DeconvOptim.jl - Signal Deconvolution with Julia

Felix Wechsler\textsuperscript{1,2} and Rainer Heintzmann\textsuperscript{1,2,3}

\textsuperscript{1}Faculty of Physics and Astronomy, Friedrich-Schiller-University, Jena, Germany
\textsuperscript{2}Leibniz Institute of Photonic Technology, Albert-Einstein-Straße 9, 07745 Jena, Germany
\textsuperscript{3}Institute of Physical Chemistry and Abbe Center of Photonics, Friedrich-Schiller-University, Helmholtzweg 4, Jena, Germany

1. Introduction

Deconvolution has been a long addressed problem especially in astronomical imaging, digital signal processing or microscopy imaging. The assumption is that images are degraded by some kind of blur which can be described as a convolution of a kernel \( h \) with the image \( S \):

\[
I(r) = (h * S)(r)
\]

where \( * \) denotes a convolution operation. \( I \) indicates the ideal measured image, \( r \) is the position vector or the pixel positions in the discrete case. \( h \) is often called the point spread function (PSF) since it characterizes how the systems maps a point in the object to a blurred spot in the image. Often this convolution operation is disturbed by different types of noise where the most common are additive Gaussian noise (e.g. due to the read out of the camera sensors) or Poisson shot noise (e.g. due to the quantum nature of photons):

\[
Y_{\text{Poisson}}(r) = \text{Poisson}[(h * S)(r)]
\]

\[
Y_{\text{Gaussian}}(r) = (h * S)(r) + \text{Gauss}_e(r)
\]

Due to noise and the low pass filter effect of the PSF, deconvolution is an ill-posed problem which cannot be solved directly. A well known approach, because of its simplicity, has been the iterative Richardson-Lucy deconvolution algorithm \([5, 12]\) for Poisson noise degraded signals. Another common method is the non-iterative Wiener filter \([19]\) because of its predictable runtime.

\section{Efficient Convolutions}

The mathematical definition of a convolution is

\[
(h * S)(r) = \int_{-\infty}^{\infty} h(r - x) \cdot S(x) \, dx.
\]

To obtain efficient deconvolution algorithms we also need an efficient way to calculate the convolution operations. Today, convolution operations of spatially small kernels \( h \) can be calculated on GPUs very efficiently in real space. However, often the blurring kernel \( h \) is not spatially small and might be also multi-dimensional. In that case, FFT based convolutions outperform sliding kernels. In microscopy and astronomy that is often the case, since a defocus broadens the PSF.

\subsection{FFT Based Convolution}

The basis for fast convolution calculation via Fourier transforms lies in the convolution theorem. It states that

\[
F\{(h * S)\}(k) = F\{h\}(k) \cdot F\{S\}(k),
\]

which means that a Fourier transform \( F \) of a convolution operation is equivalent to a point wise multiplication of the Fourier transforms of \( h \) and \( S \). Fortunately, the Fourier transform operations can be calculated even for moderate large 3D arrays quickly on modern computing machines via the Fast Fourier Transform (FFT). In Julia, FFTW.jl offers a convenient interface to the FFTW library \([6]\). The time complexity to calculate Equation 5 is \( O(N \cdot \log N) \) for a one dimensional discretized dataset with \( N \) samples. For a 2D dataset with size \( N \times M \) it is \( O(N \cdot M \log(N \cdot M)) \) and accordingly for higher dimensions.
with size $K \times L$ applied on an image with $N \times M$ data points have a runtime of $O(N \cdot M \cdot K \cdot L)$. It is clear that for larger kernel sizes ($K, L \gg 1$) the sliding method has worse computational complexity than the FFT based convolutions. The many cores of a GPU can mitigate that to a certain extent but since the FFT (CuFFT in the CUDA library [15]) can be also executed in parallel fashion, both approaches profit from parallelization.

2.2 Wrap-Around Artifacts in FFT based Convolutions

One drawback of a FFT based convolution are wrap-around artifacts (see arrow in Figure 1a). The reason is, that the FFT operation calculates the discrete Fourier transform (DFT) which inherently assumes periodic data. If one applies a FFT on a finite object the assumption is that this finite object repeats itself. To weaken the wrap-around artifacts in the convolution one can zero or mean pad the image before convolution and remove the padding afterwards. Such approaches still suffer from artifacts when boundary values do not fit to the mean or to zero as it can be observed in Figure 1b where the arrow is pointing to. To solve for the wrap-around artifacts in deconvolution we use a different approach which will be presented in subsection 3.2.

