Received: 14 December 2020 | Revised: 11 March 2021 | Accepted: 12 March 2021

DOI: 10.1002/mrm.28792

FULL PAPER

Magnetic Resonance in Medicine

MRIReco.jl: An MRI reconstruction framework written in Julia

Tobias Knopp[1,2] | **Mirco Grosser**[1,2]

[1]Institute for Biomedical Imaging, Hamburg University of Technology, Hamburg, Germany

[2]Section for Biomedical Imaging, University Medical Center Hamburg-Eppendorf, Hamburg, Germany

Correspondence
Tobias Knopp, Section for Biomedical Imaging, University Medical Center Hamburg-Eppendorf, Hamburg, Germany.
Email: tobias.knopp@tuhh.de

Purpose: The aim of this work is to develop a high-performance, flexible, and easy-to-use MRI reconstruction framework using the scientific programming language Julia.

Methods: Julia is a modern, general purpose programming language with strong features in the area of signal/image processing and numerical computing. It has a high-level syntax but still generates efficient machine code that is usually as fast as comparable C/C++ applications. In addition to the language features itself, Julia has a sophisticated package management system that makes proper modularization of functionality across different packages feasible. Our developed MRI reconstruction framework MRIReco.jl can therefore reuse existing functionality from other Julia packages and concentrate on the MRI-related parts. This includes common imaging operators and support for MRI raw data formats.

Results: MRIReco.jl is a simple to use framework with a high degree of accessibility. While providing a simple-to-use interface, many of its components can easily be extended and customized. The performance of MRIReco.jl is compared to the Berkeley Advanced Reconstruction Toolbox (BART) and we show that the Julia framework achieves comparable reconstruction speed as the popular C/C++ library.

Conclusions: Modern programming languages can bridge the gap between high performance and accessible implementations. MRIReco.jl leverages this fact and contributes a promising environment for future algorithmic development in MRI reconstruction.

KEYWORDS

image reconstruction, Julia, magnetic resonance imaging, numerical computing, open source

1 | INTRODUCTION

Magnetic resonance imaging (MRI) is a radiation-free tomographic imaging modality allowing for both high spatial resolution and high soft-tissue contrast. This, in combination with the large number of available contrasts, makes it an indispensable tool for many clinical imaging applications. In recent times, acquisition times and spatial resolution have been pushed further through the introduction of new, advanced signal processing techniques such as compressed sensing

(CS)[1,2] or structured matrix completion.[3-5] In a similar way, algorithmic developments have stimulated the development of new techniques for the quantitative imaging of tissue parameters, such as relaxation times, magnetic susceptibility, or apparent diffusion coefficients.[6-9] An important catalyst for this development is the availability of open source software. The latter facilitates the development of new methods while taking away the need of a full, self-implemented MRI signal processing pipeline.

An important aspect to keep in mind is that improvements in image quality and/or reduction in scan time are often only possible by computationally intensive algorithms shifting the bottleneck in latency from acquisition to reconstruction. To reduce the reconstruction time to a manageable level, massive parallelization including the usage of General Purpose Graphical Purpose Units (GPGPU)[10,11] hardware acceleration are often necessary. While this allows for acceleration of MRI reconstruction by a factor of 10-1000, it at the time increases the complexity of the software system.

Researchers developing new image reconstruction algorithms usually face 2 conflicting goals. On the one hand, one seeks a programming environment allowing for an easy translation of mathematical notation into programming code. On the other hand, the program should run as fast as possible. The first goal can be accomplished by using a high-level (HL) programming environment, albeit at the cost of performance. On the other hand, computation speed can be ensured by working in a low-level (LL) programming environment, which results in an increased system complexity. For this reason, one can observe that HL languages, such as Matlab and Python/Numpy,[12] are popularly used by mathematical/theoretical researchers. In contrast, the LL approach is often used by the vendor of an imaging system, since the product needs to run in a sufficiently short reconstruction time. In that case, the increase in system complexity is addressed by increasing the development resources.

A popular approach to reconcile both goals is to implement only hot loops in a LL language, while using a HL language for the remaining part. Popular examples for this hybrid approach are Matlab with its Mex interface, that is, a combination of Matlab and C/C++ and Python with its different ways to call C code (eg, ctypes). Note that the hybrid approach is not restricted to user code but that entire frameworks are built in this way. Examples include Numpy,[12] which is mainly written in C, and machine learning frameworks such as TensorFlow[13] and PyTorch,[14] which are implemented in C/C++ with Python language bindings.

The hybrid approach has not only many advantages but also has issues, which the authors in[15] summarize in the *two-language problem*:

- The LL implementation is capsuled away from the HL user interface. This is fine in most situations but becomes problematic if custom algorithms need to be implemented.
- Bridging from one language to the other often has a high computational cost and only amortizes for large problem sizes. Thus, code is often vectorized to minimize the number of calls into the LL language.

Beyond that it should be noted that two-language solutions are more complex to set up since multiple compilers and a proper management of dependencies are needed. In consequence, using an existing framework is easy whereas developing new methods/algorithms can be challenging as this often requires programming skills in the LL language used.

In the realm of MRI, the hybrid approach is very popular. Prominent examples are the packages Gadgetron[16] and BART,[17] which both have many features and follow the two-language principle with a core written in C/C++ and bindings to different HL languages. In contrast, SigPy[18] is a HL package entirely written in Python and allows for rapid prototyping of MRI reconstruction algorithms.

In this paper, we present an MRI reconstruction framework, which yields high performance while allowing for easy prototyping of new algorithms. The framework is implemented in a single programming language, thus avoiding the two-language problem. To achieve this, our framework uses the programming language Julia, which was invented by researchers at the MIT in 2012 with the aim of solving aforementioned two-language problem.[15] The aim of this paper is 2-fold. First, we describe our framework with its main components and we outline how it can be used for the development of new algorithms. Second, we perform comparisons in order to illustrate that the computation speed of our framework indeed matches that of popular frameworks such as BART.

2 | METHODS

When developing an MRI reconstruction framework, it is beneficial to formulate design goals to guide the design. For instance, the developers of Gadgetron formulated the following list of properties, which an MRI reconstruction framework should fulfill:

1. Free/open
2. Modular
3. Flexible
4. Cross platform
5. High performance
6. Facilitate prototyping
7. Facilitate deployment

For a proper definition of each of the design goals, we refer the reader to.16 MRIReco.jl has the same design goals although the deployment aspect is currently not yet addressed. Instead MRIReco.jl has the following additional design goals:

8. *Reuse of components*: The goal is to use as many existing software components as possible if and only if they are suitable for the task and appropriately maintained. This includes interdisciplinary standard tools such as the FFT and the conjugated gradients (CG) method. This design goal not only implies reusing existing software components but also to put non-MRI specific functionality into dedicated packages if such a package does not yet exist.
9. *Hackability*: It should be possible for software developers to access low-level parts of the software and develop extensions or even make modifications to the source code easily. *Hackability* only slightly overlaps with *prototyping* mentioned in.[16] In particular, it means that the gap between being a user and being a developer of a software framework is small.
10. *Accessibility*: The software components should be easy to install with only few instructions. It should be simple to access and modify the existing code.
11. *Testing*: The code should be properly tested using continuous integration services.

The last 2 design goals are certainly also followed by Gadgetron and BART. We note, however, that the two-language principle imposes limits on how well the first 2 of the additional design goals can be met. For instance, hackability is difficult to address in a two-language solution, since code modifications often require knowledge of both the HL and the LL language used.

To make design goal 8 more concrete, we note that image reconstruction frameworks often share a large number of generic building blocks. These components include common transformations such as the FFT, standard linear algebra tools, and optimization algorithms to solve the reconstruction problem at hand. In order to share these building blocks, it is advantageous to put them into dedicated software packages. This allows developers to reuse building blocks implemented by experts of the given field at hand and it reduces potential errors that could arise when self-implementing them. Finally, we note that the reuse of standard libraries also leads to a higher level of scrutiny for the latter. In turn, this decreases the likelihood of persistent bugs.

