**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.

**Conclusion:** 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.
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, 4, 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, 7, 8, 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 GPGPU (General Purpose Graphical Purpose Units) [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 two conflicting goals. On the one hand, one seeks a programming environment that allows 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. However, this is often accompanied by suboptimal computational performance. On the other hand, computation speed can be ensured by working in a low-level (LL) programming environment, albeit at the cost of 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 runs in a sufficiently short reconstruction time. In that case, the increase in system complexity is thus addressed by increasing the development resources.

A popular approach to reconcile both goals is to use a mix of a HL and a LL programming environment. In this hybrid approach, only hot loops that take most of the time are implemented in the LL language, while the remaining part is implemented in the HL programming environment. Popular examples are Matlab with its Mex interface, i.e. a combination of Matlab and C/C++ and Python with its different ways to call C code (e.g. 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 many advantages but it also has issues, which the authors in [15] summarize in the two language problem:

- For someone programming in the HL language it is often unclear what the LL implementation looks like. Thus,
the programming interface is often very high level, which 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. This leads to the situation that 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. For researchers this has the consequence that using an existing framework is easy. On the other hand 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. At the same time the framework is implemented in a single programming language, thus avoiding the two-language problem. To achieve this, our framework is written in 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 two-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 a set of 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. For instance transformations like the FFT as well as iterative solvers like the conjugated gradients method (CG) are independent of MRI and used in other domains. 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 stack and develop extensions or even make modifications to the source code easily. We note that hackability only slightly overlaps with prototyping mentioned in [16]. Hackability in particular 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 two 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 two of the additional design goals can be met. For instance, we note that 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, even for different imaging modalities, often share a large number of generic building blocks. These shared 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. Moreover, 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 all building blocks. Finally, we note that the reuse of standard libraries 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 discussed in Sec. 1. 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, it is fine to mix this with a dynamic programming style 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, it is thus possible to cross language boundaries even for scalar functions.

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] uses an approach where the user needs to manually annotate the code to be compiled and only a subset of the language can be used. Here, again, the key for reaching high performance is the ability of the system to determine the types of all variables in a function body. A more general JIT compiler for Python that does not need 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 mainly on basic building blocks that have proven to be useful for MRI image reconstruction. In particular MRIReco.jl currently offers:

- **Simulation**: Methods for simulating MRI data based on software phantoms. This includes support for taking into account $B_0$ 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 as well as a faster method that applies 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 ESPRIT [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 $\ell^1$, $\ell^2$ 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 [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. On purpose we have not introduced a further abstraction mechanism like a pipeline architecture [16]. Such patterns are useful to turn statically compiled programs into dynamic ones by introducing a new pipeline language. They are, however, not needed in dynamic programming languages like Matlab, Python, and Julia. In fact, pipelining can be written in Julia as simple function composition.

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
In order to efficiently work with MRI data, MRIReco.jl introduces two distinct datatypes for representing the latter. `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.
| unprocessed data | preprocessed data |
|------------------|------------------|
| mutable struct RawAcquisitionData | mutable struct AcquisitionData |
|   params::Dict{String,Any} |   sequenceInfo::Dict{Symbol,Any} |
|   profiles::Vector{Profile} |   traj::Vector{Trajectory} |
| end | kdata::Array{Matrix{ComplexF64}},3} |
| mutable struct Profile | subsampleIndices::Vector{Int64} |
|   head::AcquisitionHeader | encodingSize::Vector{Int64} |
|   traj::Matrix{Float32,2} | fov::Vector{Float64} |
|   data::Matrix{ComplexF32,2} | 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 Fig. 2. The syntax profiles::Vector{Profile} means that the field profiles is of 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 but it can also be extended by custom fields. In Julia the appropriate data structure for storing key-value pairs is a Dict, i.e. an associative array. In the parameter params all data that are present in the XML header of an ISMRMRD file are stored. 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

```xml
<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 that is convenient for reconstruction. The definition of AcquisitionData can be found on the right hand side of Fig. 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 three dimensions encoding

1. contrasts/echoes
2. slices
3. repetitions

Each element is a matrix whose dimensions encode

1. k-space nodes
2. channels/coils

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 is to 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 Fig. 2. The most important field is `nodes`, which contains the sampling points as a matrix where the first dimension encodes the dimensionality of the $k$-space and the second dimension encodes the number of sampling points. This general structure allows implementing arbitrary sampling trajectories. Moreover, `MRIReco.jl` provides convenient constructors for the most common types of trajectories. For instance `MRIReco.jl` has support for 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 Fig. 3.

### 2.3.4 | Image Data

`MRIReco.jl` returns image reconstruction results in the form of an `AxisArray`, which is 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 ISMRMRS 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

![FIGURE 3 Exemplary trajectories available in MRIReco.jl.](image)
code example shows how to convert a Bruker MRI file into an ISMRMRD file:

