juliacon

DeconvOptim.jl - Signal Deconvolution with Julia

Felix Wechsler[1, 2] and Rainer Heintzmann[1, 2, 3]

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

ABSTRACT

Deconvolution is a versatile method to enhance the quality of signals measured with systems which can be expressed mathematically as a convolution of a system's response function with a signal. In this paper, we present `DeconvOptim.jl`, a flexible toolbox written in Julia to deconvolve one or multiple multi-dimensional signals which have been degraded by a multi-dimensional signal response function. `DeconvOptim.jl` works both on CPUs and GPUs and utilizes recent advancements in Julias automatic differentiation ecosystem.
In this work we demonstrate that `DeconvOptim.jl` surpasses the performance of existing open source libraries clearly and is applicable to one dimensional time series datasets but also to multi-dimensional microscopical imaging datasets.

Keywords

Julia, Image Processing, Deconvolution, Microscopy

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:

$$I(\mathbf{r}) = (h * S)(\mathbf{r}) \tag{1}$$

where $*$ denotes a convolution operation. 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):

$$Y_{\text{Poisson}}(\mathbf{r}) = \text{Poisson}[(h * S)(\mathbf{r})] \tag{2}$$
$$Y_{\text{Gauss}}(\mathbf{r}) = (h * S)(\mathbf{r}) + \text{Gauss}_{\sigma}(\mathbf{r}) \tag{3}$$

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 [18], [14] for Poisson noise degraded signals. Another common method is the non-iterative Wiener filter [21] because of its predictable runtime. The Wiener filter assumes additive noise which is often not a good approximation, especially in low light conditions as in microscopy or astronomy. Recently, machine learning based methods have become more and more favoured in deconvolution because of the possibility to add learned properties of similar images to the algorithm [12]. However, those implementations are usually restricted to cases where experimental datasets are available for the characterization and training. `DeconvOptim.jl` is a toolbox written in Julia [4] which offers several deconvolution algorithms like the iterative Richardson-Lucy algorithm but also much more flexible methods which are based on minimizing a noise-dependent loss function with the help of modern gradient-descent minimization routines. Also this toolbox is not restricted in dimensionality and therefore can be applied to deconvolution problems in many different fields.

2. Efficient Convolutions

The mathematical definition of a convolution is

$$(h * S)(\mathbf{r}) = \int_{-\infty}^{\infty} h(\mathbf{r} - \mathbf{x}) \cdot S(\mathbf{x})\, \mathrm{d}\mathbf{x}. \tag{4}$$

To obtain efficient deconvolution algorithms we also need a 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.

2.1 FFT Based Convolution

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

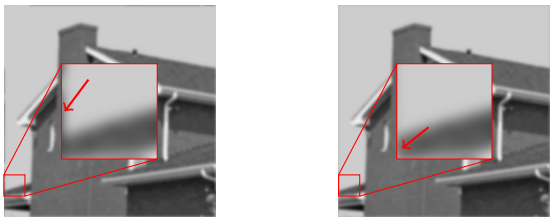
$$\mathcal{F}\left\{(h * S)\right\}(\mathbf{k}) = \mathcal{F}\{h\}(\mathbf{k}) \cdot \mathcal{F}\{S\}(\mathbf{k}), \tag{5}$$

which means that a Fourier transform $\mathcal{F}$ 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 [7]. The time complexity to calculate Equation 5 is $\mathcal{O}(N \cdot \log N)$ for a one dimensional discretized dataset with N samples. For a 2D dataset with size $N \times M$ it is $\mathcal{O}(N \cdot M \log(N \cdot M))$ and accordingly for higher dimensions. Spatially based convolution kernels with size $K \times L$ applied on an image with $N \times M$ data points have

a runtime of $\mathcal{O}(N \cdot M \cdot K \cdot L)$. 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 [16]) 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 data set the assumption is that this finite data set 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.



(a) Without padding (b) With padding.