2.1 | Julia

To meet the design goals outlined in the previous section, we focus on a new programming language/environment called Julia. Julia was invented by researchers at the MIT in 2012[15] and has the aim of solving the two-language problem. Its most important features include:

- Julia uses a just-in-time (JIT) compiler based on the LLVM compiler infrastructure[19] and is therefore capable of generating efficient machine code.
- Julia has a sophisticated type system including type inference,[20] which allows the compiler to identify statically typed code fragments and in turn emit efficient machine code.
- Julia code can be both HL and LL. While hot code paths should be kept type stable for maximizing performance, dynamic programming can be used where convenient code is more important than high performance.
- The syntax of Julia is very similar to Matlab and thus familiar to a wide range of researchers.
- Julia can call C/C++ code with no overhead.

For more details on Julia, we refer the reader to Ref. [15].

Julia was designed from the ground up to be capable of generating efficient machine code. In combination with the HL features of the language, this allows us to develop a hackable framework (design goal 9) while retaining a high performance. In contrast, many dynamically typed languages like Python are challenging to JIT compile since they contain language features preventing emitting efficient machine code. Therefore, one of the most successful Python JIT compilers Numba[21] requires the user to manually annotate the code to be compiled, and only a subset of the language can be used. Here, again, performance is achieved by enabling the system to determine the types of all variables in a function body. A more general JIT compiler for Python not requiring code annotations is PyPy.[22]

Another feature of Julia is its built-in package manager. The latter can take care of all the dependencies of a code package. As a consequence, Julia packages are considered to be cheap and it is possible to achieve a fine grained modularization across packages. This modularization is pushed by a large community of package developers, which enrich the power of the Julia programming environment substantially. By exploiting this aspect of the Julia programming environment, *MRIReco.jl* manages to fulfill design goal 8 (reuse of components). Beyond that the clear versioning performed by the julia package manager can greatly simplify reproducible research. This stands in contrast to other approaches, where it is in the hand of the user to download the correct versions of the dependencies at hand.

2.2 | Functionality

MRIReco.jl offers a wide range of functionality while concentrating on basic building blocks that have proven useful

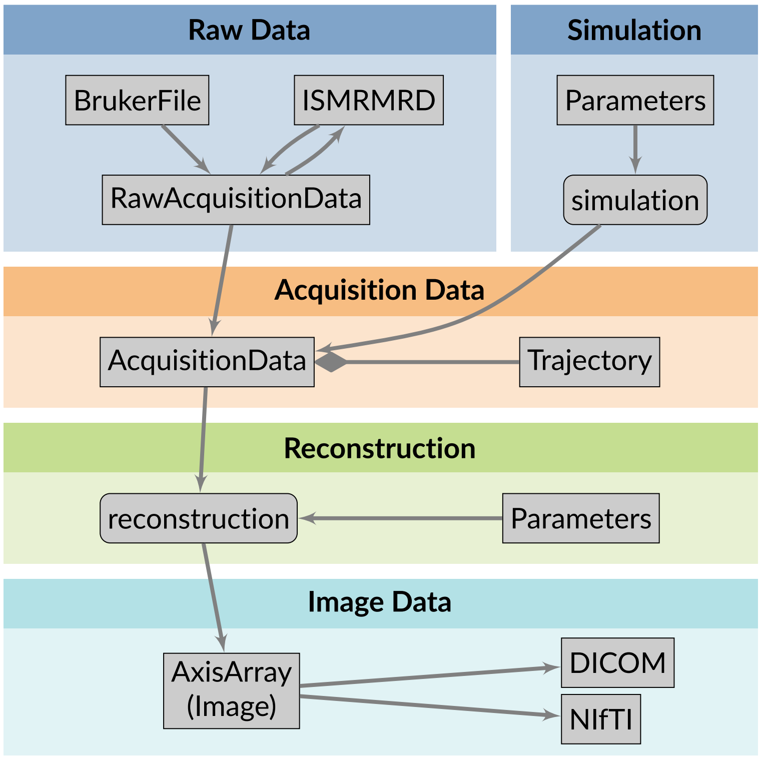


FIGURE 1 Overview of the internal architecture and the typical data flow in *MRIReco.jl*. An `AcquisitionData` object can be constructed either from a file through a `RawAcquisitionData` object or using `simulation`. The `AcquisitionData` object and additional reconstruction parameters are passed to the `reconstruction` method that yields an `AxisArray` object that can be stored in DICOM or NIfTI files

for MRI image reconstruction. In particular *MRIReco.jl* currently offers:

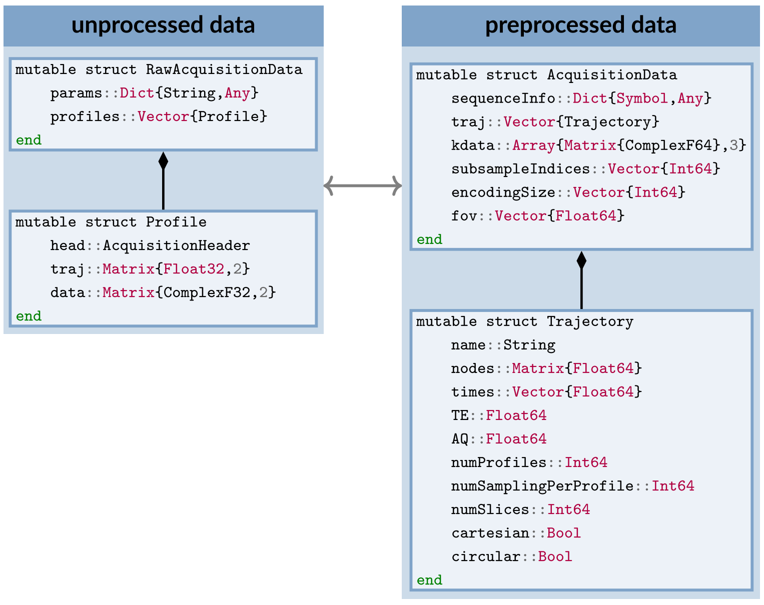
- *Simulation:* Methods for simulating MRI data based on software phantoms. This includes support for modeling B_0 inhomogeneity, R_2^* relaxation, and multiple receive coils. Moreover, basic support for simulating multi-echo sequences is implemented. Simulation can be performed using a direct evaluation of the imaging equation or using common approximations such as the NFFT.
- *Imaging operators:* Basic Cartesian and non-Cartesian imaging operators based on FFT and NFFT. Off-resonance and relaxation term aware imaging operators are available as well. These can be evaluated using both data-driven[23] and analytical[24,25] approximations. All operators can be combined to form encoding operators for more extended acquisitions such as parallel imaging[26] and multi-echo data acquisition.
- *Coil estimation: MRIReco.jl* implements methods for determining sensitivity maps including the sum-of-squares method and ESPIRiT.[27]
- *Iterative reconstruction:* Iterative reconstruction algorithms using solvers such as CGNR,[28] FISTA,[29] ADMM,[30] and the split Bregman method[31] are available. For regularization, *MRIReco.jl* offers TV, ℓ^1, ℓ^2 priors including wavelet sparsification. Density compensation weighting is available as a method for speeding up convergence of iterative solvers.[32,33] Sampling density weights can be computed for arbitrary trajectories based on the method described in Ref. [34]

MRIReco.jl is implemented in a Julia idiomatic way and uses multiple dispatch to execute different code paths depending on the type of function arguments. We note that this is a similar form of polymorphism known in classical object-orientated languages but that the dispatch is more generic. Since julia allows for dynamic programming, it was not necessary to introduce a further abstraction mechanism like a pipeline architecture.[16]

An overview of the internal architecture and a typical data flow within *MRIReco.jl* is given in Figure 1. In the following sections we sketch individual parts of the framework using small code snippets to illustrate some of the design principles. For more detailed information, we refer to the documentation and the source code of the package.

