Wagner, Christoph; Semper, Sebastian; Kirchhof, Jan

fastmat: efficient linear transforms in Python

Original published in:	SoftwareX. - Amsterdam [u.a.] : Elsevier. - 18 (2022), art. 101013, 8 pp.
Original published:	2022-02-26
ISSN:	2352-7110
DOI:	10.1016/j.softx.2022.101013
[*Visited:*	2022-05-10]

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

fastmat: Efficient linear transforms in Python

Christoph W. Wagner [*], Sebastian Semper, Jan Kirchhof

Electronic Measurements and Signal Processing (EMS) Group, Technische Universität Ilmenau, Germany

A R T I C L E   I N F O

Article history:
Received 15 September 2021
Received in revised form 7 February 2022
Accepted 7 February 2022

Keywords:
Linear transforms
Linear algebra
Matrix-free
Compressed sensing

A B S T R A C T

Scientific computing requires handling large linear models, which are often composed of structured matrices. With increasing model size, dense representations quickly become infeasible to compute or store. Matrix-free implementations are suited to mitigate this problem at the expense of additional implementation overhead, which complicates research and development effort by months, when applied to practical research problems. Fastmat is a framework for handling large structured matrices by offering an easy-to-use abstraction model. It allows for the expression of matrix-free linear operators in a mathematically intuitive way, while retaining their benefits in computation performance and memory efficiency. A built-in hierarchical unit-test system boosts debugging productivity and run-time execution path optimization improves the performance of highly-structured operators. The architecture is completed with an interface for abstractly describing algorithms that apply such matrix-free linear operators, while maintaining clear separation of their respective implementation levels. Fastmat achieves establishing a close relationship between implementation code and the actual mathematical notation of a given problem, promoting readable, portable and re-usable scientific code.

Code metadata

Current code version	v0.2
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00172
Code Ocean compute capsule	
Legal Code License	Apache License Version 2.0
Code versioning system used	git
Software code languages, tools, and services used	Python, Cython
Compilation requirements, operating environments & dependencies	Numpy, Scipy, Cython, some C-Compiler
If available Link to developer documentation/manual	https://fastmat.readthedocs.io/en/latest/
Support email for questions	sebastian.semper.tu-ilmenau.de christoph.wagner@tu-ilmenau.de

1. Motivation and significance

Linear transformations are one of the corner stones in applied math, physics, engineering and data science for they enable the modeling of objects using vector spaces. In case of finite dimensional vector spaces, linear mappings are represented by matrices that encode how coordinates of an image space can be used to express the image of all basis vectors of another vector space.

However, in many applications linear mappings are not completely arbitrary, but exhibit some structure that often stems from physical models [1,2], descriptions of natural processes [3,4] or assumptions from side constraints [5,6]. One prominent example is the Discrete Fourier Transform (DFT) of order $n \in \mathbb{N}$, which transforms periodic discrete signals into the respective frequency domain and which also has a plethora of applications in spectral analysis, radar, array processing and beyond. Given the canonical standard basis in $\mathbb{C}^n$, the corresponding matrix elements are expressed as

$$\boldsymbol{F} = \left[f_{i,k}\right]_{i,k=1}^{n} = \left[\exp\left(\frac{-\jmath 2\pi}{n} \cdot i \cdot k\right)\right]_{i,k=1}^{n}, \tag{1}$$

where $\jmath = \sqrt{-1}$. As we can see, the Fourier matrix $\boldsymbol{F} \in \mathbb{C}^{n\times n}$ is highly structured with no degrees of freedom, since the size of the involved vector space already fully defines its elements.

In order to actually carry out a linear mapping $\boldsymbol{m}$, which means evaluating it at a given input vector $\boldsymbol{x}$ to get its image $\boldsymbol{y} = \boldsymbol{m}(\boldsymbol{x})$, one simply calculates $\boldsymbol{y} = \boldsymbol{M} \cdot \boldsymbol{x}$, if $\boldsymbol{M} \in \mathbb{C}^{m\times n}$ is the matrix encoding the linear mapping $\boldsymbol{m}$. Generally, this is accomplished

[*] Corresponding author.
E-mail address: christoph.wagner@tu-ilmenau.de (Christoph W. Wagner).

by using

$$\boldsymbol{y} = [y_i]_{i=1}^{m} = \left[\sum_{j=1}^{n} \boldsymbol{m}_{i,j} \cdot x_j \right]_{i=1}^{m} = \boldsymbol{M} \cdot \boldsymbol{x}. \tag{2}$$

Note that this formula entails a computational complexity of $\mathcal{O}(n \cdot m)$, both in terms of runtime and memory consumption.