```python
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 a simulation. It is thus possible to cache simulated data in a file, 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 \( m \in \mathbb{C}^N \) from a given set of measurements \( s \in \mathbb{C}^M \) and a signal encoding model described by a linear operator \( H \in \mathbb{C}^{M \times N} \). Thus, one seeks a solution to an inverse problem of the form

\[
\arg\min_m \| s - H m \|_2^2 + R(m),
\]  

(1)

where \( R \) 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 describing the action of the signal encoding operator \( H \) and its adjoint \( H^H \). Moreover, they describe transformations applied to \( m \) within the regularization term \( R(m) \) (e.g. sparsifying transforms used in CS).
- **Solvers**: An optimization algorithm used to solve problem (1).
- **Proximal Maps**: These are associated with the given regularization functions.

The interface in `MRRRec`.jl is designed in such a way 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 then 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 \( H \) is a discrete approximation of the underlying signal model

\[
s_p(t) = \int_{\mathbb{R}^d} c_p(r) m(r) e^{-z(r) t} e^{-2\pi i k(t) \cdot r} \, dr.
\]  

(2)

Here \( s_p(t) \) denotes the demodulated signal received in the \( p \)-th of \( P \) receive coils at time \( t \). Moreover, \( m(r) \) is the transverse magnetization of the object at position \( r \) and \( c_p(r) \) are the receive coil sensitivities. Finally, the term \( z(r) \in \mathbb{C} \) contains the \( R_z^+ \) and \( B_0 \) maps in its respective real and imaginary parts. As a second application, linear operators can
be applied to $m$ 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 an operator is characterized solely by its action when applied to a vector. This allows to evaluate an operator 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 $\star$-operator or form its adjoint using the postfix $'$. Similarly, linear operators can be composed using either of the operators $\star$ and $\circ$, and their size can be determined using the function $\text{size}$.\(^1\)

`MRIReco.jl` provides implementations of the operators that are most 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 multiplies an image by a set of coil-sensitivities as used in SENSE-type image reconstructions.
- **SamplingOp**: An operator for (sub)sampling ($k$-space) data. The operator is used for compressed-sensing reconstruction.
- **WeightingOp**: A weighting operator used for tasks such as sampling density compensation.

Additional operators are reexported from the Julia package `SparsityOperators.jl`. These include common sparsifying transforms such as the Wavelet transform and finite differences operators.

In addition to these 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 $N = H^H H$ of the encoding operator. Thus, algorithms can sometimes be accelerated by optimizing the normal operator instead of optimizing the encoding operator itself. A relevant example is the case when $H$ is an NFFT. In that case the corresponding normal operator has a Toeplitz structure. Thus, multiplication with the normal operator can be carried out using two FFTs with an oversampling factor of 2 [36]. To allow for this kind of optimization, `MRIReco.jl` reexports the type `normalOperator`, which is implemented in the packages `SparsityOperators.jl` and `RegularizedLeastSquares.jl`. Its default implementation performs a sequential multiplication with the underlying operator followed by a multiplication with the adjoint. 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 given 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

---

\(^1\) We note that the term operator is used here for both the binary mathematical operation and the linear mapping.
ADMM. For all of 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 of the regularization function. This approach is very generic in the sense that most common solvers incorporate regularization in the form of such a proximal map.

For convenience, RegularizedLeastSquares.jl implements several common regularization functions such as TV, $\ell^1$, $\ell^2$ and low rank regularization. Analogously to the solver to be used, these regularization functions can be specified by assigning their respective names to the parameter :regularization (e.g. params[:regularization] = "L1"). Alternatively, one can also 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 the implementation of 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 having to use long argument lists. The dictionary can also be stored in an XML or TOML file.