2.3 | Datatypes

In order to efficiently work with MRI data, *MRIReco.jl* introduces 2 distinct datatypes for its represention. `RawAcquisitionData` describes the data as it is stored in a data file. Since this data is typically not stored in a form suitable for reconstruction, it is first converted into the type `AcquisitionData`, describing the pre-processed data. The latter can then be passed to the reconstruction method in order to obtain an image. `Trajectory` is another important datatype, which is used for describing the MRI sampling trajectory. We outline each datatype in more detail next.



```
mutable struct RawAcquisitionData
    params::Dict{String,Any}
    profiles::Vector{Profile}
end
```

```
mutable struct Profile
    head::AcquisitionHeader
    traj::Matrix{Float32,2}
    data::Matrix{ComplexF32,2}
end
```

```
mutable struct AcquisitionData
    sequenceInfo::Dict{Symbol,Any}
    traj::Vector{Trajectory}
    kdata::Array{Matrix{ComplexF64},3}
    subsampleIndices::Vector{Int64}
    encodingSize::Vector{Int64}
    fov::Vector{Float64}
end
```

```
mutable struct Trajectory
    name::String
    nodes::Matrix{Float64}
    times::Vector{Float64}
    TE::Float64
    AQ::Float64
    numProfiles::Int64
    numSamplingPerProfile::Int64
    numSlices::Int64
    cartesian::Bool
    circular::Bool
end
```

FIGURE 2 Datatypes used to represent MRI data. The left-hand side contains the julia definition of the datatype `RawAcquisitionData` used for storing unprocessed MRI data. The right-hand side contains the corresponding definition of the type `AcquisitionData` used for describing the pre-processed MRI data

2.3.1 | Raw data

`RawAcquisitionData` is a datatype closely resembling the ISMRMRD data format.[35] Its Julia definition is contained in the left-hand side of Figure 2. The syntax `profiles::Vector{Profile}` means that the field `profiles` has the type `Vector{Profile}`. The philosophy of the ISMRMRD format is to store all global metadata in an XML header. This header has a static structure which can be extended by custom fields. `RawAcquisitionData` uses the `Dict params` (ie, an associative array) to store all data that are present in the XML header of an ISMRMRD file. If possible the content of each parameter is converted to an appropriate Julia datatype. For convenience, the structure of the ISMRMRD header is flattened, since this makes all parameters directly accessible. For instance, the parameter

```
<reconSpace>
  <matrixSize>
    <x>128</x>
    <y>128</y>
    <z>128</z>
  </matrixSize>
</reconSpace>
```

is available in the length-3 vector `head["reconSize"]`.

Each measurement profile is stored in the type `Profile`. It describes the data measured after a single excitation during an MRI experiment. It has members `head`, `traj`, and `data`, which exactly correspond to the structures specified by the ISMRMRD file format. The members of the `Profile` datatype are also bit-compatible with corresponding HDF5 structs in an ISMRMRD file.

2.3.2 | Pre-processed data

The goal of the `RawAcquisitionData` datatype is to have a very flexible representation for storing a great variety of measurements. As a matter of fact, this generic representation can be inconvenient when used for reconstruction. For instance, the profiles can be stored in a different order than required for image reconstruction. For this reason, *MRIReco.jl* uses an additional datatype `AcquisitionData` to store the data in a way convenient for reconstruction. The definition of `AcquisitionData` can be found on the right hand side of Figure 2. It contains the sequence information stored in a dictionary, the *k*-space trajectory, the *k*-space data, several parameters describing the dimensionality of the data, and some additional index vectors. The *k*-space data `kdata` has 3 dimensions encoding

1. contrasts/echoes
2. slices
3. repetitions

Each element is a matrix whose dimensions encode

1. *k*-space nodes
2. channels/coils

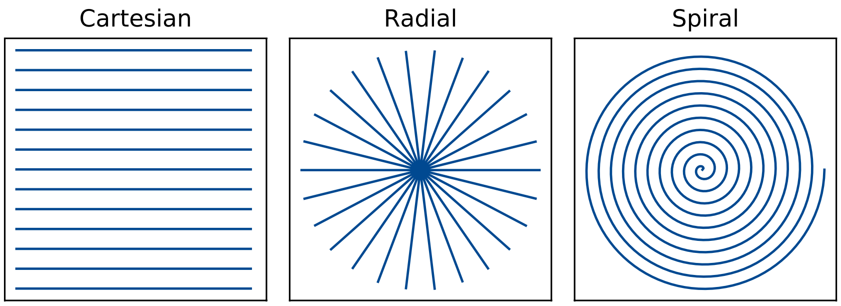


FIGURE 3 Exemplary trajectories available in *MRIReco.jl*

For undersampled data, the indices of the measured samples are stored in the field `subsampleIndices`. We note that both the `traj` and the `subsampleIndices` fields are defined as vectors with one entry for each contrast/echo.

The encoded space is stored in the field `encodingSize`. It is especially relevant for non-Cartesian trajectories where it is not clear upfront, how large the grid size for reconstruction should be chosen. Finally, the parameter `fov` describes the physical size of the encoding grid.

2.3.3 | Trajectory

The type `Trajectory` describes the sampling trajectory of the imaging sequence. Its definition is also contained on the right-hand side of Figure 2. Most importantly, the field `nodes` contains the sampling points as a matrix where the first dimension encodes the *k*-space dimensionality and the second dimension encodes the number of sampling points. This structure allows implementing arbitrary sampling trajectories. Moreover, *MRIReco.jl* provides constructors for the most common types of trajectories. These include Cartesian, radial, spiral, dual density spiral, variable density spiral, and perturbed spiral trajectories. Beside 2D trajectories, the framework also implements 3D trajectories like the kooshball or the stack of stars trajectory. A selection of exemplary trajectories is illustrated in Figure 3.

2.3.4 | Image data

MRIReco.jl returns image reconstruction results in the form of an `AxisArray`, which is a special datatype allowing to encode dimensions of an array. For image processing of reconstructed data one can use the package *Images.jl*. Storage of this data is possible using *NIfTi.jl*, *DICOM.jl* or *HDF5.jl*.

2.4 | Raw data file handling

The file handling in *MRIReco.jl* is build around the ISMRMRD file format for which full read and write support is implemented. In addition, a file reader for proprietary files from the vendor Bruker is implemented. The following code example shows how to convert a Bruker MRI file into an ISMRMRD file:

```
f = BrukerFile(filenameBruker) # create file handle
raw = RawAcquisitionData(f)     # load the data
fout = ISMRMRDFile("outputfile.h5") # create file handle
save(fout, raw) # store the data in the ISMRMRD file
```

We note that the raw data object `raw` can also be created by simulation. It is thus possible to cache simulated data, which is useful to save computation time associated with more complex simulations.

2.5 | Reconstruction building blocks

In general, MRI image reconstruction aims to recover a discrete image of the transverse magnetization $\boldsymbol{m} \in \mathbb{C}^N$ from a given set of measurements $\boldsymbol{s} \in \mathbb{C}^M$ and a signal encoding model described by a linear operator $\boldsymbol{H} \in \mathbb{C}^{M\times N}$. Thus, one seeks a solution to an inverse problem

$$\underset{\boldsymbol{m}}{\operatorname{argmin}} \|\boldsymbol{s} - \boldsymbol{H}\boldsymbol{m}\|_2^2 + \mathcal{R}(\boldsymbol{m}), \tag{1}$$

where $\mathcal{R}$ denotes a regularization function expressing prior knowledge about the solution. As a consequence most image reconstruction schemes can be formulated using a set of basic building blocks

- *Linear operators:* These are used to describe the signal encoding operator $\boldsymbol{H}$. Moreover, they describe transformations applied to $\boldsymbol{m}$ within the regularization term $\mathcal{R}(\boldsymbol{m})$ (eg, sparsifying transforms used in CS).