Returning to our DFT example, instead of simply applying (2) literally, one should make use of the Fast Fourier Transform (FFT) [7] in order to calculate $\boldsymbol{y} = \boldsymbol{F} \cdot \boldsymbol{x}$ with runtime complexity $\mathcal{O}(n \log n)$. The FFT exploits structural redundancy by decomposing the DFT matrix into smaller so-called butterfly-structures, depending on the transform size and its prime factorization. Such redundancy reductions are key in enabling practical applications of modern signal processing.

Dedicated and highly optimized libraries [7–10] exist that offer "simple" access to the FFT, but performance and developer-experience alone are not their only benefits: Since they are highly tested and thoroughly examined, they tend to produce less "soft" errors (i.e. numerical accuracy or -stability) than some custom code written from scratch. Especially since devising numerically stable algorithms is far from trivial, such abstract libraries are often considered highly trusted. As a result, the FFT is commonly considered as the primary choice for spectral analysis over the DFT – not only for its better performance, but also due to developer-experience and trust.

For many such matrix–vector-products an efficient algorithm, tailored to the specific linear mapping at hand, can be implemented, giving rise to the distinction of *two* representations of that mapping: The matrix itself as a rectangle of scalars (in our example the DFT-matrix $\boldsymbol{F}$), which we call the *dense* representation, and said algorithm (FFT) that replaces the matrix–vector product entirely, which we call the *matrix-free* representation. We further define the *forward transform* $\boldsymbol{\phi_M} : \mathbb{C}^n \to \mathbb{C}^n$ as

$$\boldsymbol{x} \mapsto \boldsymbol{\phi_M}(\boldsymbol{x}) = \boldsymbol{m}(\boldsymbol{x}) = \boldsymbol{Mx} \tag{3}$$

and the *backward transform* $\boldsymbol{\beta_M} : \mathbb{C}^n \to \mathbb{C}^n$ as

$$\boldsymbol{x} \mapsto \boldsymbol{\beta_M}(\boldsymbol{x}) = \boldsymbol{M}^H \boldsymbol{x}, \tag{4}$$

where $\boldsymbol{M}^H$ denotes the Hermitian transpose of the matrix $\boldsymbol{M}$.

To derive an important implication of handling (more complex) *matrix-free* representations, let us consider the cyclic convolution with a vector $\boldsymbol{c} \in \mathbb{C}^n$, which is usually implemented as

$$\boldsymbol{y} = \mathrm{ifft}(\mathrm{fft}(\boldsymbol{c}) \odot \mathrm{fft}(\boldsymbol{x})), \tag{5}$$

where $\odot : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n$ carries out a pointwise multiplication (i.e. Hadamard product) of two vectors in $\mathbb{C}^n$. We can now see that the two representations begin to diverge (both in notation and program code), making it both harder to cope with on a theoretical level and keeping both representations in-sync. As a consequence, the risk of implementation errors increases and researchers get distracted from their main quest by obfuscated ("bloated") code and all kinds of maintenance issues.

2. Software description

Our software provides an object-oriented programming interface for matrix-free linear operators that closely ties resulting application code to its corresponding mathematical notation. The unique `Matrix`-class Application Programming Interface (API) abstracts any (internal) *matrix-free* representation in such form that they can be treated like *dense* representations (externally). Binary and unary operators (such as matrix–matrix products, unfoldings of structured tensors or block matrices) further provide the ability to combine existing operators for constructing complicated linear mappings abstractly, while exploiting matrix-free benefits along the way. A user may choose from a wide range of built-in matrices (see Fig. 1), or decide to implement a custom one using `fastmat`'s rich built-in functionality for quickly assessing numerical accuracy, class performance or run-time execution-path-optimization. Finally, the `Matrix`-class-API is complemented by the `Algorithm`-API for implementing (signal processing) algorithms that utilize `Matrix` operators.

