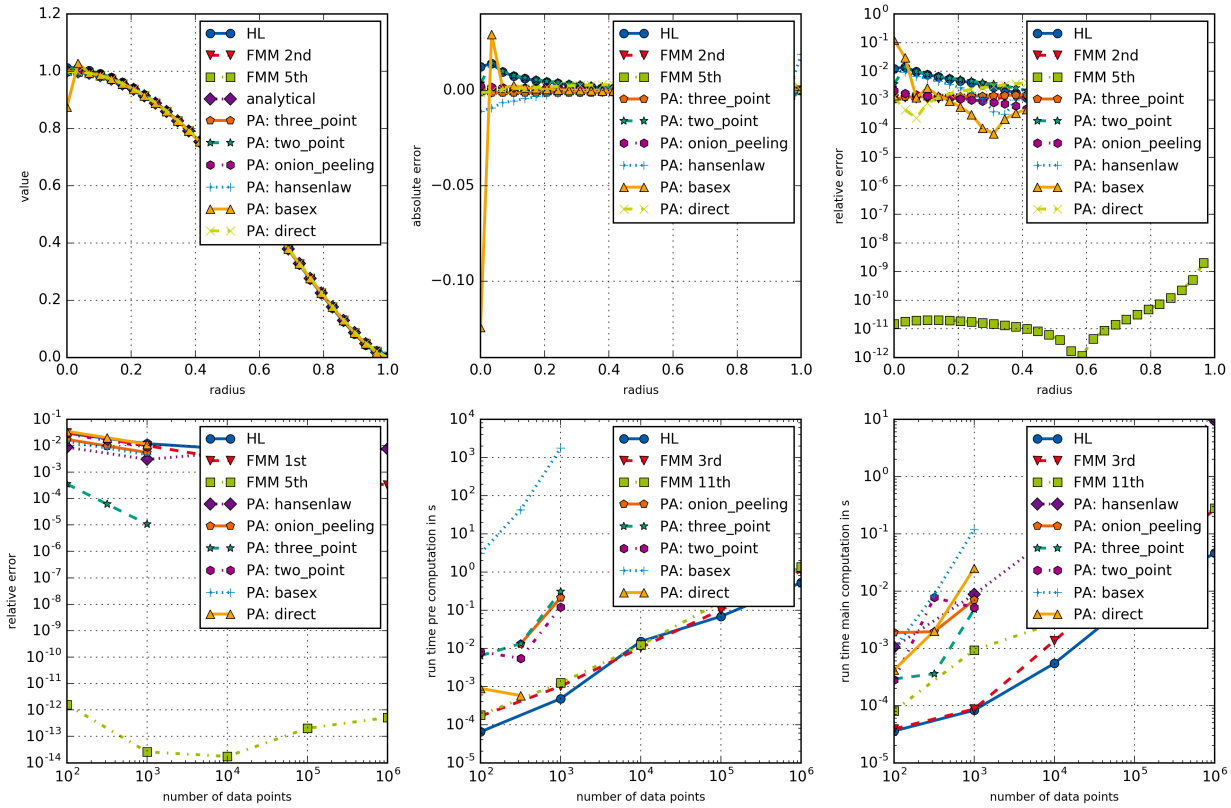

Fig. 6: Comparison of different openAbel with PyAbel methods.

```
##############################################################################################################
# Example to showcase the different openAbel methods in comparison with PyAbel methods.
# Backward transform is done here since PyAbel focuses on that.
# One obviously needs PyAbel installed for this example
##############################################################################################################


import openAbel as oa
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as mpl
import time as ti
import abel
import os, glob

```

(continues on next page)