- *Solvers:* Optimization algorithms used to solve problem (1).
- *Proximal maps:* These are associated with the given regularization functions.

The interface in *MRIReco.jl* is designed so that all relevant parameters can be passed to the reconstruction method via a dictionary. The reconstruction method uses these parameters to form the main building blocks and solves the corresponding image reconstruction problem. In the following sections, we provide more information on the use and implementation of aforementioned building blocks.

2.5.1 | Linear operators

Linear operators are used in multiple places of a reconstruction pipeline. Most importantly, the signal encoding operator $\boldsymbol{H}$ is a discrete approximation of the underlying signal model

$$s_p(t) = \int_{\mathbb{R}^d} c_p(\boldsymbol{r}) m(\boldsymbol{r}) \mathrm{e}^{-z(\boldsymbol{r})t} \mathrm{e}^{-2\pi \boldsymbol{k}(t)\cdot \boldsymbol{r}} \, \mathrm{d}\boldsymbol{r}. \tag{2}$$

Here $s_p(t)$ denotes the demodulated signal received in the p-th of P receive coils at time t. Moreover, $m(\boldsymbol{r})$ is the transverse magnetization of the object at position $\boldsymbol{r}$ and $c_p(\boldsymbol{r})$ are the receive coil sensitivities. Finally, the term $z(\boldsymbol{r}) \in \mathbb{C}$ contains the R_2^* and B_0 maps in its respective real and imaginary parts. As a second application, linear operators can be applied to $\boldsymbol{m}$ in the regularizer $\mathcal{R}$. This commonly happens in compressed-sensing-type reconstructions, where the image needs to be transformed into a sparse representation.

All operators are implemented in a matrix-free manner. This means that they are characterized solely by their action when applied to a vector. This allows to evaluate operators using efficient algorithms, such as the FFT and NFFT, while avoiding storage of the underlying matrix representation. Other common matrix operations are implemented in complete analogy. For instance, one can apply a linear operator using the `*`-operator or form its adjoint using the postfix `'`. Similarly, linear operators can be composed using either of the operators `*` and `∘`, and their size can be determined using the `size`-function (We note that the term operator is used here for both the binary mathematical operation and the linear mapping).

MRIReco.jl provides implementations of the operators commonly used for MRI image reconstruction. These include

- `FFTOp`: A multidimensional FFT operator.
- `NFFTOp`: A multidimensional NFFT operator.
- `FieldmapNFFTOp`: An extension of the NFFT operator taking into account complex fieldmaps.
- `SensitivityOp`: An operator, which multiples an image by a set of coil-sensitivities as used in SENSE-type reconstructions.
- `SamplingOp`: An operator for (sub)sampling (k-space) data.
- `WeightingOp`: A weighting operator used for tasks such as sampling density compensation.

Additional operators, such as the Wavelet transform and finite differences operators, are reexported from the Julia package *SparsityOperators.jl*.

In addition to theses methods, *MRIReco.jl* provides high-level constructors that compose aforementioned operators and return signal encoding operators for the different reconstruction schemes. These are automatically called by the reconstruction methods implemented. Alternatively, each operator can be built manually by calling the corresponding constructor. In this way, the preimplemented operators can be used as building blocks when developing new algorithms.

Finally, we note that iterative solvers often require repeated application of the normal operator $\boldsymbol{N} = \boldsymbol{H}^H\boldsymbol{H}$ of the encoding operator. Thus, algorithms can sometimes be accelerated by optimizing the normal operator instead of optimizing the encoding operator itself. For instance, one can exploit the Toeplitz structure of $\boldsymbol{H}^H\boldsymbol{H}$ when $\boldsymbol{H}$ is an NFFT. To allow for this kind of optimization, *MRIReco.jl* reexports the type `normalOperator` from the packages *SparsityOperators.jl* and *RegularizedLeastSquares.jl*. When an optimized implementation of the normal operator exists, the latter can be used by simply overloading the constructor function `normalOperator`.

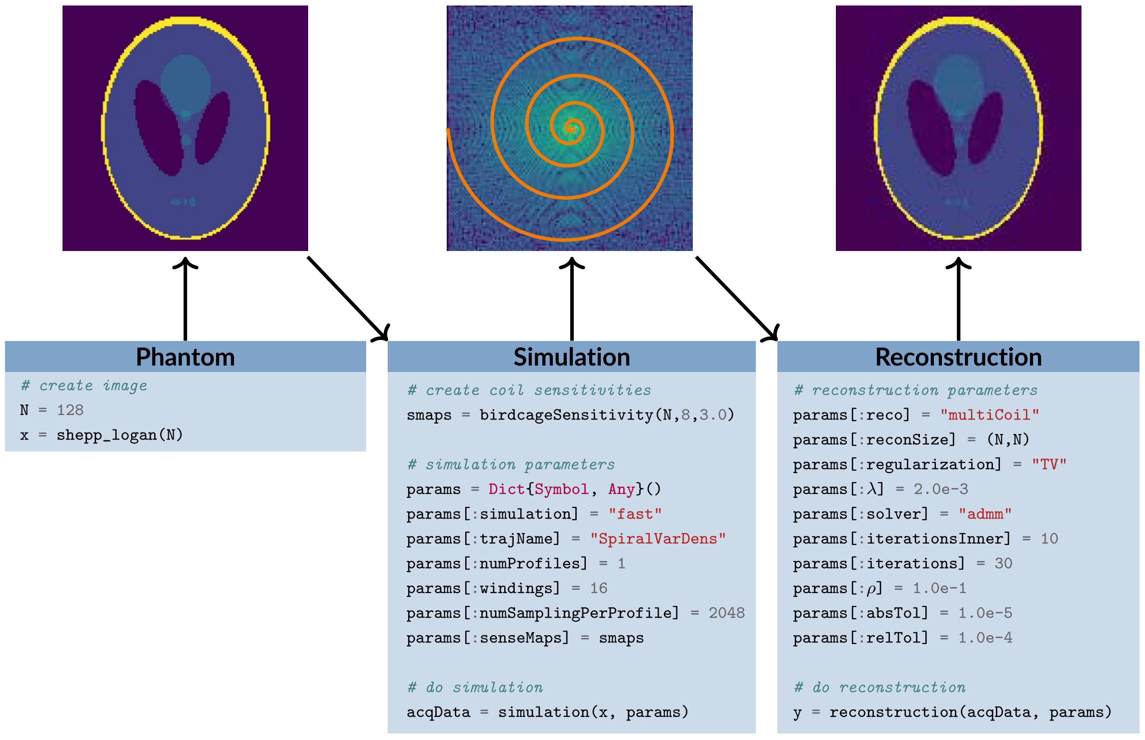
2.5.2 | Solvers

In order to solve the reconstruction problem at hand, *MRIReco.jl* uses the infrastructure provided by the package *RegularizedLeastSquares.jl*. The latter implements popular iterative optimization methods, such as the CGNR, FISTA, and ADMM. For all the reconstruction methods implemented in *MRIReco.jl*, the solver can be determined by assigning its name to the parameter `:solver` in the dictionary containing the reconstruction parameters.

2.5.3 | Regularization

To describe regularization functions, *MRIReco.jl* uses the type `Regularization` from the package *RegularizedLeastSquares.jl*. Most notably, this type contains a function to compute the associated proximal map. This approach is very generic in the sense that most common solvers incorporate regularization in the form of a proximal map.