Fig. 1: Comparison between two blurred images. The image a) is a straightforward FFT based blurred image. The zoomed region shows dark artifacts at the left side originated from the right hand side. Also at the top of the image some dark regions from the bottom scattered into the bright sky. Image b) was padded before blurring with 10 pixels on each side with value 0.7 which results in bright regions around the image. The blur kernel was a Gaussian kernel with standard deviation $\sigma = 3\,\text{pixel}$.

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 $x(\mathbf{r})$. 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 $x(\mathbf{r})$ 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.

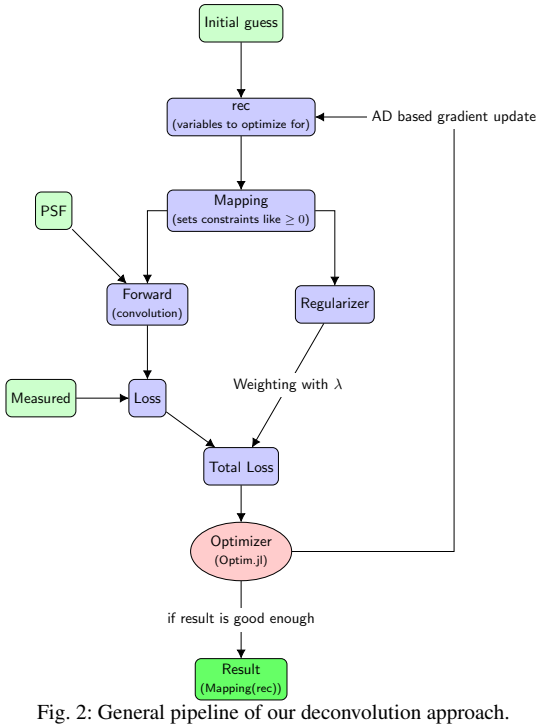


Fig. 2: General pipeline of our deconvolution approach.

As typical for Julia packages, except `FFTW.jl` and `CUDA.jl` [2], [3], all code 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` [11]. As default for the optimization we use the L-BFGS [13] routine provided by `Optim.jl` [15]. 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 $\text{fwd}(\text{rec})$ we compare $\text{fwd}(\text{rec}^2)$ 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 since it is an bijective function in $[-1, 1]$ and monotonic increasing in the whole definition space. Hence the final result is rec^2 in case of the quadratic mapping.

3.2 Wrap-around Artifacts Suppression

To prevent that wrap-around artifacts occur during deconvolution we can instead of reconstructing an object having the same size as the measurement, reconstruct an object having a slightly larger size. This is shown in Figure 3. Consequently, we only compare the grey

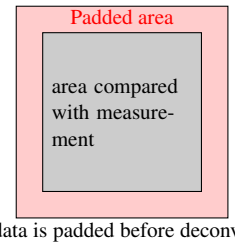


Fig. 3: Reconstructed data is padded before deconvolution but only the red region is compared with the measurement.

part of the reconstruction with the measurement. Through the AD and the convolution operations we still optimize for the whole area (including the red one). With that padding we reduce that images on the boundary influence in circular manner on reconstruction in the grey area.

4. Loss Function

Apart from forward model, the loss function is the most critical part in the optimization. It has to account for the noise created by the measurement process or instrument.

4.1 Poisson Data Term

Poisson shot noise is often the dominant source of noise hence we explain the details. The key is to interpret the measurement as a stochastic process. Our aim is to find a deconvolved image which predicts the measured image as accurately as possible. Mathematically the probability for a certain measurement Y under Poisson noise is

$$p(Y(\mathbf{r})|\mu(\mathbf{r})) = \prod_r \frac{\mu(\mathbf{r})^{Y(\mathbf{r})}}{\Gamma(Y(\mathbf{r})+1)} \exp(-\mu(\mathbf{r})) \tag{6}$$