The example simulation uses a 128 × 128 pixelsized 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. After simulating $k$-space data, reconstruction is performed using a SENSE-type compressed sensing reconstruction. Since the Shepp-Logan phantom is piece-wise constant, edge-preserving TV-regularization is used to mitigate undersampling artifacts. In the example code, the latter is specified by the assignment params[:regularization] = "TV". 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 Fig. 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. This is achieved by a dispatch to different individual reconstruction methods, such as direct reconstruction, iterative single-channel reconstruction, iterative multi-channel reconstruction or iterative multi-echo reconstruction. 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. We assume that a dataset has been either loaded or simulated. With the AcquisitionData at hand, a gridding reconstruction could then be implemented using the following code snippet. First, the density compensation weights are determined using the method samplingDensity. Next the sampling trajectory is extracted from the acqData and the gridding operator is formed by calling the function NFFTOp. Afterwards, the $k$-space data is extracted from acqData and weighted with the obtained density compensation weights. Finally, the reconstructed
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 is shown in the top part. The example starts by generating a $128 \times 128$ Shepp–Logan phantom (left column), which is used for simulating 8-fold undersampled $k$-space data (middle column). Finally, a simple $\ell_1$-regularized iterative parallel imaging reconstruction is performed (right column).
image is obtained by applying the adjoint of the imaging operator to the weighted $k$-space data. This illustrates that the notation within MRIReco.jl is very close to what one expects from the 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,a,1,1,1).* (weights.^2)
reco = adjoint(F) * kdata
```

2.8 Parallelization

One important way to speed up reconstruction is to use some form of parallel code execution. Nowadays most computing hardware provides multiple cores that can operate in parallel. There are basically three types of parallel computing: shared memory parallelism, distributed memory parallelism and the usage of specialized computing hardware that has massively parallelized circuits (usually a GPU). Julia allows for implementing all three forms of parallel computing.

MRIReco.jl supports the use of multi-threading (i.e. shared memory parallelism) to speed up image reconstruction. Multi-threading is a very popular form of parallelism since it does not require additional hardware and can be implemented without majorly rewriting an existing codebase. In particular it does not require to copy memory between processes, or systems, which is required both in distributed memory systems and GPU computing. Julia’s multi-threading capabilities are based on a prototype developed in [37] and first published as an experimental feature in Julia v0.6. Unlike in Python, the threads run truly in parallel and thus it is not required to write C/C++ to use concurrency. While we focus on multi-threading in this work, we note that with version 0.6 the library NFFT.jl gained GPU support. The latter will be integrated into MRIReco.jl in the future, such that it will be possible to perform image reconstruction on the GPU.

Multi-threading usually complicates the implementation, and also needs to be done with care to get decent speedups. One major challenge arises when different components within a software stack perform independent parallelization. For instance, low-level libraries such as BLAS and FFTW have dedicated thread pools that are not aware of a parallelism in the high-level code calling BLAS and FFTW. These issues can be solved by using centralized thread pool implementations like offered for C applications by OpenMP [38], Cilk [39], and Intel TBB [40]. In version 1.3, Julia introduced a similar system called partr (parallel tasks runtime) that allows for nested parallelism.

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. By parallelizing the corresponding loops, multiple NFFTs can be computed simultaneously resulting in significant speed-ups. 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 coil images, when no parallel imaging is applied. As a consequence, all loops over independent reconstructions are parallelized as well. To avoid interference with the Julia thread pool, the thread pools of BLAS and FFTW are disabled when loading MRIReco.jl. For FFTW this is not strictly necessary since this C library has already been adapted to be partr-aware.

For the implementation of parallel loops we use the @spawn macro and rewrite serial loops of the form