For convenience, *RegularizedLeastSquares.jl* implements several common regularization functions such as TV, ℓ^1, ℓ^2 and low rank regularization. Analogously to the solver to



```
# create image
N = 128
x = shepp_logan(N)
```

```
# create coil sensitivities
smaps = birdcageSensitivity(N,8,3.0)

# simulation parameters
params = Dict{Symbol, Any}()
params[:simulation] = "fast"
params[:trajName] = "SpiralVarDens"
params[:numProfiles] = 1
params[:windings] = 16
params[:numSamplingPerProfile] = 2048
params[:senseMaps] = smaps

# do simulation
acqData = simulation(x, params)
```

```
# reconstruction parameters
params[:reco] = "multiCoil"
params[:reconSize] = (N,N)
params[:regularization] = "TV"
params[:λ] = 2.0e-3
params[:solver] = "admm"
params[:iterationsInner] = 10
params[:iterations] = 30
params[:ρ] = 1.0e-1
params[:absTol] = 1.0e-5
params[:relTol] = 1.0e-4

# do reconstruction
y = reconstruction(acqData, params)
```

FIGURE 4 Example of a high-level simulation and reconstruction script using the software package *MRIReco.jl*. In the lower part of the figure the generating code snippets are shown while the resulting data are shown in the top part. The example starts by generating a 128×128 Shepp–Logan phantom (left column), which is used for simulating 8-fold undersampled *k*-space data (middle column). Finally, a simple *TV*-regularized iterative parallel imaging reconstruction is performed (right column)

be used, these regularization functions can be specified by assigning their respective names to the parameter `:regularization` (eg, `params[:regularization] = "L1"`). Alternatively, one can assign one (or more) `Regularization` objects to aforementioned parameter. This allows the incorporation of custom regularization functions. In this case, the main work to be done is implementing the corresponding proximal map. Finally, the preimplemented regularization objects can serve as building blocks when developing new optimization algorithms.

2.6 | High-level reconstruction

Next we sketch a high-level simulation and reconstruction with *MRIReco.jl*. As outlined before, the interface is designed in such a way that all parameters are passed to the routine `simulation` and `reconstruction` via a parameter dictionary. This approach is very generic and allows specifying a large set of parameters without using long argument lists. The dictionary can also be stored in an XML or TOML file.

The example simulation uses a 128×128 pixel sized Shepp–Logan phantom, 8 birdcage coil sensitivities, and a variable density spiral trajectory with 1 interleave and 2048 samples. This corresponds to an 8-fold undersampling. Afterward, reconstruction is performed using a SENSE-type compressed sensing reconstruction with TV-regularization. The reconstruction problem is then solved using ADMM with 10 inner CG-iterations and 30 outer iterations. The resulting image is shown on the right-hand side of Figure 4.

2.7 | Low-level reconstruction

The high-level reconstruction outlined in the previous section allows performing image reconstruction for a wide range of MRI imaging scenarios. Alternatively, the building blocks in *MRIReco.jl* can be used in a more low-level way to implement custom reconstruction methods. In the next example, we illustrate this using the example of a simple gridding reconstruction. With an `AcquisitionData` object at hand, a gridding reconstruction could be implemented using the following code snippet. First, density compensation weights (`samplingDensity`) are computed and the trajectory to be used is extracted. After forming the gridding operator (`NFFTOp`), the reconstructed image is obtained by applying its adjoint to the weighted *k*-space data. This illustrates that the notation within *MRIReco.jl* is very close to the underlying mathematical description.

```
using MRIReco

# load or simulate acqData

# custom low-level reconstruction
reconSize = acqData.encodingSize[1:2]
weights = samplingDensity(acqData, reconSize)[1]
tr = trajectory(acqData)
F = NFFTOp(reconSize, tr)
kdata = kData(acqData,1,1,1) .* (weights.^2)
reco = adjoint(F) * kdata
```

2.8 | Parallelization

One important way to speed up reconstruction is to make use of parallel code execution. *MRIReco.jl* supports multi-threading (ie, shared memory parallelism) as the primary form of parallelism and has also preliminary support for GPU acceleration, which has been recently added to the package *NFFT.jl* (since version 0.6). To achieve decent speed-ups irrespective of the reconstruction setting, multi-threading is implemented in different parts of *MRIReco.jl*. On a low level, several imaging operators, such as the multi-coil imaging operator and the fieldmap-aware imaging operator, require the computation of multiple NFFTs, which can be done in parallel. On a higher level, one often wants to perform multiple independent reconstructions. This includes the reconstruction of multiple slices of a 2D acquisition or the reconstruction of multiple independent coil images. Therefore, all loops over independent reconstructions are parallelized as well. Parallelization on different levels is possible without risking oversubscription of CPU cores with too many threads since Julia uses a thread pool that allows for nested parallelism similar to OpenMP,[36] Cilk,[37] and Intel TBB.[38]

2.9 | Availability and platform support

MRIReco.jl is developed within a public Git repository hosted at Github (https://github.com/MagneticResonanceImaging/MRIReco.jl). The project contains online documentation that can be accessed from the project homepage. Bug reports, feature requests and comments can be made using an issue tracker. Any commit made to *MRIReco.jl* is tested using continuous integration services. *MRIReco.jl* is supposed to run on any operating system and platform that Julia itself supports. Currently, the test suite runs successfully on Linux, OS X, and Windows.

The software package is licensed under the MIT license (https://opensource.org/licenses/MIT), as are most parts of Julia and its package ecosystem. The MIT license is a permissive license and allows to use the code even in a closed-source application. *MRIReco.jl* has only one GPL dependency (FFTW[39]), which would need to be replaced prior inclusion into a closed source application (Massachusetts Institute of Technology (MIT) and Intel (MKL, Math Kernel Library) provide binary compatible FFTW implementations that can be used in closed-source applications).

2.10 | Experimental evaluation

After giving an overview of the functionality and implementation of *MRIReco.jl*, we next apply *MRIReco.jl* to openly available MRI data and perform a comparison with an existing MRI reconstruction framework.

The first test aims at testing/demonstrating the full reconstruction pipeline from loading MRI data in the commonly used ISMRMRD format to performing image reconstruction. For this purpose, we downloaded a publicly available MRI dataset from the database http://mridata.org.[40] The dataset contains data of a human knee acquired using a 3d FSE sequence and an 8-channel receive coil on a GE scanner. The data were acquired with a FOV of 160 mm $\times$ 160 mm $\times$ 124.8 mm, a matrix size of $320 \times 274 \times 208$ and a TR/TE of 1400 ms/20 ms. The data were measured using variable density Poisson disk sampling with a fully sampled calibration area of size 35×35 and an overall undersampling factor of 7.13. After loading it, the data were converted to 2D data by applying a Fourier transform along the readout direction. Next, coil sensitivy maps were obtained using the ESPIRiT implementation contained in *MRIReco.jl*. Finally, image slices were reconstructed using a SENSE-type reconstruction with ℓ^1-regularization in the Wavelet domain and a regularization parameter of 0.2. The reconstruction problem was solved using the ADMM with 30 iterations and 10 iterations of the inner CG method.