(continued from previous page)

```
################################################################################################
↪#####################################################
# Plotting setup

params = {
   'axes.labelsize': 8,
   'font.size': 8,
   'legend.fontsize': 10,
   'xtick.labelsize': 10,
   'ytick.labelsize': 10,
   'text.usetex': False,
   'figure.figsize': [12., 8.]
   }
mpl.rcParams.update(params)
# Color scheme
colors = ['#005AA9','#E6001A','#99C000','#721085','#EC6500','#009D81','#A60084','
↪#0083CC','#F5A300','#C9D400','#FDCA00']
# Plot markers
markers = ["o", "v" , "s", "D", "p", "*", "h", "+", "^", "x"]
# Line styles
linestyles = ['-', '--', '-.', ':','-', '--', '-.', ':','-', '--', '-.', ':']
lw = 2

fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = mpl.subplots(2, 3)


################################################################################################
↪#####################################################
# Error over radius of different methods and orders

def errorAbel(nData, method, order):

    dx = 1./(nData-1);
    xx = np.linspace(0., 1., nData)

    dataIn = 3./8.*np.pi*(1-xx**2)**2
    dataAna = np.sqrt(1-xx**2)**3
    abelObj = oa.Abel(nData, 1, 0., dx, method = method, order = order)
    dataOut = abelObj.execute(dataIn)
    abserr = dataOut-dataAna
    relerr = np.abs(abserr/np.clip(dataAna, 1.e-300, None))

    return (xx, abserr, relerr, dataOut, dataAna)


# Loop over several methods and orders
names = ['HL', 'FMM 2nd', 'FMM 5th']
orders = [-1, 2, 5]
methods = [1, 3, 3]

for ii in range(len(orders)):

    (xx, abserr, relerr, dataOut, dataAna) = errorAbel(30, methods[ii], orders[ii])
    ax1.plot(xx, dataOut, label=str(names[ii]), color = colors[ii],
             linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax2.plot(xx, abserr, label=str(names[ii]), color = colors[ii],
                 linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
```

(continues on next page)

(continued from previous page)

```
    ax3.semilogy(xx[:-1], relerr[:-1], label=str(names[ii]), color = colors[ii],
                 linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

ii += 1
ax1.plot(xx, dataAna, label = 'analytical', color = colors[ii],
         linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)


def errorAbelPyAbel(nData, method, function):

    dx = 1./(nData-1);
    xx = np.linspace(0., 1., nData)

    dataIn = 3./8.*np.pi*(1-xx**2)**2
    dataAna = np.sqrt(1-xx**2)**3
    dataOut = function(dataIn, basis_dir='.', dr=dx, direction="inverse")
    abserr = dataOut-dataAna
    relerr = np.abs(abserr/np.clip(dataAna, 1.e-300, None))

    return (xx, abserr, relerr, dataOut, dataAna)

jj = ii+1
# Loop over several methods and orders
methods = ['three_point', 'two_point', 'onion_peeling',
           'hansenlaw', 'basex', 'direct']
functions = [abel.dasch.three_point_transform, abel.dasch.two_point_transform, abel.
↪dasch.onion_peeling_transform,
             abel.hansenlaw.hansenlaw_transform, abel.basex.basex_transform, abel.
↪direct.direct_transform]
for ii in range(len(methods)):

    (xx, abserr, relerr, dataOut, dataAna) = errorAbelPyAbel(30, methods[ii],␣
↪functions[ii])
    ax1.plot(xx, dataOut, label='PA: '+str(methods[ii]), color = colors[jj+ii],
             linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)
    ax2.plot(xx, abserr, label='PA: '+str(methods[ii]), color = colors[jj+ii],
                 linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)
    ax3.semilogy(xx[:-1], relerr[:-1], label='PA: '+str(methods[ii]), color =␣
↪colors[jj+ii],
                 linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)


ax1.legend()
ax1.set_xlabel('radius')
ax1.set_ylabel('value')
ax1.grid(True)

ax2.legend()
ax2.set_xlabel('radius')
ax2.set_ylabel('absolute error')
ax2.grid(True)

ax3.legend()
ax3.set_xlabel('radius')
ax3.set_ylabel('relative error')
ax3.grid(True)

```

(continues on next page)

(continued from previous page)

```
################################################################################################
↪#####################################################
# Convergence of different methods

def convergenceAbel(nArray, method, order):

    conv = np.empty(nArray.shape[0])
    for ii in range(nArray.shape[0]):

        nData = nArray[ii]
        dx = 1./(nData-1);
        xx = np.linspace(0., 1., nData)

        dataIn = 3./8.*np.pi*(1-xx**2)**2
        abelObj = oa.Abel(nData, 1, 0., dx, method = method, order = order)
        dataOut = abelObj.execute(dataIn)
        dataAna = np.sqrt(1-xx**2)**3

        conv[ii] = np.sqrt(np.sum(((dataOut[:-1]-dataAna[:-1])/dataAna[:-1])**2)/
↪(nData-1))

    return conv


# Loop over several methods and orders
names = ['HL', 'FMM 1st', 'FMM 5th']
orders = [-1, 1, 5]
methods = [1, 3, 3]
nArray = 10**(np.arange(5)+2)

for ii in range(len(orders)):
    conv = convergenceAbel(nArray, methods[ii], orders[ii])
    ax4.loglog(nArray, conv, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

def convergenceAbelPyAbel(nArray, method, function):

    conv = np.empty(nArray.shape[0])
    for ii in range(nArray.shape[0]):

        nData = nArray[ii]
        dx = 1./(nData-1);
        xx = np.linspace(0., 1., nData)

        dataIn = 3./8.*np.pi*(1-xx**2)**2
        dataOut = function(dataIn, basis_dir='.', dr=dx, direction="inverse")
        dataAna = np.sqrt(1-xx**2)**3

        conv[ii] = np.sqrt(np.sum(((dataOut[:-1]-dataAna[:-1])/dataAna[:-1])**2)/
↪(nData-1))

    return conv


jj = ii+1
# Loop over several methods
methods = ['hansenlaw', 'onion_peeling', 'three_point', 'two_point', 'basex', 'direct
↪']
```