```julia
for n=1:N
    # do calculations
end
```

into the form

```julia
@sync for n=1:N
    Threads.@spawn begin
        # do calculations
    end
end
```

In case of data dependencies a mutex is used to lock access to the critical section. Tasks in Julia do not directly map to threads and thus it is usually fine to spawn more tasks than threads are available in the thread pool. Note, however, that task creation of course also has a small overhead so that this form of loop parallelization should only be done when the body of the for loop contains a sufficient number of operations. For instance the loop over the coils in a SENSE reconstruction needs to perform a Fourier transform in the function body, which is a sufficient workload and thus the overhead of task creation can be neglected.

### 2.9 Availability and Platform Support

MRIReco.jl is developed within a public Git repository hosted at Github.² 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 license³, 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 [41]), which would need to be replaced prior inclusion into a closed source application⁴.

²[https://github.com/MagneticResonanceImaging/MRIReco.jl](https://github.com/MagneticResonanceImaging/MRIReco.jl)
³[https://opensource.org/licenses/MIT](https://opensource.org/licenses/MIT)
⁴Massachusetts 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 the final image reconstruction. For this purpose, we downloaded a publicly available MRI dataset from the database http://mridata.org [42]. 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 acquisition was performed using a field of view of 160 mm x 160 mm x 124.8 mm and a matrix size of 320 x 274 x 208. The repetition time and echo time are 1400 ms and 20 ms, respectively. The data was measured using variable density Poisson disk sampling with a fully sampled calibration area of size 35x35 and an overall undersampling factor of 7.13. After loading the data, the data was converted to 2d data by applying a Fourier transform along the readout direction. Next, coil sensitivity maps were obtained using the ESPIRiT implementation contained in MRIReco.jl. Finally, image slices were reconstructed using a SENSE-type reconstruction with $\ell_1$-regularization in the Wavelet domain. For the reconstruction we employed a regularization parameter of 0.2. The corresponding reconstruction problem was solved using the ADMM with 30 iterations and 10 iterations of the inner CG method.

Secondly, we performed 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 [43]. It consists of a brain dataset acquired with 12 coils using a radial trajectory with 96 profiles 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 was 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 $\ell_2$ regularized SENSE reconstruction based on the conjugated gradient method. In all cases we used 20 iterations and a regularization parameter of 0.01. It is run on a workstation equipped with two 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 72fbbd0485fded314df75b30a61b42ab9606ca188, and MRIRecoBenchmarks at 10.5281/zenodo.4467979, SHA-1 hash b24bb60ca512589ec6284b31d8e7c07628c2c5886. 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 Data Set

Fig. 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 one or more MRI scanners. As is illustrated in the example code accessible in the Git repository, this can be achieved quite easily by using 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. Afterwards, the data can be reconstructed using either one of the preimplemented reconstruction methods of MRIReco.jl or by a custom reconstruction method making use of the low-level building blocks provided by the package.
FIGURE 5  Results of the $\ell^1$-Wavelet reconstruction of the knee data set. The reconstructed images are shown for the slices 50, 100, 150 and 200 (along readout direction).

3.2  |  Runtime Performance

Designing a proper performance benchmark for independent software frameworks can be a challenging task since each of the frameworks often differs 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 (1,4,8,12). 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 taking the minimum time of several runs 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 running the code without the Toeplitz optimization but with a smaller oversampling factor such as $\sigma = 1.25$ (see [44] 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 Fig. 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 code paths achieve similar performance while for $R = 4$ the non-Toeplitz reconstruction with an oversampling factor of $\sigma = 1.25$ 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 Fig. 7. Shown are the results for reduction factors $R = 1, 2, 3, 4$ using the same reconstruction parameters. For MRIReco.jl the images obtained using the Toeplitz optimization are shown. For both frameworks only a minor decrease of image quality can be observed with increasing reduction factor. When looking at difference maps between reduction factor $R = 1$ and higher reduction factors (row two and four) one can see that $R = 3$ shows a smaller deviation in outer image regions of the head than reduction factors $R = 2$ and $R = 4$. 
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 $\sigma=1.25$.