Second, we perfomed a comparison of *MRIReco.jl* with the popular C/C++ reconstruction framework BART.[17] As a model problem, we chose an iterative SENSE reconstruction using the radial dataset published as part of the ISMRM reproducibility challenge 1.[41] It consists of a brain dataset acquired with 12 coils using a radial trajectory with 96 profiles

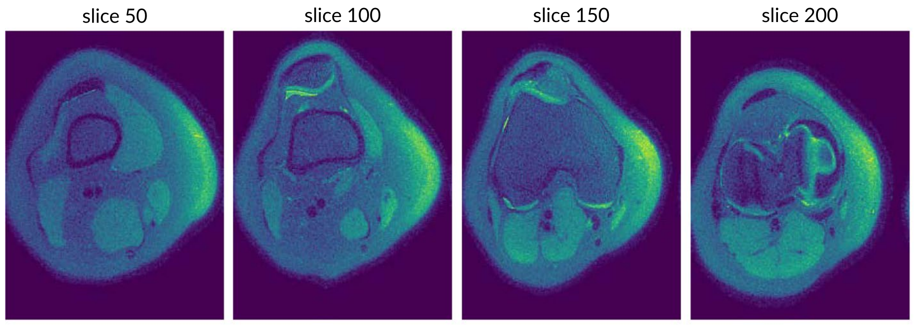


FIGURE 5 Results of the ℓ^1-Wavelet reconstruction of the knee dataset. The reconstructed images are shown for the slices 50, 100, 150, and 200 (along readout direction)

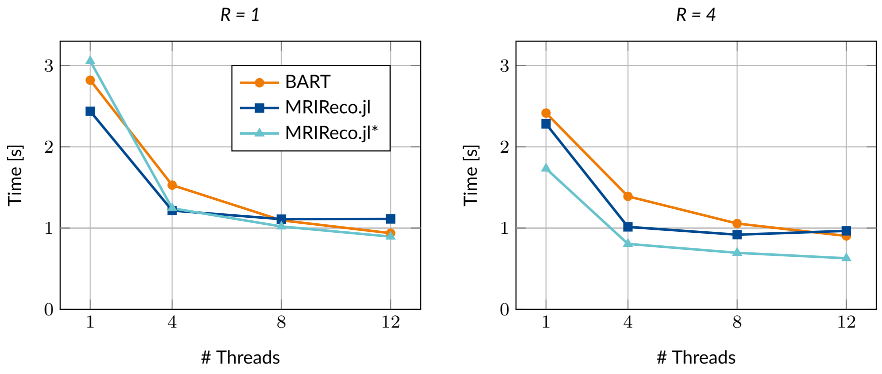


FIGURE 6 Performance comparison between BART and *MRIReco.jl*. Shown are the minimum reconstruction times of an iterative SENSE reconstruction for different numbers of threads ranging from 1 to 12. On the right, the results for reduction factor $R = 4$ are shown while on the left, the results without data reduction can be seen. Orange shows the reconstruction times for BART while dark blue shows the reconstruction times for *MRIReco.jl* both using the Toeplitz optimization. Light blue shows reconstruction time for *MRIReco.jl* without the Toeplitz approach using an oversampling factor $\sigma = 1.25$

and 512 samples per profile. The coil sensitivities are estimated from the data itself using the ESPIRiT implementation that is part of each framework. As proposed in the reproducibility challenge, the fully sampled dataset was retrospectively undersampled by reduction factors between $R = 1$ and $R = 4$. For both frameworks, images were recovered using an iterative ℓ^2-regularized SENSE reconstruction based on the CG method. In all cases, we used 20 iterations and a regularization parameter of 0.01. It is run on a workstation equipped with 2 AMD EPYC 7702 CPUs running at 2.0 GHz (256 cores in total) and a main memory of 1024 GB.

The data/software that support the findings of this study are openly available in *MRIReco.jl* at 10.5281/zenodo.4464857, SHA-1 hash 72fbbd0, and *MRIRecoBenchmarks* at https://doi.org/10.5281/zenodo.4467979, SHA-1 hash b24b60c. To be precise, the reconstruction of the knee dataset is contained in the examples folder in *MRIReco.jl*, while the code for the comparison with BART is contained in *MRIRecoBenchmarks*.

3 | RESULTS

3.1 | Reconstruction results of the knee dataset

Figure 5 shows images of the reconstructed knee dataset for some exemplary slices. These results illustrate a typical application for researchers, who wish to test their reconstruction methods not only using simulation data but also data from MRI scanners. As illustrated in the example code accessible in the Git repository, this can by achieved quite easily with *MRIReco.jl* in conjunction with the ISMRMRD format. Thus, all that is needed to work with raw data is to convert the latter from its vendor specific format into the ISMRMRD format. Afterward, the data can be reconstructed using either one of the preimplemented reconstruction methods or by a custom reconstruction method making use of the low-level building blocks provided by *MRIReco.jl*.

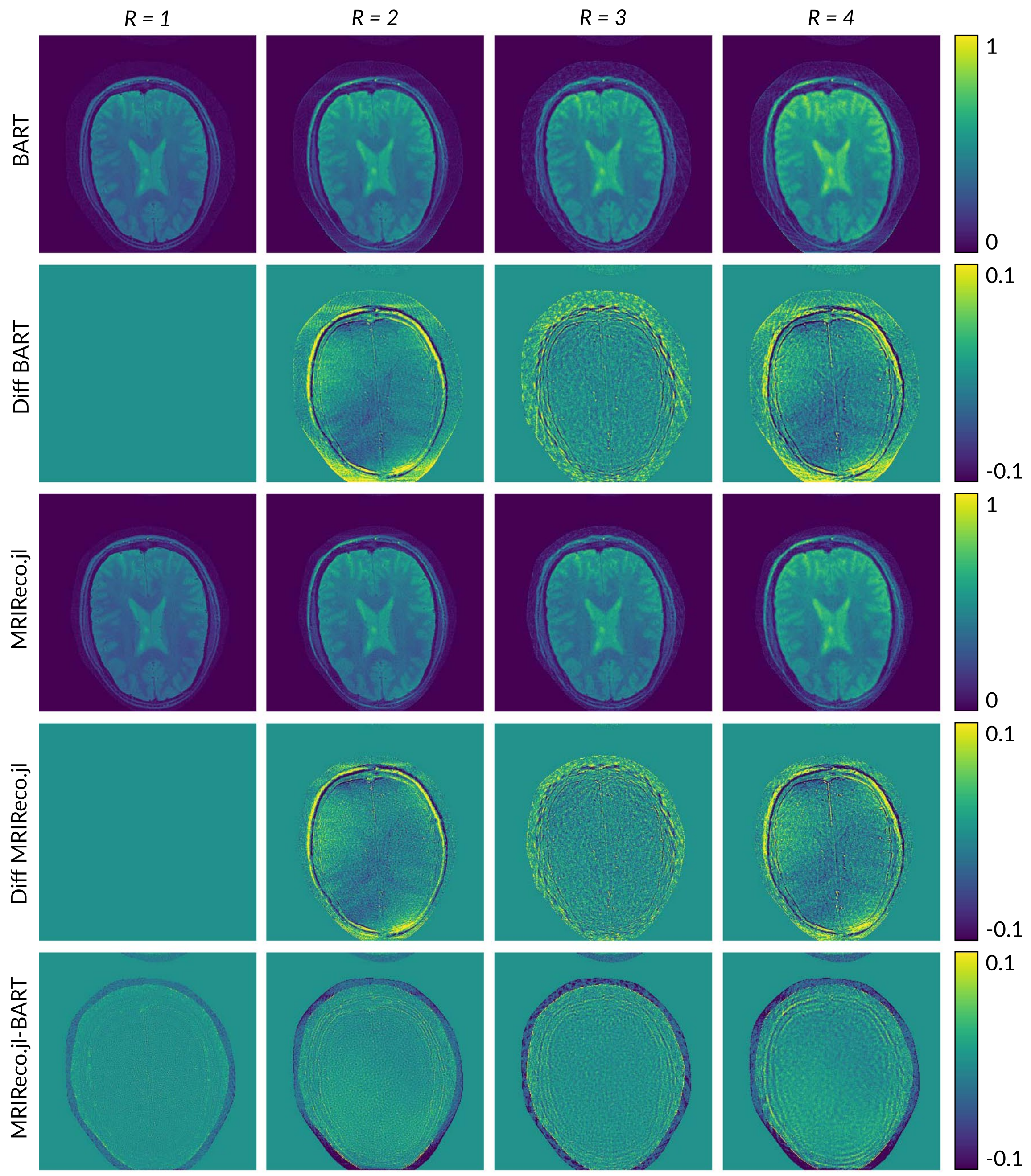


FIGURE 7 Comparison of SENSE reconstructions of a public brain imaging dataset using BART and *MRIReco.jl*. The reconstruction results are shown in rows 1 and 3, while rows 2 and 4 show difference maps to the reference reconstruction ($R = 1$). Row 5 shows difference maps between the images reconstructed using both frameworks for the same reduction factor