(continues on next page)

(continued from previous page)

```
functions = [abel.hansenlaw.hansenlaw_transform, abel.dasch.onion_peeling_transform, abel.dasch.three_point_transform,
             abel.dasch.two_point_transform, abel.basex.basex_transform, abel.direct.direct_transform]
for ii in range(1):
    conv = convergenceAbelPyAbel(nArray, methods[ii], functions[ii])
    ax4.loglog(nArray, conv, label='PA: '+str(methods[ii]), color = colors[jj+ii],
               linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)

nArray = (10**(np.arange(4)*0.5+2)+0.5).astype(int)
for ii in range(1,len(methods)):
    conv = convergenceAbelPyAbel(nArray, methods[ii], functions[ii])
    ax4.loglog(nArray, conv, label='PA: '+str(methods[ii]), color = colors[jj+ii],
               linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)


ax4.legend()
ax4.set_xlabel('number of data points')
ax4.set_ylabel('relative error')
ax4.grid(True)


################################################################################################################
# Run times of different methods and orders

def runtimesAbel(nArray, nMeasure, method, order):

    runtimes = np.zeros(nArray.shape[0])
    runtimesPre = np.zeros(nArray.shape[0])

    for ii in range(nArray.shape[0]):

        dataIn = np.ones(nArray[ii])
        T = np.empty(nMeasure)
        for jj in range(nMeasure):
            t0 = ti.time()
            abelObj = oa.Abel(nArray[ii], 1, 0., 1., method = method, order = order)
            t1 = ti.time()
            T[jj] = t1-t0
        runtimesPre[ii] = np.sum(T)/nMeasure

        abelObj = oa.Abel(nArray[ii], 1, 0., 1., method = method, order = order)
        t0 = ti.time()
        for jj in range(nMeasure):
            dataOut = abelObj.execute(dataIn)
        t1 = ti.time()

        runtimes[ii] = (t1-t0)/nMeasure

    return (runtimesPre, runtimes)


# Loop over several methods and orders
names = ['HL', 'FMM 3rd', 'FMM 11th']
orders = [-1, 3, 11]
methods = [1, 3, 3]
```

(continues on next page)

(continued from previous page)

```
nArray = 10**(np.arange(5)+2)

for ii in range(3):

    (runtimesPre, runtimes) = runtimesAbel(nArray, 1, methods[ii], orders[ii])
    ax5.loglog(nArray, runtimesPre, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax6.loglog(nArray, runtimes, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)

nArray = 10**(np.arange(4)+2)
for ii in range(3,len(names)):

    (runtimesPre, runtimes) = runtimesAbel(nArray, 1, methods[ii], orders[ii])
    ax5.loglog(nArray, runtimesPre, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)
    ax6.loglog(nArray, runtimes, label=str(names[ii]), color = colors[ii],
               linestyle = linestyles[ii], marker = markers[ii], linewidth=lw)


def runtimesAbelPyAbel(nArray, nMeasure, method, function):

    runtimes = np.zeros(nArray.shape[0])
    runtimesPre = np.zeros(nArray.shape[0])

    for ii in range(nArray.shape[0]):

        dataIn = np.ones(nArray[ii])
        T = np.empty(nMeasure)
        for jj in range(nMeasure):
            for filename in glob.glob(method + '*'):
                os.remove(filename)
            t0 = ti.time()
            dataOut = function(dataIn, basis_dir='.', dr=1., direction="inverse")
            t1 = ti.time()
            T[jj] = t1-t0
        runtimesPre[ii] = np.sum(T)/nMeasure

        t0 = ti.time()
        for jj in range(nMeasure):
            dataOut = function(dataIn, basis_dir='.', dr=1., direction="inverse")
        t1 = ti.time()

        runtimes[ii] = (t1-t0)/nMeasure
        runtimesPre[ii] -= runtimes[ii]

    return (runtimesPre, runtimes)

jj = ii+1
# Loop over several methods
methods = ['hansenlaw', 'onion_peeling', 'three_point', 'two_point', 'basex', 'direct
↪']
functions = [abel.hansenlaw.hansenlaw_transform, abel.dasch.onion_peeling_transform,␣
↪abel.dasch.three_point_transform,
             abel.dasch.two_point_transform, abel.basex.basex_transform, abel.direct.
↪direct_transform]
nArray = 10**(np.arange(5)+2)
```

