openConv Documentation

Release 0.1

Oliver Sebastian Haas

Jan 09, 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

openConv README

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

1.1 Introduction

The main goal of **openConv** is to provide fast and efficient numerical convolutions of symmetric and smooth kernels and data of equispaced data in Python with all actual calculations done in Cython. It is intended to work in conjunction with openAbel to calculate 2D convolutions of radially symmetric functions in atomic collisions. The most useful methods implemented in my module for that purpose use the Fast Multipole Method combined with arbitrary order end correction of the trapezoidal rule to achieve both fast convergence and linear run time. Other methods are implemented for comparisons.

1.2 Quick Start

In most cases this should be pretty simple:

- Clone the repository: `git clone https://github.com/oliverhaas/openConv.git`
- Install: `sudo python setup.py install`
- Run example: `python example000_exponential.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
- FFTW3 3.3.4

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 **openConv** to run on vastly different systems, like e.g. Windows systems, but obviously I haven't extensively tested different setups.

1.4 Issues

In contrast to other codes I made available, **openConv** has as of now only very specific use-cases I actually needed, thus implemented and debugged. I strongly recommend every user to thourougly check if the methods work as intended for their specific problem. For most people **openConv** will thus not be a useable code as is, but more a starting point or inspiration for their own code. If there are any issues, bugs or feature request just let me know. Gaps in the implementation might be filled by me if requested.

1.5 Transform Methods

It is fairly common to use directly the discrete convolution to approximate the convolution integral, often with smaller improvements like using trapezoidal rule instead of rectangle rule. This yields usually neither good order of convergence (second order with trapezoidal rule), nor fast calculation (quadratic computational complexity). **openConv** intends to provide methods to calculate these convolutions efficiently, fast, and with high accuracy. Beside the common "fast convolution" algorithm based on the Fast Fourier Transform we provide methods based on the Fast Multipole Method and high order end correction, which outclass common methods in many cases in most aspects (convergence order, error, computational complexity, etc.), as long as the kernel is smooth.

For the most important methods of we adapted the Chebyshev interpolation Fast Multipole Method (FMM) as described by Tausch and calculated end corrections for smooth functions 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 an linear *O(N)* computational complexity algorithm. For approximately exponentially decaying functions, like e.g. often encountered in atomic physics, we introduced an exponential shift into the Chebyshev interpolation to get relative errors of the convolution result of up to machine precision.

In both error and computational complexity there is no better existing method for the intended purpose to my knowledge.

In the documentation and the examples more details are discussed and mentioned; in general both are a good way to learn how to understand and use the code.

1.6 Copyright and License

CHAPTER 2

Convolution Methods

The convolution integral in one dimension is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \ ,$$

and the discrete equivalent (which implies uniformly discretized data and kernel)

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m].$$

Often both f and g often have some kind of symmetry around 0; **openConv** deals mainly with both f and g having some kind of symmetry. In the examples some more details are discussed and mentioned; in general the examples are a good way to learn how to understand and use the code.

```
import openConv
convObj = oc.Conv(nData, symData, kern, kernFun, symKern, stepSize, nResult, method =␣
↪method, order = order)
result = convObj.execute(data)
```

2.1 Direct Convolution / Trapezoidal Rule

It is fairly common to use directly the discrete convolution to approximate the convolution integral, often with smaller improvements like using trapezoidal rule instead of rectangle rule. Especially relevant in case both f and g are smooth functions, this yields usually neither good order of convergence (second order with trapezoidal rule), nor fast calculation (quadratic *O(N^2)* computational complexity).

Direct convolution is chosen by setting `method=0`.

2.2 FFT Convolution

One option to get a faster calculation is instead of direct calculation is use of the Fast Fourier Transform (FFT) based convolution, or often called "fast convolution" algorithm. This approach gives linearithmic *O(Nlog(N))* computational

complexity, but possibly large relative and unpredictable errors of the result, since the error scales with the maximum value of the result. In case of functions with high dynamic range, e.g. exponential functions, the tails of the results are poorly resolved.

FFT convolution is chosen by setting `method=1`.

2.3 Fast Multipole Method with Chebyshev Interpolation

Even better computational complexity (linear *O(N)*) can be achieved by the Fast Multipole Method (FMM). There are so called black box FMM described in literature (e.g. by Tausch), which in principle work well for smooth kernels with not too high dynamic range.

FMM convolution is chosen by setting `method=2`.