When comparing the reconstruction results of the two 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. However, these were not investigated in detail in this paper. 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 two reconstruction frameworks shown in the fifth row. These show mostly noise in the head region and one cannot identify the structure of the underlying image. Hence, there appears to be no bias between the different reconstructions.

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 allow to perform image reconstruction with a comparably complex interface and we acknowledge that the complexity of the final code depends a lot on the preferences of the user at hand.
FIGURE 7  Comparison of SENSE reconstructions of a public brain imaging dataset using BART and MRIReco.jl. The reconstruction results are shown in rows one and three, while rows two and four show difference maps to the reference reconstruction ($R = 1$). Row five shows difference maps between the images reconstructed using both frameworks for the same reduction factor.
| task                                | BART | MRIReco.jl |
|------------------------------------|------|------------|
| data loading & conversion          | 5    | 7          |
| gridding & ESPRIT                  | 4    | 2          |
| undersampling data                 | 3    | 1          |
| reconstruction parameters          | 0    | 10         |
| reconstruction                     | 1    | 1          |
| sum                                | 13   | 21         |

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

## 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 very 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 into a state making 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 in some cases 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 therefore 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 software repository where it belongs. This approach has many advantages, but it also increases the complexity in version management. Therefore, this approach requires a package management system that is capable of handling complex version dependencies. Julia addresses this aspect with the package manager, which is an integral part of the programming environment. One important new responsibility in a package with many dependencies is that it requires the authors to follow the developments of the depending packages. This is simple if the dependencies are managed by oneself, but it can increase the complexity if not. Fortunately, the Julia dependency system allows to pin packages to certain versions such that breaking API changes in depending packages can be avoided.

Compared to existing packages *MRIReco.jl* has more in common with SigPy [18] than with Gadgetron or BART. SigPy allows to easily combine linear operators, optimization algorithms, and proximal mappings and keeps this orthogonal functionality in separate code units. Since *MRIReco.jl* was developed independently of SigPy both packages have not yet influenced each other. Both projects are licensed under a permissive license and it is therefore possible to reuse components, which may benefit both packages. We emphasise that it is simple to convert between Python and Julia code since both programming languages use similar notation.

One of the weak points of *MRIReco.jl* is that no functionality for deployment on an MRI scanner is implemented so far. There are basically three different implementations thinkable. The first is to write a server in Julia and communicate with the host server via TCP/IP. A similar architecture has been implemented by the authors in [45] for the tomographic imaging modality magnetic particle imaging where data is measured in real-time and directly processed.
The TCP-based data acquisition server can be found at https://github.com/tknopp/RedPitayaDAQServer. The second possibility is to embed MRIReco.jl into an existing C/C++ program. This can be done, since Julia has an embedding ABI/API that can be called from a variety of programming languages, in particular from C/C++. 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. Similarly it would also be possible to integrate MRIReco.jl with BART. Finally, deployment can be achieved using automated offline reconstruction techniques such as Yarra [46] and Autorec [47].

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 extend by so-called pre-compilation but currently (Julia version 1.5.3) only an intermediate representation and not the native machine code is stored. 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 magnetic resonance imaging data, which is both performant and accessible. The framework is implemented purely in the programming language Julia. Our tests have shown that the latter is very suitable for implementing such a framework. 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. In a comparison with the MRI reconstruction framework BART we have shown that MRIReco.jl reaches similar performance and yields similar image quality, which proves that Julia can reach a similar performance as comparable C/C++ implementations not only in micro benchmarks but also in real-world applications.

### references

[1] Lustig M, Donoho D, Pauly JM. Sparse MRI: The application of compressed sensing for rapid MR imaging. Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 2007;58(6):1182–1195.

[2] Lustig M, Pauly JM. SPIRiT: iterative self-consistent parallel imaging reconstruction from arbitrary k-space. Magnetic resonance in medicine 2010;64(2):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 Transactions on Computational Imaging 2016;2(4):480–495.