where Y is the measurement, μ is the expected measurement (ideal measurement without noise) and Γ is the generalized factorial function. In the deconvolution process we get Y as measured input and want to find the ideal specimen S which results in a measurement $\mu(\mathbf{r}) = (S * \text{PSF})(\mathbf{r})$. Since we want to obtain the best reconstruction, we want to find a $\mu(\mathbf{r})$ so that $p(Y(\mathbf{r})|\mu(\mathbf{r}))$ gets maximized. Because that means that we find the specimen which describes the measurement with the highest probability. Instead of maximizing $p(Y(\mathbf{r})|\mu(\mathbf{r}))$ a common trick is to minimize $-\log(p(Y(\mathbf{r})|\mu(\mathbf{r})))$. Mathematically, the optimization of both functions provides same results but the latter is numerically more stable.

$$\arg\min_{S(\mathbf{r})}(-\log(p(Y(\mathbf{r})|\mu(\mathbf{r})))) = \tag{7}$$

$$\arg\min_{S(\mathbf{r})} \sum_r \mu(\mathbf{r}) + \log(\Gamma(Y(\mathbf{r})+1)) - Y(\mathbf{r})\log(\mu(\mathbf{r})) \tag{8}$$

which is equivalent to

$$\arg\min_{S(\mathbf{r})} L = \arg\min_{S(\mathbf{r})} \sum_r (\mu(\mathbf{r}) - Y(\mathbf{r})\log(\mu(\mathbf{r})) \tag{9}$$

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

$$\nabla L = 1 - \frac{Y(\mathbf{r})}{\mu(\mathbf{r})} \tag{10}$$

The function L and the gradient ∇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 ϵ to ϵ 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 $\mu(\mathbf{r}) = 0$ 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 [20], [8] 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:

$$\text{Reg}(S(\mathbf{r})) = \sum_r |(\nabla S)(\mathbf{r})| \tag{11}$$

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

$$\begin{aligned}\text{Reg}(S(\mathbf{r})) = \sum_{x,y} \big[&|S(x+1,y) - S(x,y)|^2 \\ &+ |S(x,y+1) - S(x,y)|^2\big]^{\frac{1}{2}}\end{aligned} \tag{12}$$

In many frameworks such a regularizer is implemented via functions like `circshift`. However, such approaches create often several copies of the data. We currently use `Tullio.jl`[1] 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 here:

```
function total_variation(arr, ϵ=eltype(arr)(1e-8))
    @tullio r = sqrt(ϵ +
                abs2(arr[i,j] - arr[i+1,j]) +
```

```
                    abs2(arr[i,j] - arr[i,j+1]))
end
```

ϵ is needed for regions where `arr == 0` because otherwise the gradient returns infinite values. 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(num_dims=4, sum_dims=[1,2,3],
                weights=[1,1,2])
#169 (generic function with 1 method)

julia> reg(randn((10,20,30,40)))
417810.11332404887
```

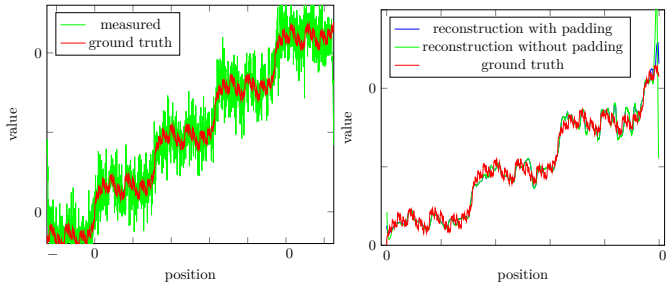
results in a regularizer accepting four dimensional arrays. However, the regularization only happens along the first three dimension and the third dimensions is weighted differently. Such options are available for most of the regularizers implemented in our toolbox.

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 1.6.1. 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 additionally made use of the following packages in Julia 1.6.2: ArrayInterface.jl[17], BenchmarkTools.jl[6], FFTW.jl[7], LLVM.jl[3], Optim.jl[15] and Zygote.jl[11].

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.



(a) (b)

Fig. 4: 1D data which has been convolved with a PSF and was degraded with additive Gaussian noise with $\sigma = 1.0$. Note that for the deconvolution we have chosen a position interval which is well within the full series so that there are not wrap-around artifacts in the measurement.

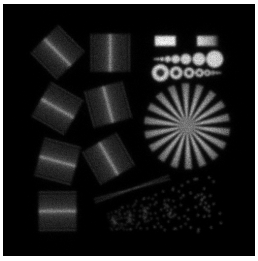
On the right hand we can see the results of the following deconvolution:

```
r_p, o = deconvolution(measured, psf, loss=Gauss(),
     iterations=12, regularizer=TV(num_dims=1),
     λ=0.05, padding=1)
r, o = deconvolution(measured, psf, loss=Gauss(),
     iterations=12, regularizer=TV(num_dims=1),
     λ=0.05, padding=0)
```

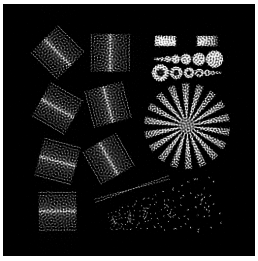
The first statement applies a total padding of 1 meaning that the size is doubled and therefore the measured data is padded. As we can see, the blue curve in Figure 4b does not show these intense spikes at the boundaries as the green curve shows. Also we notice that the deconvolved data in Figure 4b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

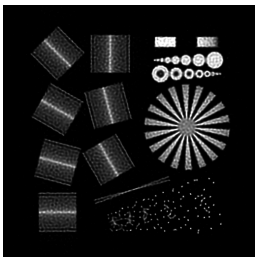
In this part we want to demonstrate the influence of the regularizer. Figure 5a is a simulated imaged which has been affected by Poisson shot noise and blurring by a PSF.
This test chart is suited to judge the resolution of a reconstruction since the pieces of the Siemens star have decreasing spacing 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 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 5c 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 5d 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 λ one can adjust the regularization strength. In Figure 6a we show the relative energy regain G_R (REG) [9]. The meaning of the REG is how well certain frequencies are reconstructed in comparison to the ground truth. $G_R = 1$ is a perfect



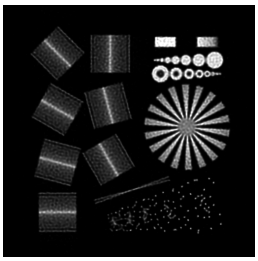
(a) Measured with Poisson noise



(b) Without regularizer $N = 50$



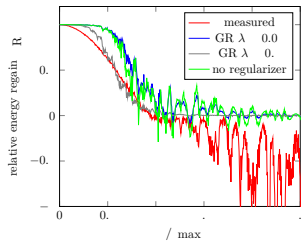
(c) Without regularizer $N = 10$



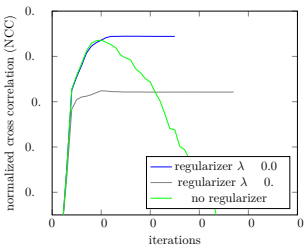
(d) With regularizer $N = 24$.

Fig. 5: a) has been convolved with a PSF and affected by Poisson shot noise (100 photons expected for the brightest pixel). Deconvolved b) for 50 iterations without a regularizer c) for 10 iterations without a regularizer d) with Good's roughness regularizer and converged after 24 iterations with $\lambda = 0.03$.

reconstruction, $G_\mathrm{R} = 0$ indicates that this frequency could not be recovered at all, $G_\mathrm{R} < 0$ 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



(a) REG G_R over different frequencies.



(b) NCC over the number of iterations.

Fig. 6: NCC and REG are quantitative measurements to judge the quality of a deconvolution result. In a) we show the REG for the hand-optimized non-regularized version with 10 iterations and the (converged) regularized versions with $\lambda = 0.03$ and $\lambda = 0.5$.

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 λ. However, without the ground truth image it is not clear which iteration or λ 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 5c it shows slightly more artifacts in homogenous regions.

5.3 Multi-Dimensional Microscopy Data

To more completely demonstrate the features available in `DeconvOptim.jl`, we deconvolve a four dimensional 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.