2.4 Fast Multipole Method with Chebyshev Interpolation for Approximately Exponential Functions

In **openConv** we extend the FMM to functions with asymptotic somewhat exponential decay and thus can deal with a large class of functions with high dynamic range.

FMMEXP is chosen by setting `method=3`.

2.5 End Corrections

To increase the order of convergence **openConv** uses end corrections for the trapezoidal rule as described in the reference by Kapur. These end corrections can be used together with every convolution method by setting the keyword `order` to the desired order. If data points outside of the integration interval can be provided these end corrections are arbitrary order stable. Otherwise it is not recommended to go higher than 5th order. As of now we provide the coefficients up to 20th order. The Mathematica notebook which calculated these coefficients can be found in this repository as well.

CHAPTER 3

Examples

3.1 example000_exponential

This is a simple example which calculates the convolution of a Gaussian with a symmetric approximately exponential kernel.

```
1  ##############################################################################
   ↪#####################################################
2  # Simple example which calculates the convolution of two somewhat exponential␣
   ↪functions.
3  # Results are compared with the direct solution. Mostly default parameters are used.
4  ##############################################################################
   ↪#####################################################
5
6
7  import openConv as oc
8  import numpy as np
9  import matplotlib.pyplot as mpl
10
11
12 ##############################################################################
   ↪#####################################################
13 # Plotting setup
14
15 params = {
16    'axes.labelsize': 8,
17    'font.size': 8,
18    'legend.fontsize': 10,
19    'xtick.labelsize': 10,
20    'ytick.labelsize': 10,
21    'text.usetex': False,
22    'figure.figsize': [12., 8.]
23    }
24 mpl.rcParams.update(params)
```

(continues on next page)

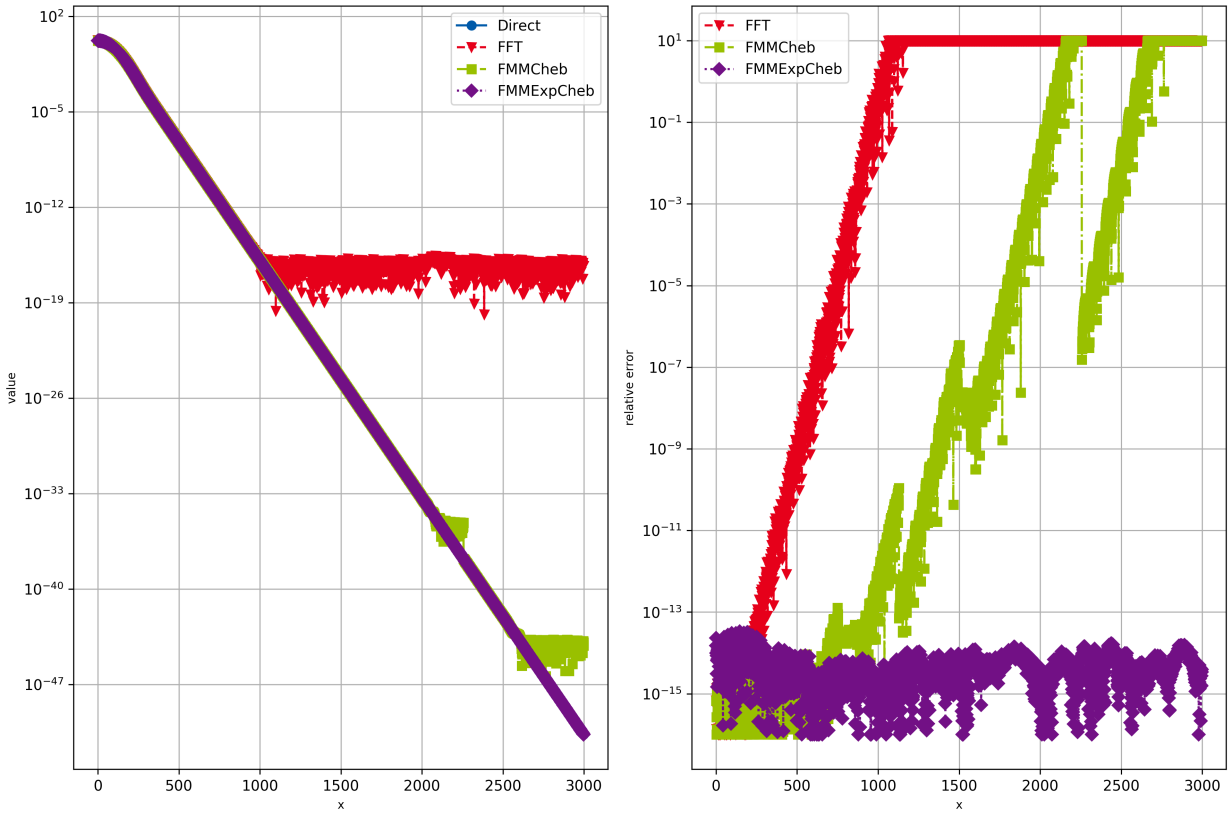

Fig. 1: Simple convolution with somewhat exponential kernel.