2.1. General Matrix-free Representations

Every linear transform can be described by a matrix (or interactions of matrices) in either its *dense* or *matrix-free* representation. The `Matrix` baseclass implements a general and structure-independent basic API, from which more specific class implementations can be derived using Object-Oriented-Programming (OOP)-principles. The user is thus empowered to easily and selectively extend an existing `Matrix` class with only having to implement new functionality, as new classes by default inherit all structure from their parent. This is possible since the class-API makes no structural assumptions about the linear transform internally. Our architecture thus reflects the path from no structural assumptions (`Matrix` baseclass) to more constraints, less generality and more efficiency (inherited classes) in a way that is completely transparent to the user, which promotes code portability and -reuse.

Many properties of a given matrix $\boldsymbol{M}$ can also be calculated from the forward and backward transforms in Eqs. (3) and (4). For instance, the single matrix element $m_{i,j}$ can be accessed via $\boldsymbol{e}_i^T \boldsymbol{\phi_M}(\boldsymbol{e}_j)$, independently of $\boldsymbol{M}$ or $\boldsymbol{\phi_M}$. The `Matrix` baseclass offers plenty such general implementations of common quantities and properties (see Table 1), which also benefit from overloading efficient means for $\boldsymbol{\phi}$ and $\boldsymbol{\beta}$ in a specialized class.

All properties and methods of the class-API are wrapped with some entry-level interface code that ensures input sanity and offers extended features, such as run-time execution optimizations (see Appendix B). Still, the mechanism ensures that a user can override their central computation code without regard for not breaking this interface code (see Appendix C). By definition, all `Matrix` instances and their instantiation arguments are considered immutable, as this provides the immense benefit of allowing for pre-computations and caching of static results without impeding run-time consistency. The latter is provided transparently by the class-API interface code for the respective general properties.

A good illustration for these mechanisms is the `Circulant` class, which realizes the computation given in (5). Since $\boldsymbol{c}$ accurately defines the circulant matrix, pre-computing $\mathrm{fft}(\boldsymbol{c})$ is done during instantiation and its result is then reused in every subsequent call to $\boldsymbol{\phi}$ and $\boldsymbol{\beta}$.

As a manifestation of the overall architecture philosophy, the `Circulant` matrix is also not implemented by hacking down (5) in closed form, but rather by defining it as a product of three matrices: A conjugated and transposed Fourier matrix, a diagonal matrix and another Fourier matrix. Given that we have implemented the `Product`, `Fourier` and `Diag` classes correctly, we can rely on them to serve as abstraction layers for the definition of `Circulant` with an overall efficiency comparable to a closed-form implementation.

A wide range of operators is provided with `fastmat` out-of-the-box (see Fig. 1), which regularly reference each-other. As a consequence, the code of `fastmat` also resembles the mathematical notation closely, while runtime and memory performance benefits from any piece of *matrix-free* implementation anywhere in the class architecture.