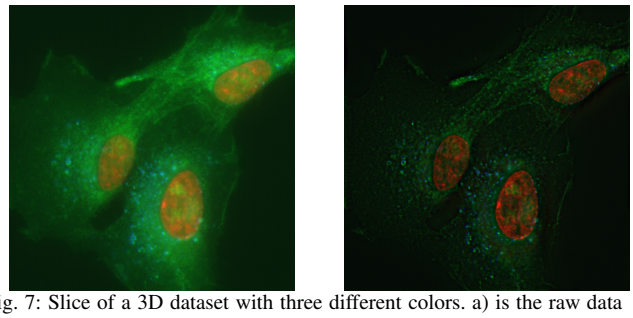


Fig. 7: Slice of a 3D dataset with three different colors. a) is the raw data and b) is the deconvolved image.

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 all 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 performance to an implementation written in Python with the Framework JAX [5]. The GPU version seems to outperform the Julia version by roughly a factor of $2-3$ in both forward and gradient pass. However, on the CPU `Tullio.jl` is considerably faster. In general one

	plain	∇ plain	`Tullio.jl`	∇ `Tullio.jl`
CPU	83.9 ms	7180 ms	5.53 ms	119.8 ms
GPU	2.5 ms	22.9 ms	4.00 ms	12 530 ms
	JAX	∇ JAX		
CPU	48 ms	364 ms		
GPU	0.887 ms	10.2 ms		

Table 1. : Runtimes of a single Total Variation regularizer calculation for different Julia versions and JAX. We used a 3 dimensional Float32 array with size $300 \times 300 \times 300$.

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 2 we show the deconvolution time for a 3D dataset with a size of $512 \times 256 \times 128$[1] 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` [10], `Huygens` Professional version 21.04 (Scientific Volume Imaging, The Netherlands, `http://svi.nl`), `DeconvolutionLab2` [19] and `GenericDeconv`.

	CPU in s	GPU in s	NCC
`DeconvOptim.jl`	32.9	2.20	0.86
`DeconvOptim.jl` - TV	39.9	4.46	0.79
Richardson-Lucy (Julia)	40.0	3.05	0.74
`ThreeDeconv.jl`	1550	27.1	0.85
`GenericDeconv`	140	6.32	0.86
`GenericDeconv` - TV	233	9.35	0.73
`Huygens`	30.1	6.8	0.86
`DeconvolutionLab2` - LR	1640		0.76

Table 2. : Runtimes of different deconvolution implementations. See Table 3 for the number of iterations for each method. We have chosen the same number of iterations for the deconvolutions with and without regularizers.

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 8. 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.

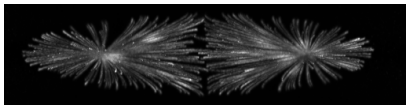
	Loss	iterations
`DeconvOptim.jl`	Poisson	45
`DeconvOptim.jl` TV	Poisson	45
Richardson-Lucy	Poisson	300
`ThreeDeconv.jl`	Gauss+Poisson	300
`GenericDeconv`	Poisson	45
`GenericDeconv` TV	Poisson	45
`Huygens`	Poisson	25
`DeconvolutionLab2` LR	Poisson	300

Table 3. : Overview which loss function (noise model) and number of iterations are used.

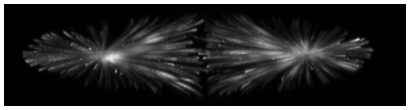
6. Summary

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 mechanism to prevent wrap-around artifacts which occur with FFT based convolutions. Julia's metaprogramming capabilities together with `Tullio.jl` allow to generate regularizer 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.

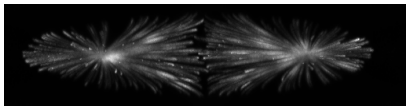
[1] Dataset was taken from `http://bigwww.epfl.ch/deconvolution/data/microtubules/`[19]. During our experiments we noticed a pixel offset between the ground truth and the noisy image. To reproduce our results, one has to correct this offset if the images are compared quantitatively.



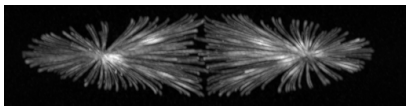