2.3 Preferred Convolution

The current state of DeconvOptim.jl uses the FFT approach to calculate the convolution. However, in principle one could add the possibility to choose spatially based convolutions. But since the runtime gets especially long in multi-dimensional signal deconvolution where FFT based methods are faster, we always use FFT based convolutions here.

3. Inverse Modelling

In Figure 2 we show our approach to deconvolution. The general idea is to create a (physical) forward model describing the measurement process.

At the beginning of the optimization we start with an initial guess $x(r)$. Via a mapping function we can impose some restrictions to the final solution (e.g. non-negativity, value intervals). The forward model is then applied to the mapped reconstruction. We provide a default function where the forward function is a convolution but we also allow to plug in a different forward model. This output of the forward function is compared to our noisy measurement under a given loss function. Since deconvolution is an ill-posed problem, we regularize the loss function with a regularizer. Afterwards we calculate the gradient of the loss function with respect to the reconstruction $x(r)$ via automatic differentiation (AD). The values of the total loss and the gradient are then used by an optimizer (to minimize the total loss) to apply a gradient descent optimization to get a proper reconstruction.

Except FFTW.jl and CUDA.jl, the entire package is written in Julia. Therefore it is easy to replace the forward model with any Julia code, as long as we can automatically differentiate through the code. The gradient step is currently calculated with the reverse mode AD of Zygote.jl [10]. As default for the optimization we use the L-BFGS [12] routine provided by Optim.jl [13]. L-BFGS is suitable for optimization with many million parameters as it only stores a sparse representation of the inverse Hessian matrix, which is built from a finite number of past function and gradient evaluations. Results are often good after 10 to 50 iterations whereas Richardson-Lucy takes a few hundreds iterations. However, as we will see later due to the more complex optimization routine the speed-up in using L-BFGS is for a simple deconvolution minor.

3.1 Mapping Function

Mapping functions are an easy way to constrain the resulting reconstruction to predefined ranges. For example, if the optimizer optimizes the variables rec which are then squared, we know that $rec^2$ is definitely non-negative. Instead of directly using fwd(rec) we compare fwd(rec)$^2$ with the measurement $Y$ where fwd is the function describing the physical operation (mostly a convolution). This mapping function does not change the outcome as long as it is an surjective function for the desired number space (e.g. a parabola is non-negative and covers all positive values) and does not change the convexity of a problem. Functions like shifted tanh are possible.
as well as since it is an bijective function in $[-1,1]$ and monotonic increasing in the whole definition space. Hence the final result is the

\[ \arg \min_{S(r)} (- \log(p(Y(r)|\mu(r)))) = \]  

\[ \arg \min_{S(r)} \sum_r \mu(r) + \log(\Gamma(Y(r) + 1)) - Y(r) \log(\mu(r)) \]  

which is equivalent to

\[ \arg \min_{S(r)} L = \arg \min_{S(r)} \sum_r (\mu(r) - Y(r) \log(\mu(r))) \]  

since the second term only depends on the constant $Y(r)$ but not on $\mu(r)$. The gradient of $L$ with respect to $\mu(r)$ is simply

\[ \nabla L = 1 - \frac{Y(r)}{\mu(r)} \]  

The function $L$ and the gradient $\nabla L$ are needed for any gradient descent optimization algorithm. The numerical evaluation of the Poisson loss can lead to issues. Since $\mu(r) = 0$ can happen for a measurement with zero intensity background. However, the loss is not defined for $\mu \leq 0$. In our source code we set all intensity values below a predefined threshold $\epsilon$ to $\epsilon$ itself. This prevents the evaluation of the logarithm at undefined values. The final source code has a modified version of Equation 10 as registered gradient for Equation 9 The reason is, that for values of $\mu(r) = 0$ the equation is not defined and has to be handled separately.