(continues on next page)

(continued from previous page)

```
for ii in range(1):
    (runtimesPre, runtimes) = runtimesAbelPyAbel(nArray, 1, methods[ii], functions[ii])
    ax6.loglog(nArray, runtimes, label='PA: '+str(methods[ii]), color = colors[jj+ii],
               linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)

nArray = (10**(np.arange(4)*0.5+2)+0.5).astype(int)
for ii in range(1,len(methods)):
    (runtimesPre, runtimes) = runtimesAbelPyAbel(nArray, 1, methods[ii], functions[ii])
    ax5.loglog(nArray, runtimesPre, label='PA: '+str(methods[ii]), color = colors[jj+ii],
               linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)
    ax6.loglog(nArray, runtimes, label='PA: '+str(methods[ii]), color = colors[jj+ii],
               linestyle = linestyles[jj+ii], marker = markers[jj+ii], linewidth=lw)

ax5.legend()
ax5.set_xlabel('number of data points')
ax5.set_ylabel('run time pre computation in s')
ax5.grid(True)

ax6.legend()
ax6.set_xlabel('number of data points')
ax6.set_ylabel('run time main computation in s')
ax6.grid(True)


mpl.tight_layout()
mpl.savefig('example005_comparisonPyAbel.png', dpi=300)

mpl.show()
```

CHAPTER 5

Remarks

In this section I want to mainly give some fairly informal remarks. Many are on different Abel transform algorithms described in literature, and for those I'm giving the reasoning on why these algorithms were not useful to me. The remaining remarks are just a collection of findings of me related to the Abel transform I thought worth mentioning. Some of the remarks are fairly subjective for the use cases I was and am interested in. Furthermore, while this section is meant to be fairly informal, I try to be direct, especially since I feel like many publications are somewhat misleading, e.g. algorithms are often mislabeled as "fast" or "similar to the Fast Fourier Transform" even though the computational complexity is nowhere near similar to the Fast Fourier Transform.

5.1 PyAbel Python Package

There is a Python package called PyAbel, which focuses mainly on the inverse (or backward/reconstruction/etc.) Abel transform. It's open source and pretty well documented and has a several nice and communicative developers which are active on the **PyAbel** github repository.

If the reader is interested in testing many of the algorithms mentioned in this remarks section, I can only recommend to look at **PyAbel**, as it does implement more algorithms for the inverse Abel transform than **openAbel** (basically all except the main recommended one in **openAbel**) . The **example_005** of **openAbel** uses **PyAbel** to compare many of the algorithms talked about here as well.

5.2 Hansen-Law Method

Although the Hansen-Law method is still implemented in **openAbel**, it unfortunately has one major flaw (and one slightly problematic one). I think in principle the space state model approximation is a pretty clever idea. That's why I actually tried to improve the algorithm or apply the basic idea to other integrals (both to no success, yet). Unfortunately several things have to fall into place for it to work nicely, which isn't the case for the Abel Transform.

The first approximation which Hansen and Law make is – roughly speaking – that they implicitly approximate the data by a piecewise polynomial, specifically piecewise linear in the original formulation. Going to higher order gets messy quickly, but is in principle possible. I successfully tried that, but due to the next approximation (which is somewhat flawed) it's not useful.

The second approximation is to rewrite the Abel transform kernel and approximate it by a sum of exponentials. At first it looks like one could get a linear computational complexity $O(N)$ algorithm. Problem is that the kernel has (even after rewriting) a singularity, so it's obviously pretty difficult to approximate a singularity by a sum of exponentials (again I tried many published approaches for that; most have flaws as well or are at least difficult and don't lead to good enough results). Increasing the number of exponentials used increases the computational complexity. My guess is that the algorithm is $O(Nlog(N))$ at best because of the increasing number of required exponentials. In practice it turns out it's pretty much impossible to get anything really universally useful, unless one aims only for fairly large errors (like Hansen-Law with roughly 10^{-3}). For many use cases this is probably enough (e.g. experimental ones which are usually fairly noisy anyway), so the method has still some value. But even if that is the case and one ignores every problem I mentioned here the method is still not more efficient than the main **openAbel** methods, so it's never the best choice as long as one does not have to implement the algorithms (FMM is a lot more work to implement than Hansen-Law). And I have to mention that I did a lot of thinking on the topic, and for example one could subtract the singularity in the impulse response, but this usually leads to not useful algorithms or the use of basically the same approaches and algorithms as **openAbel** to make it competitive, just one is taking a huge detour.