3.2 | Runtime performance

Designing a proper performance benchmark for independent software frameworks can be a challenging task since frameworks often differ not only in the implementation but also in the choice of implemented algorithms/optimizations and in the choice of default reconstruction parameters. Since both BART and *MRIReco.jl* implement multi-threading we performed benchmarks for different numbers of threads (1, 4, 8, and 12). For each framework, the reconstruction is performed multiple times and the minimum time is used for comparison. In this way, both frameworks are benchmarked under idealized but comparable conditions. In particular, this procedure considers hot CPU caches.

BART by default uses the Toeplitz optimization for efficient multiplication with the normal matrix. During the development of the benchmark, we implemented this feature as well to match the implementation in BART. In addition to the Toeplitz optimization, which implies using FFTs with an oversampling of factor $\sigma = 2.0$, *MRIReco.jl* also allows

TABLE 1 Number of code lines needed to perform the central tasks associated with the reconstruction of the brain dataset

Task	BART	MRIReco.jl
Data loading & conversion	5	7
Gridding & ESPIRiT	4	2
Undersampling data	3	1
Reconstruction parameters	0	10
Reconstruction	1	1
Sum	13	21

running the code without the Toeplitz optimization but with a smaller oversampling factor such as $\sigma = 1.25$ (see Ref. [42] for investigation of oversampling factor sizes). Although this may slightly reduce the accuracy of the NFFT approximation, we observe that in practice the approximation error is so small that it is not visually perceptible in the reconstructed images.

The results of the performance comparison are summarized in Figure 6 for the reduction factors $R = 1$ and $R = 4$. One can see that both reconstruction frameworks achieve very similar reconstruction times. *MRIReco.jl* is slightly faster for 1 and 4 threads while BART is faster for 12 threads. When comparing the result with and without Toeplitz optimization one can see that for $R = 1$ both approaches achieve similar performance while for $R = 4$ the non-Toeplitz reconstruction with oversampling factor $\sigma = 1.25$ is clearly faster. This can be explained by the smaller size of the FFT and the fact that for $R = 4$ significantly fewer points need to be gridded compared to the case where $R = 1$.

3.3 | Reconstruction accuracy

Reconstruction results are summarized in Figure 7. They are shown for reduction factors $R = 1, 2, 3$, and 4 using the same reconstruction parameters. For *MRIReco.jl* the images obtained using the Toeplitz optimization are shown. For both frameworks, only a minor decrease in image quality can be observed with increasing reduction factor. When looking at difference maps between reduction factor $R = 1$ and higher reduction factors (rows 2 and 4) one observes that $R = 3$ shows a smaller deviation in outer image regions of the head than reduction factors $R = 2$ and $R = 4$.

When comparing the reconstruction results of the 2 frameworks one can hardly see a difference. Only the outer regions beyond the head look slightly different, which is likely caused by differences in the coil estimation algorithm. Even the difference maps compared to the $R = 1$ reconstruction look very similar, which indicates that both frameworks implement the iterative SENSE reconstruction in a similar way. This is further supported by the difference maps between the 2 reconstruction frameworks shown in the fifth row. These show mostly noise in the head region.

3.4 | Syntactic comparison

In order to provide a rough estimate for the complexity of the user interface, Table 1 summarizes the number of code lines required to perform the central tasks associated with the reconstruction of the brain dataset. This count excludes code lines specific to the benchmarking, such as setting up the benchmark or running multiple trials of the reconstruction. The results show that both frameworks require a similar number of code lines to perform reconstruction. The main difference arises from the fact that the MRIReco.jl-implementation uses a dictionary to store reconstruction parameters, which is not required for BART. We note, however, that passing the parameters as a dictionary is done solely for the purpose of keeping the code readable. Alternatively, the parameters could be passed directly as keyword arguments as done by BART. We conclude that both frameworks perform image reconstruction with a comparably complex interface and we acknowledge that the complexity of the final code depends on the preferences of the user at hand.

4 | DISCUSSION

MRIReco.jl started as an experiment of the authors to check the suitability of Julia for developing an MRI reconstruction framework. Initial success was measured in a rapid development experience while still generating programs that can compete in terms of runtime speed with equivalent C/C++ programs. From this point, the framework was developed to make it not only usable for the authors themselves but also for other users. We characterize the status of the project as usable but not yet finished. This implies that interfaces of upcoming versions might slightly change and that parts of the documentation are still incomplete.

In a performance benchmark, it was shown that *MRIReco.jl* achieves similar and sometimes even better performance than the state-of-the-art C/C++ MRI reconstruction framework BART. We note that the benchmark was performed for a very specific reconstruction algorithm (iterative SENSE) and its results are not directly applicable to other aspects of each of the frameworks.

One key philosophy of *MRIReco.jl* is to reuse existing code and keep the code base small. This makes the package more maintainable as it avoids code duplication and keeps the responsibility for code in the original software repositories. Despite its advantages, this approach requires a package management system capable of handling complex version dependencies. Julia readily addresses this aspect with its integrated package manager. An important responsibility in a package with many dependencies is to maintain compatibility with the depending packages. This can increase the complexity if the packages are not managed by oneself. Fortunately, the Julia dependency system allows to pin packages to certain versions such that breaking API changes in depending packages can be avoided.

One weak point of *MRIReco.jl* is that no functionality for deployment on an MRI scanner is implemented yet. There are basically 3 different implementations thinkable. The first is to write a server in Julia and communicate with the host server via TCP/IP, as was done in.[43] The second possibility is to embed *MRIReco.jl* into an existing C/C++ program. In this way, it would be possible to integrate *MRIReco.jl* into for instance Gadgetron and implement a *JuliaGadget* similar to the existing `PythonGadget`. Since Julia arrays have a C-compatible binary format, it is possible to pass data from C to Julia by only passing the pointer to the data. Finally, deployment can be achieved using automated offline reconstruction techniques such as Yarra[44] and Autorec.[45]

Julia itself as a language has evolved since its introduction in 2012 into a stable, featureful language that can compete with more widely developed languages such as Python and C/C++. One of the remaining issues of Julia compared to other programming languages is its latency. Code compilation in Julia is currently done right before code execution and thus can add a certain amount of latency. The latency of packages is reduced to some extent by so-called pre-compilation but the current Julia version (1.5.3) stores only an intermediate representation and not the native machine code. The remaining latency issue can be mitigated by the package *Revise.jl*, which allows to cache compiled code during a julia session. An alternative for deployment usage is the package *PackageCompiler.jl*, which allows to either compile packages into the system image of Julia or to generate standalone executables.

5 | CONCLUSIONS

In conclusion, we have introduced a new image reconstruction framework for MRI, which is both performant and accessible. Two key aspects of *MRIReco.jl* are its modularity and its open interface. These make it a useful tool not only for performing image reconstruction but also for the development and testing of new reconstruction methods. The framework is implemented purely in the programming language Julia and reaches similar performance and similar image quality as other popular image reconstruction frameworks that are implemented in a low-level programming language.

ACKNOWLEDGEMENTS

We acknowledge many fruitful discussions with Martin Uecker on the usage of BART and the design of the benchmarks.