### 4.2 Regularizer

The regularizer is an additional part in the total loss to add some additional priors like sparsity and smoothness. Most real samples are smooth over large regions and such regularizers enhance the quality of the reconstruction. In our toolbox we currently implement several regularizer like (smoothed) Total Variation, Good’s roughness and Tikhonov.

#### 4.2.1 Total Variation as example.

As the name suggests, Total Variation tries to penalize variation in the image intensity. Therefore it sums up the gradient strength at each point of the image:

\[ \text{Reg}(S(r)) = \sum_r |(\nabla S)(r)| \]  

Since we look at the magnitude of the gradient strength, this regularizer is anisotropic. In 2D TV is defined like:

\[ \text{Reg}(S(r)) = \sum_{x,y} \left[ |S(x+1,y) - S(x,y)|^2 + |S(x,y+1) - S(x,y)|^2 \right]^{\frac{1}{2}} \]  

In many frameworks, such a regularizer is implemented via functions like circshift.

Listing 1: Smoothed Total Variation

```julia
function total_variation(arr, \*elttype(arr)(1.0-\))
    @tullio r = sqrt((abs2(arr[1,j] - arr[i+1,j]) + abs2(arr[1,j] - arr[i,j+1]))
end
```

However, such approaches create often several copies of the data. We currently use Tullio.jl to create efficient functions calculating the regularizer and the gradient. For simple expressions such
as below. Tullio.jl registers an analytical gradient and therefore allows to apply the same efficient mechanisms to the gradient calculation which have been used for the regularizer evaluation itself. A (smoothed) Total Variation with Tullio.jl is defined in

We have chosen a position interval which is well within the full series so that there are not wrap-around artifacts in the measurement. Note, that definition just demonstrates how the regularizer can be written down. In our toolbox we allow for much more options. Since Tullio.jl cannot handle generic multi-dimensional data yet we create regularizers which can vary in step size, dimensionality and different weighting in dimensions via Julia’s metaprogramming capability.

A call like

```
 julia> using DeconvOptim
 julia> reg = TV(sum_dims=4, sum_dims=[1,2,3], weights=[1,1,2])
 #169 (generic function with 1 method)
```

results in a regularizer accepting four dimensional arrays. However, the regularization only happens along the first three dimension and the third dimensions is weighted differently. Such options are available for most of the regularizers implemented in our toolbox.

5. Experiments

In this section we show some experiments and compare parts of the software to other solutions. All experiments were executed on Ubuntu 20.04 using a AMD Ryzen 5 5600X 6-Core processor (12 threads) with 32 GB DDR4 RAM and Julia 1.6.2. If possible, all threads were used. As GPU accelerator we used a GeForce RTX 2060 SUPER with 8 GB memory and CUDA 11.3 and CUDA.jl 3.3.1. Our experiments were done in Julia 1.6.2 under the use of BenchmarkTools.jl [5].

5.1 One-Dimensional Time Series

First, we want to demonstrate the deconvolution of an artificial one dimensional dataset blurred with a certain PSF. Additionally, we want to demonstrate the mechanism to reduce wrap-around artifacts in the deconvolution. On the left hand side in Figure 4 we can see a data series which has been simulated as a convolution with a PSF and degradation with additive Gaussian noise.

On the right hand we can see the results of the following deconvolution. The first statement applies a total padding of 1 meaning that the size is doubled and therefore the measured data is padded. As we can see, the blue curve in Figure 4b does not show these intense spikes at the boundaries as the green curve shows.

Also we notice that the deconvolved data in Figure 4b does not show all the details which are visible in the ground truth. The reason is that the PSF is a low pass filter and therefore only frequencies which have been passed through the filter can be enhanced in terms of contrast in a deconvolved result.

5.2 Influence of Regularizer

In this part we want to demonstrate the influence of the regularizer. Figure 5a is a simulated image which has been affected by Poisson shot noise and blurring by a PSF.

Fig. 5: a) is the ground truth image, b) has been convolved with a PSF and affected by Poisson shot noise (100 photons expected for the brightest pixel), c) is deconvolved for 50 iterations without a regularizer, d) for 10 iterations without a regularizer, e) converged with Good’s roughness regularizer and after 24 iterations with \( \lambda = 0.03 \).