5.3 Desingularized Quadrature

Like the Hansen-Law method the desingularized quadrature is implemented in **openAbel**, but still mostly a remnant of testing different Abel transform methods and I don't recommend using it. The basic idea of desingularizing integrals should be familiar to almost anyone who tried to numerically integrate a function with a singularity and wanted to improve convergence. In context of a problem similar to the Abel transform it is discussed by Tausch, which is one of the main references for the Fast Multipole Method in **openAbel** as well. I actually tried several higher order versions of this, and the effort is not really worth it, since the end corrections used in **openAbel** are much more efficient. And it gets very complicated – maybe impossible – to program if one tries to avoid instabilities; I'm actually not sure if my test implementations were definitely reliable. And of course this method, it's $O(N^2)$, is slower than the main **openAbel** methods.

5.4 Piecewise Polynomial Analytic Integration

There are many different publications which basically use the same idea: Interpolate the data by a piecewise polynomial and use the known analytic integral for every polynomial piece. Dasch calls it onion peeling or filtered back projection, Bordas does basically the same, and there are probably more publications. I actually thought initially when I decided to use the Fast Multipole Method, that the piecewise polynomial analytic integration would be useful in combination. In principle it works, but is again pretty messy and the end corrections used in **openAbel** are overall much more efficient in every sense once implemented. But without the Fast Multipole Method it's a slow $O(N^2)$ method as well, and that slow way is what all publications do to my knowledge.

5.5 Analytic Integration of a Basis Set Expansion

There are many publications which use some kind of basis set expansion applied to the data, then use analytic transform of each basis function to construct the total transform. So this is similar to **Piecewise Polynomial Analytic Integration**, but with a basis for the whole domain and not just piecewise.

Often a polynomial basis set expansion for the whole data set and then transform each basis polynomial analytically. This obviously only works well (regarding error) if the data has somewhat polynomial behavior regarding the whole domain. Since one can chose orthogonal polynomial basis sets the expansion is at least fairly fast, but since the analytical transforms of polynomials are not very nice this approach overall is not very efficient. In a specific case where the data is basically a low order polynomial this of course would work really well, but in the general case it's not useful.

Other basis sets might have nicer transforms, but are not orthogonal, so the expansion of the data is more difficult. I tried to find some "good" basis, but in one way or another one shifts the difficulty to another area, e.g. function approximation, and I did not get a useful approach. Again, for some very specific data sets one might find a very small but usable basis set.

One example of such an algorithm in literature is the BASEX algorithm by Dribinski. In this method a "Gaussian" basis set – just to note it's somewhat Gaussian, not the "normal" Gaussian – is used. It seems to be very popular, as the publication has 791 citations as of writing this. I'm guessing mainly because the code was freely available and the basis set implicitly applied some smoothing in the transform, which usually produces nicer pictures without tweaking than other algorithms. I'm fairly convinced that one can achieve similarly nice results with other methods and some smoothing. Similar to other methods described here the method can be tested in **PyAbel**. The preprocessing is incredibly painfully slow (it's $O(N^3)$ I think, and it takes minutes for even small arrays N=1000, where **openAbel**'s main methods are $O(N)$ and take milliseconds), and the actual transform is not much better ($O(N^2)$ and **openAbel** is $O(N)$ again). Overall **BASEX** is a fairly often cited algorithm nevertheless.

I can see how in some cases one might be able to chose a nicely suitable basis set to enforce some structure in either the projected or reconstructed data. I expect this would be the only case where such an approach would make sense, but this is very problem specific and thus much less universal than the main methods of **openAbel** intend to be. One example of such an approach is described by Gerber, and often called linBASEX (e.g. in **PyAbel**). Due to the underlying physical process Gerber expects or knows that his data has some structure, and enforces it by choosing a specific basis set. In the general case one could probably achieve similar results by fitting the expected structure basis set to the data and then using accurate black-box Abel transform functions like in **openAbel**.