ORCID

Tobias Knopp http://orcid.org/0000-0002-1589-8517

REFERENCES

1. Lustig M, Donoho D, Pauly JM. Sparse MRI: the application of compressed sensing for rapid MR imaging. *Magn Reson Med*. 2007;58:1182-1195.
2. Lustig M, Pauly JM. SPIRiT: iterative self-consistent parallel imaging reconstruction from arbitrary k-space. *Magn Reson Med*. 2010;64:457-471.
3. Jin KH, Lee D, Ye JC. A general framework for compressed sensing and parallel MRI using annihilating filter based low-rank Hankel matrix. *IEEE Trans Comput Imaging*. 2016;2:480-495.
4. Haldar JP. Low-rank modeling of local k-space neighborhoods (LORAKS) for constrained MRI. *IEEE Trans Med Imaging*. 2013;33:668-681.
5. Shin PJ, Larson PE, Ohliger MA, Elad M, Pauly JM, Vigneron DB, et al. Calibrationless parallel imaging reconstruction based on structured low-rank matrix completion. *Magn Reson Med*. 2014;72:959-970.
6. Doneva M, Börnert P, Eggers H, Stehning C, Sénégas J, Mertins A. Compressed sensing reconstruction for magnetic resonance parameter mapping. *Magn Reson Med*. 2010;64:1114-1120.
7. Zhang T, Pauly JM, Levesque IR. Accelerating parameter mapping with a locally low rank constraint. *Magn Reson Med*. 2015;73:655-661.
8. Schweser F, Sommer K, Deistung A, Reichenbach JR. Quantitative susceptibility mapping for investigating subtle susceptibility variations in the human brain. *Neuroimage*. 2012;62:2083-2100.
9. Mani M, Jacob M, Kelley D, Magnotta V. Multi-shot sensitivity-encoded diffusion data recovery using structured low-rank matrix completion (MUSSELS). *Magn Reson Med*. 2017;78:494-507.
10. Stone SS, Haldar JP, Tsao SC, Sutton B, Liang ZP, et al. Accelerating advanced MRI reconstructions on GPUs. *J Parallel Distributed Comput*. 2008;68:1307-1318.
11. Sorensen TS, Atkinson D, Schaeffter T, Hansen MS. Real-time reconstruction of sensitivity encoded radial magnetic resonance imaging using a graphics processing unit. *IEEE Trans Med Imaging*. 2009;28:1974-1985.
12. Van Der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. *Comput Sci & Eng*. 2011;13:22.
13. Abadi M, Barham P, Chen J et al. Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({ OSDI } 16). 2016:265-283.
14. Paszke A, Gross S, Massa F, et al. Pytorch: an imperative style, high-performance deep learning library. *Adv Neural Inf Process Syst*. 2019;8026-8037
15. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. *SIAM Rev*. 2017;59:65-98.
16. Hansen MS, Sørensen TS. Gadgetron: an open source framework for medical image reconstruction. *Magn Reson Med*. 2013;69:1768-1776.
17. Uecker M, Ong F, Tamir JI, et al. Berkeley advanced reconstruction toolbox. *Proc Int Soc Mag Reson Med*. 2015;23:2486.
18. Ong F, Lustig M. SigPy: a python package for high performance iterative reconstruction. *Proc Int Soc Mag Reson Med*. 2019;27:4819.

19. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. IEEE Computer Society; 2004:75
20. Milner R. A theory of type polymorphism in programming. *J Comput System Sci*. 1978;17:348-375.
21. Lam SK, Pitrou A, Numba SS. Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. ACM; 2015:7.
22. Bolz CF, Cuni A, Fijalkowski M, Rigo A. Tracing the meta-level: PyPy's tracing JIT compiler. In: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. ACM; 2009:18-25.
23. Fessler JA, Nol DC. Model-based MR image reconstruction with compensation for through-plane field inhomogeneity. In: *2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE; 2007:920-923.
24. Eggers H, Knopp T, Potts D. Field inhomogeneity correction based on gridding reconstruction for magnetic resonance imaging. *IEEE Trans Med Imaging*. 2007;26:374-384.
25. Knopp T, Eggers H, Dahnke H, Prestin J, Sénégas J. Iterative off-resonance and signal decay correction for improved multi-echo imaging in MRI. *IEEE Trans Med Imaging*. 2009;28:394-404.
26. Pruessmann KP, Weiger M, Scheidegger MB, Boesiger P. SENSE: sensitivity encoding for fast MRI. *Magn Reson Med*. 1999;42:952-962.
27. Uecker M, Lai P, Murphy MJ, Virtue P, Elad M, Pauly JM, et al. ESPIRiT–an eigenvalue approach to autocalibrating parallel MRI: where SENSE meets GRAPPA. *Magn Reson Med*. 2014;71:990-1001.
28. Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. *J Res Nat Bureau Standards*. 1952;49.
29. Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J Imaging Sci*. 2009;2:183-202.
30. Parikh N, Boyd S, et al. Proximal algorithms. *Foundations Trends® Optim*. 2014;1:127-239.
31. Goldstein T, Osher S. The split Bregman method for L1-regularized problems. *SIAM J Imaging Sci*. 2009;2:323-343.
32. Pruessmann KP, Weiger M, Börnert P, Boesiger P. Advances in sensitivity encoding with arbitrary k-space trajectories. *Magn Reson Med*. 2001;46:638-651.
33. Knopp T, Kunis S, Potts D. A note on the iterative MRI reconstruction from nonuniform k-space data. *Int J Biomed Imaging*. 2007;2007.
34. Pipe JG, Menon P. Sampling density compensation in MRI: rationale and an iterative numerical solution. *Magn Reson Med*. 1999;41:179-186.
35. Inati SJ, Naegele JD, Zwart NR, Roopchansingh V, Lizak MJ, Hansen DC, et al. ISMRM Raw data format: a proposed standard for MRI raw datasets. *Magn Reson Med*. 2017;77:411-421.
36. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput Sci Eng*. 1998;5:46-55.
37. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multithreaded runtime system. *J Parallel Distributed Comput*. 1996;37:55-69.
38. Kukanov A, Voss MJ. The foundations for scalable multi-core software in intel threading building blocks. *Inte Technol J*. 2007;11.
39. Frigo M, Johnson SGFFTW. FFTW: An adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181), IEEE; 1998;3:1381-1384.
40. Ong F, Amin S, Vasanawala S, Lustig M. Mridata.org: an open archive for sharing MRI raw data. *Proc Int Soc Mag Reson Med*. 2018;26:1.
41. Maier O, Baete SH, Fyrdahl A, Hammernik K, Harrevelt S, Kasper L, et al. CG-SENSE revisited: results from the first ISMRM reproducibility challenge. *Magn Reson Med*. 2020.
42. Eggers H, Boernert P, Boesiger P. Comparison of gridding- and convolution-based iterative reconstruction algorithms for sensitivity-encoded non-Cartesian acquisitions. In: Proceedings of the 10th Annual Meeting of ISMRM, Honolulu; 2002:743.
43. Gräser M, Thieben F, Szwargulski P, Werner F, Gdaniec N, Boberg M, et al. Human-sized magnetic particle imaging for brain applications. *Nat Commun*. 2019;10:1-9.
44. Block KT, Wiggins R. *Yarra—A Toolbox for Clinical MRI Research*. Accessed: 2020-01-20. http://http://yarraframework.com
45. Borisch EA, Grimm RC, Riederer SJ. Automated reconstruction processing. *Proc Intl Soc Mag Reson Med*. 2019;27:1.

How to cite this article: Knopp T, Grosser M. MRIReco.jl: An MRI reconstruction framework written in Julia. *Magn Reson Med*. 2021;86:1633–1646. https://doi.org/10.1002/mrm.28792