(continued from previous page)

```
# 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)) = mpl.subplots(1, 2)

################################################################################################################################
# Parameters and input data
order = 5
orderM1Half = max(int((order-1)/2),0)

nData = 1000
xMaxData = 8.
sigData = 0.5
stepSize = xMaxData/(nData-1)
xData = np.linspace(-orderM1Half*stepSize, orderM1Half*stepSize+xMaxData, nData+2*orderM1Half)
data = np.exp(-0.5*xData**2/sigData**2)     # data can actually be arbitrary

# Kernel
lamKernel = 0.2
def kern(xx):
    return np.exp(-xx/lamKernel) + 10.*np.exp(-3.*xx/lamKernel)
nKernel = 2000


# Parameters and output result
nResult = nData+nKernel-1   # Can be chosen arbitrary, but use typical length for example
xResult = np.linspace(0., (nResult-1)*stepSize, nResult)

xKernel = np.linspace(0., (nResult+nData-2)*stepSize, nResult+nData-1)
kernel = kern(xKernel)


################################################################################################################################
# Create convolution object, which does all precomputation possible without knowing the exact
# data. This way it's much faster if repeated convolutions with the same kernel are done.
convObj = oc.Conv(nData, 2, kern, None, 2, stepSize, nResult, method = 0, order = order)
result = convObj.execute(data, leftBoundary = 3, rightBoundary = 3)

convObj = oc.Conv(nData, 2, kern, None, 2, stepSize, nResult, method = 1, order = order)
result2 = convObj.execute(data, leftBoundary = 3, rightBoundary = 3)

convObj = oc.Conv(nData, 2, kern, None, 2, stepSize, nResult, method = 2, order = order)
```

(continues on next page)

(continued from previous page)

```
result3 = convObj.execute(data, leftBoundary = 3, rightBoundary = 3)

convObj = oc.Conv(nData, 2, kern, None, 2, stepSize, nResult, method = 3, order =␣
↪order)
result4 = convObj.execute(data, leftBoundary = 3, rightBoundary = 3)

ax1.semilogy(xResult/stepSize, np.abs(result), color = colors[0], marker=markers[0],␣
↪linestyle=linestyles[0], label='Direct')
ax1.semilogy(xResult/stepSize, np.abs(result2), color = colors[1], marker=markers[1],␣
↪linestyle=linestyles[1], label='FFT')
ax1.semilogy(xResult/stepSize, np.abs(result3), color = colors[2], marker=markers[2],␣
↪linestyle=linestyles[2], label='FMMCheb')
ax1.semilogy(xResult/stepSize, np.abs(result4), color = colors[3], marker=markers[3],␣
↪linestyle=linestyles[3], label='FMMExpCheb')
ax1.legend()
ax1.set_xlabel('x')
ax1.set_ylabel('value')
ax1.grid(True)

ax2.semilogy(xResult[:-1]/stepSize, np.clip(np.abs((result2[:-1]-result[:-1])/
↪result[:-1]),1.e-16,10.), color = colors[1], marker=markers[1],␣
↪linestyle=linestyles[1], label='FFT')
ax2.semilogy(xResult[:-1]/stepSize, np.clip(np.abs((result3[:-1]-result[:-1])/
↪result[:-1]),1.e-16,10.), color = colors[2], marker=markers[2],␣
↪linestyle=linestyles[2], label='FMMCheb')
ax2.semilogy(xResult[:-1]/stepSize, np.clip(np.abs((result4[:-1]-result[:-1])/
↪result[:-1]),1.e-16,10.), color = colors[3], marker=markers[3],␣
↪linestyle=linestyles[3], label='FMMExpCheb')
ax2.legend()
ax2.set_xlabel('x')
ax2.set_ylabel('relative error')
ax2.grid(True)

mpl.tight_layout()
mpl.savefig('example000_exponential.png', dpi=300)

mpl.show()
```