This test chart is suited to judge the resolution of a reconstruction since the pieces of the Siemens star have decreasing spacing towards its middle which is equivalent to higher spatial frequencies. Figure 5c shows the result after 50 iterations of the deconvolution. We see that the image is heavily affected by some artifacts (the dotted patterns) which usually occur after too many iterations without a regularizer. Additionally, we notice that the core in the Siemens star is denser requiring higher spatial resolution to resolve. Figure 5d shows the results after 10 iterations of the deconvolution without a regularizer. We have chosen 10 iterations since the normal cross correlation (NCC) shows the maximum at 10 iterations. Optimizing for more iterations seems to introduce more and more
artifacts to the image as it can be seen visually as well. Figure 5c shows the results after 24 iterations of the deconvolution but regularized with a Good’s roughness regularizer. The total image quality is much better since the regularizer dampens the artifacts but also results in lower resolution. With the regularizer weight $\lambda$ one can adjust the regularization strength. In Figure 6a we show the relative energy regain $G_R$ (REG) [8]. The meaning of the REG is how well certain frequencies are reconstructed in comparison to the ground truth. $G_R = 1$ is a perfect reconstruction, $G_R < 0$ indicates that this frequency could not be recovered at all, $G_R > 0$ means that the reconstruction introduces wrong information at this frequencies which are further from the ground truth than not reconstructing these frequencies. We can see that the red measured curve shows the typical contrast loss until the maximum frequency $f_{max}$ (the PSF has zero transmittance over this frequency threshold). Beyond $f_{max}$ Poisson noise introduces fake frequencies. The green curve shows the REG of the deconvolution without a regularizer. For regions below $f_{max}$ the deconvolution enhances the contrast. However, around and beyond $f_{max}$ several artifacts are introduced. The regularized blue curve performs better and only some frequencies show a REG below zero. To some extent the regularized version can also recover information above $f_{max}$, which are not supported by the PSF. Figure 6b shows the NCC over the number of iterations. Especially the deconvolution without a regularizer suffers from poor performance after a high number of iterations. Either one has to stop early after a hand optimized iteration number (here 10) or use a regularizer. The regularized version seems to be robust against a higher number of iterations and even converges after 24. In conclusion we can say that both the regularized and the non-regularized versions produce good results. In the non-regularized one has to hand-optimize the number of iterations whereas in the regularized one has to optimize $\lambda$. However, without the ground truth image it is not clear which iteration or $\lambda$ is optimal and hence this depends on the user. Advantage of the non-regularized version is, that it performs more than twice as fast but as it can be seen in Figure 5d it shows slightly more artifacts in homogenous regions.

5.3 Multi-Dimensional Microscopy Data

To more completely demonstrate the features available in DecomOptim.jl, we deconvolve a four dimensional (three spatial dimensions and a color dimension) dataset where the measured volume consists of a 3D datasets with three different color channels, shown in Figure 7. Here we assumed a common single 3D PSF for all color channels. Julia’s broadcast mechanism is able to understand that the data has more dimensions that the PSF and expands the PSF so that a multiplication still works. We only need to specify for the FFT over which dimensions the convolution should happen. Clearly we can see that the deconvolution removed a lot of the background blur.

5.4 Performance Comparison

Since Julia is often advertised with great performance we compare parts and the full deconvolution to other implementations in terms of their total performance.

5.4.1 Regularizer. In Table 1 we can see the runtimes for different TV regularizer implementations. As we can see, Tullio.jl outperforms a plain Julia implementation by more than order of magnitude. Partially, that is caused by Tullio.jl being able to use of threads whereas Julia’s built in broadcast mechanism is not multi threaded. By plain Julia we are referring to an implementation which is written with standard Julia Base functions. The same plain Julia implementation was then used together with CUDA.jl to make use of the GPU capabilities. We see that there is a minor benefit in using a GPU for the regularizer instead of the Tullio.jl version on the CPU. Unfortunately, GPU support by Tullio.jl is still experimental and especially on the gradient step we could not achieve reasonable results with it. We also compare the perf-
roughly a factor of $2 - 3$ in both forward and gradient pass. However, on the CPU Tullio.jl is considerably faster. In general one could always squeeze more performance out of the gradient calculations if the kernels would be hand optimized. In our package we especially try avoid that because of the multi-dimensionality it is much more convenient to rely on AD and as shown above the performance is usually acceptable.