[4] Haldar JP. Low-rank modeling of local k-space neighborhoods (LORAKS) for constrained MRI. IEEE transactions on medical imaging 2013;33(3):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. Magnetic resonance in medicine 2014;72(4):959–970.

[6] Doneva M, Börnert P, Eggers H, Stehning C, Sénégas J, Mertins A. Compressed sensing reconstruction for magnetic resonance parameter mapping. Magnetic Resonance in Medicine 2010;64(4):1114–1120.
[7] Zhang T, Pauly JM, Levesque IR. Accelerating parameter mapping with a locally low rank constraint. Magnetic resonance in medicine 2015;73(2):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(3):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). Magnetic resonance in medicine 2017;78(2):494–507.

[10] Stone SS, Haldar JP, Tsao SC, Sutton B, Liang ZP, et al. Accelerating advanced MRI reconstructions on GPUs. Journal of parallel and distributed computing 2008;68(10):1307–1318.

[11] Sørensen TS, Atkinson D, Schaeffter T, Hansen MS. Real-time reconstruction of sensitivity encoded radial magnetic resonance imaging using a graphics processing unit. IEEE transactions on medical imaging 2009;28(12):1974–1985.

[12] Van Der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering 2011;13(2):22.

[13] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16); 2016. p. 265–283.

[14] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. In: Advances in neural information processing systems; 2019. p. 8026–8037.

[15] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM review 2017;59(1):65–98.

[16] Hansen MS, Sørensen TS. Gadgetron: an open source framework for medical image reconstruction. Magnetic resonance in medicine 2013;69(6):1768–1776.

[17] Uecker M, Ong F, Tamir JI, Bahri D, Virtue P, Cheng JY, et al. Berkeley advanced reconstruction toolbox. In: Proc. Intl. Soc. Mag. Reson. Med, vol. 23; 2015. p. 2486.

[18] Ong F, Lustig M. SigPy: A Python Package for High Performance Iterative Reconstruction. In: Proc. Intl. Soc. Mag. Reson. Med, vol. 27; 2019. p. 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. p. 75.

[20] Milner R. A theory of type polymorphism in programming. Journal of computer and system sciences 1978;17(3):348–375.

[21] Lam SK, Pitrou A, Selibert S. Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC ACM; 2015. p. 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. p. 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. p. 920–923.

[24] Eggers H, Knopp T, Potts D. Field inhomogeneity correction based on gridding reconstruction for magnetic resonance imaging. IEEE Transactions on Medical Imaging 2007;26(3):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(3):394–404.

[26] Pruessmann KP, Weiger M, Scheidegger MB, Boesiger P. SENSE: sensitivity encoding for fast MRI. Magnetic resonance in medicine 1999;42(5):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. Magnetic resonance in medicine 2014;71(3):990–1001.

[28] Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 1952;49(1).

[29] Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM journal on imaging sciences 2009;2(1):183–202.

[30] Parikh N, Boyd S, et al. Proximal algorithms. Foundations and Trends® in Optimization 2014;1(3):127–239.

[31] Goldstein T, Osher S. The split Bregman method for L1-regularized problems. SIAM journal on imaging sciences 2009;2(2):323–343.

[32] Pruessmann KP, Weiger M, Börnert P, Boesiger P. Advances in sensitivity encoding with arbitrary k-space trajectories. Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 2001;46(4):638–651.

[33] Knopp T, Kunis S, Potts D. A note on the iterative MRI reconstruction from nonuniform k-space data. International journal of biomedical imaging 2007:2007.

[34] Pipe JG, Menon P. Sampling density compensation in MRI: rationale and an iterative numerical solution. Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 1999;41(1):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. Magnetic resonance in medicine 2017;77(1):411–421.

[36] Fessler JA, Sutton BP. Nonuniform fast Fourier transforms using min-max interpolation. IEEE transactions on signal processing 2003;51(2):560–574.

[37] Knopp T. Experimental multi-threading support for the Julia programming language. In: Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages IEEE Press; 2014. p. 1–5.

[38] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering 1998;5(1):46–55.

[39] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing 1996;37(1):55–69.

[40] Kukanov A, Voss MJ. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. Intel Technology Journal 2007;11(4).

[41] Frigo M, Johnson SG. 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), vol. 3 IEEE; 1998. p. 1381–1384.

[42] Ong F, Amin S, Vasanawala S, Lustig M. Mridata. org: An open archive for sharing MRI raw data. In: Proc. Intl. Soc. Mag. Reson. Med, vol. 26; 2018. p. 1.
[43] Maier O, Baete SH, Fyrdahl A, Hammernik K, Harrevelt S, Kasper L, et al. CG-SENSE revisited: Results from the first ISMRM reproducibility challenge. Magnetic Resonance in Medicine 2020;.

[44] 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. p. 743.

[45] Gräser M, Thieben F, Szwargulski P, Werner F, Gdaniec N, Boberg M, et al. Human-sized magnetic particle imaging for brain applications. Nature communications 2019;10:1–9.

[46] Block KT, Wiggins R, Yarra - A Toolbox for Clinical MRI Research;. Accessed: 2020-01-20. http://http://yarraframework.com.

[47] Borisch EA, Grimm RC, Riederer SJ. Automated Reconstruction Processing. In: Proc. Intl. Soc. Mag. Reson. Med, vol. 27; 2019. p. 1.