(a) `DeconvOptim.jl` . NCC $= 0.86$.



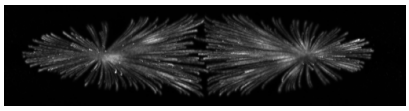
(b) Julia Richardson-Lucy. NCC $= 0.74$.



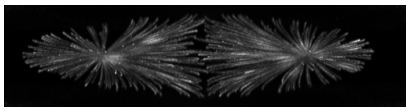
(c) `Deconvolution Lab2`. NCC $= 0.76$.



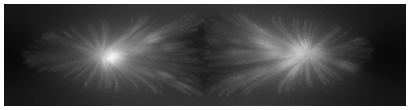
(d) `ThreeDeconv.jl`. NCC $= 0.85$.



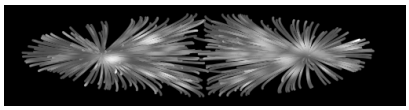
(e) `GenericDeconv`. NCC $= 0.86$.



(f) `Huygens`NCC $= 0.86$.



(g) Measured image. NCC $= 0.41$.



(h) Ground truth. NCC $= 1.0$

Fig. 8: Maximum intensity projections along one dimensions of some of the deconvolution results.

7. Acknowledgement

This work was partially funded by the Honours Programme of the Friedrich Schiller University of Jena[2]. 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.

[2] `https://www.uni-jena.de/en/honours_programme`

8. References

[1] Michael Abbott, Dilum Aluthge, N3N5, Simeon Schaub, Carlo Lucibello, Chris Elrod, and Johnny Chen. mcabbott/Tullio.jl: v0.3.0, June 2021.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[3] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. *GitHub*, 2018.

[6] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments, Aug 2016.

[7] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] I. J. Good and R. A. Gaskins. Nonparametric roughness penalties for probability densities. *Biometrika*, 58(2):255–277, 1971.

[9] Rainer Heintzmann. Estimating missing information by maximum likelihood deconvolution. *Micron*, 38(2):136–144, 2007. Special issue on Super-resolution and other Novel Microscopies.

[10] Hayato Ikoma, Michael Broxton, Takamasa Kudo, and Gordon Wetzstein. A convex 3d deconvolution algorithm for low photon count fluorescence imaging. *Scientific reports*, 8(1):11489, 2018.

[11] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.

[12] Jakob Kruse, Carsten Rother, and Uwe Schmidt. Learning to push the limits of efficient fft-based image deconvolution. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[13] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 45(1–3):503–528, August 1989.

[14] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 79:745, June 1974.

[15] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018.

[16] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[17] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[18] William Hadley Richardson. Bayesian-based iterative method of image restoration∗. *J. Opt. Soc. Am.*, 62(1):55–59, Jan 1972.

[19] D. Sage, L. Donati, F. Soulez, D. Fortun, G. Schmit, A. Seitz, R. Guiet, C. Vonesch, and M. Unser. DeconvolutionLab2: An open-source software for deconvolution microscopy. *Methods—Image Processing for Biologists*, 115:28–41, February 15, 2017.

[20] Peter J. Verveer and Thomas M. Jovin. Image restoration based on good's roughness penalty with application to fluorescence microscopy. *J. Opt. Soc. Am. A*, 15(5):1077–1083, May 1998.

[21] Nobert Wiener. *Extrapolation, Interpolation, and Smoothing of Stationary Time Series, with Engineering Applications*. Wiley, 1949.