5.5 Full Deconvolution

In Table 3 we show the deconvolution time for a 3D dataset with a size of $512 \times 256 \times 128$. The basic settings of the different algorithms and packages are noted in Table 3. The aim of this comparison is not to compare the quality of the results but the computing time to achieve similar reconstructions. The algorithms used in DeconvOptim.jl are not fundamentally different to a plain Richardson-Lucy deconvolution and therefore the results will not exceed those results significantly in terms of quality.

One should keep in mind that such performance comparisons are usually biased since we are not familiar with all options which should be chosen to achieve best performance. The comparison should give a rough overview which runtime is expected for each toolbox.

The different packages used were ThreeDeconv.jl [9], Huygens Professional version 21.04 (Scientific Volume Imaging, The Netherlands, http://svi.nl), DeconvolutionLab2 [17] and GenericDeconv.

For the results of GenericDeconv (written in Matlab by Rainer Heintzmann) we used a Poisson loss function with a L-BFGS minimizer and a quadratic non-negativity constraint. When the TV regularizer was added the weight was $\lambda = 0.001$. The general idea of GenericDeconv is very similar to DeconvOptim.jl. DeconvolutionLab2 offers a wide variety of different algorithms which however only run on a CPU. In the table we show the results of an iterative Richardson-Lucy deconvolution. The regularizer weight was $\lambda = 0.001$.

In DeconvOptim.jl we measured the runtime of the default deconvolution routine with a Poisson loss function and the runtime of straightforward Richardson-Lucy deconvolution without a regularizer.

ThreeDeconv.jl is another Julia deconvolution toolbox for deconvolution of three dimensional data, the regularizer weight was $\lambda = 10^{-7}$. Since it is hard to judge the quality of those results with a numerical loss function value, we compare them by the NCC, see Figure 3. The CPU versions were multi threaded (if possible). DeconvolutionLab2, Huygens and GenericDeconv were tested under Windows 10 and not Ubuntu. As summary of those performance tests we can state that our Julia implementation is very competitive and especially on the GPU it was the fastest of the tested packages. We also see that our routine was faster than a plain Richardson-Lucy mainly due to the much lower number of iterations. On the CPU DeconvOptim.jl is only beaten by the commercial Huygens software. Despite ThreeDeconv.jl written in Julia it is notably slower in comparison to our implementation. DeconvolutionLab2 is written in Java and also uses hand written gradients but still its speed is poor in comparison to all other packages.

6. Summary

In conclusion, DeconvOptim.jl is a very performant, flexible deconvolution software to deconvolve multi-dimensional datasets. Its focus has been from the beginning to be fast and the source code should be automatic differentiated by the available packages in Julia. Usually we would expect that AD decreases performances but still our software is faster than many other packages which have been written with hand-optimized gradients. Furthermore, our toolbox is not restricted in dimensionality and has mechanisms to prevent wrap-around artifacts which occur with FFT based convolutions. Julia’s metaprogramming capabilities together with Tullio.jl allow to generate regularizers which have high performance and are easy to adapt. In future, we want to support more regularizers on GPUs since they can not be handled efficiently with the current Tullio.jl solution. Also we plan to extend the package to deconvolution of spatially varying kernels which are of importance in optical systems with large field of view or poor manufacturing quality.

7. Acknowledgement

This work was partially funded by the Honours Programme of the Friedrich Schiller University of Jena. We want to thank Michael Abbott because he had the initial idea for the metaprogramming regularizers based on Tullio.jl. Additionally, we want to thank Scientific Volume Imaging for providing a trial license for Huygens and especially Mark Koenis for advising us how to adjust the settings to get optimal performance.

---

1 Dataset was taken from [http://bigwww.epfl.ch/deconvolution/data/microtubules/][1]. During our experiments we noticed a pixel offset between the ground truth and the noisy image. To reproduce our results, one has to correct this offset if the images are compared quantitatively.

2 [https://www.uni-jena.de/en/honours_programme][2]
Fig. 8: Maximum intensity projections along one dimension of some of the deconvolution results.

8. References