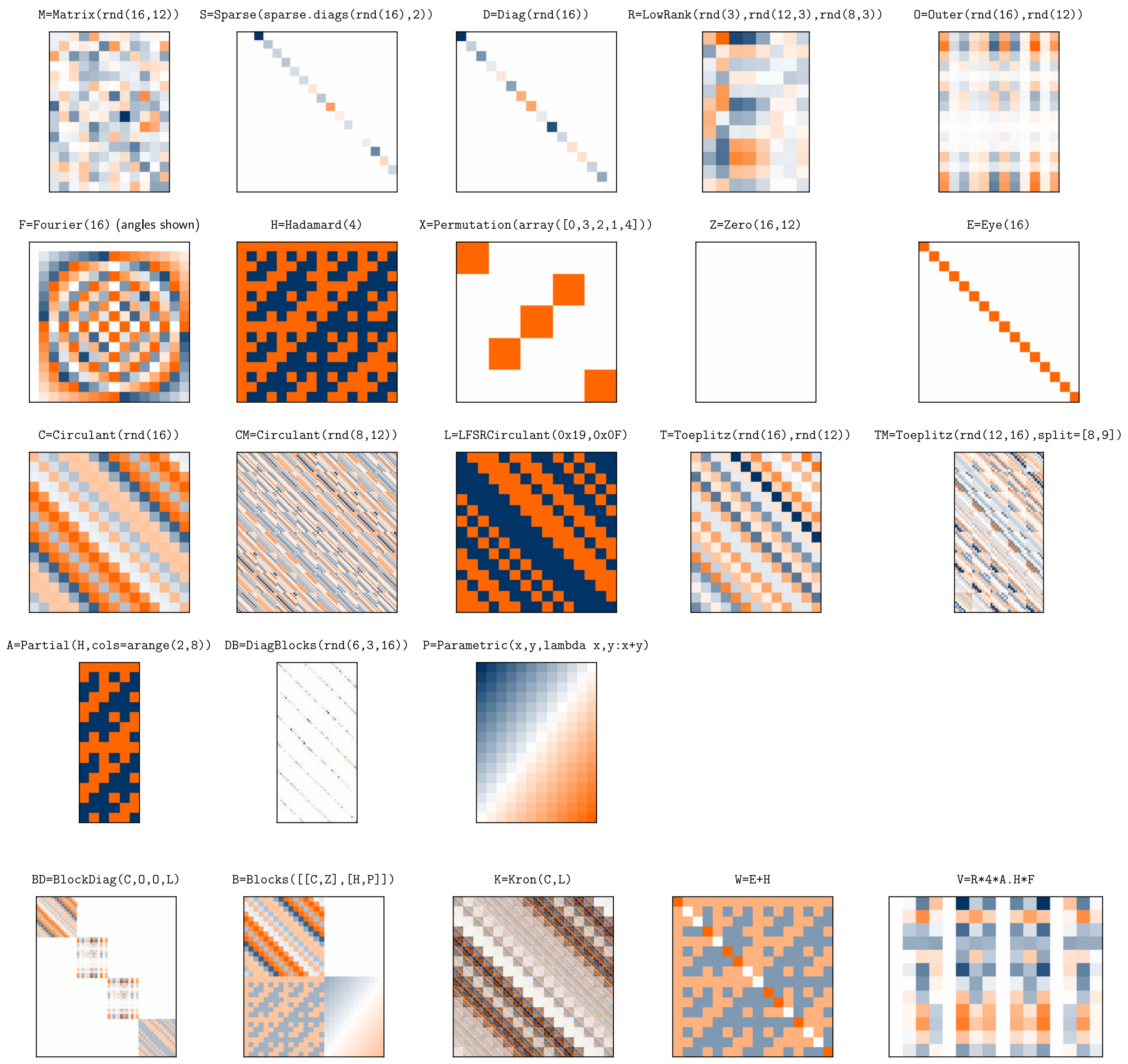

Fig. 1. Listing of `fastmat` classes, grouped by elementary and composite type.

For being able to rely on individual operator correctness (especially as they tightly interact with each other), `fastmat` offers a flexible unit-test mechanism, which is capable of running fine-granular automated tests for any `fastmat` class implementation (including user-defined ones) at run-time. Even subtle implementation errors rooted deep inside a complicated web of hierarchical *matrix-free* operators can be identified quickly using the hierarchical analysis of the unit-test results. For more information consult the package documentation.

In summarizing, for certain types of composite operators, their respective *forward* and *backward* transforms can themselves be described solely from the respective transforms of their operator terms. Structural knowledge is fully retained even through multiple layers of structural abstraction and a close relationship between `fastmat`-powered code and its theoretical computation description using mathematical notation is established, meeting `fastmat`'s main architectural goal.

2.2. Matrix-free Algorithm Representation

On top of the pure matrices we also offer the `Algorithm` baseclass that facilitates the usage of `Matrix` instances for more complex routines that make use of linear transforms relying on the `fastmat Matrix` class-API. This provides some significant advantages: First, the algorithm implementation itself is completely agnostic about the actual structure of the underlying matrices it employs. As a direct consequence, any `Matrix` instance, that exploits structural knowledge for efficiency, also passes its benefits on to any `Algorithm` that it is employed in. Secondly, the `Algorithm` baseclass offers means to easily establish user callbacks for various purposes, making it easily possible to inspect, tune or override certain aspects of an algorithm during runtime, such as break conditions, step sizes, thresholding or the custom determination of regularization parameters. And last but not least, of course does the algorithm's code structure also reflect

Table 1
The `Matrix` baseclass: properties, methods, overloading and caching.

Feature	Notation	Operator	Matrix Attribute
Dense representation	$\boldsymbol{M}$	`M[...]`	`A.array`
Element Indexing*	$\boldsymbol{M}_{i,j}$	`M[i, j]`	`A.getItem(i, j)`
Row Access*	$\boldsymbol{M}_{i,\bullet}$	`A[row, :]`	`A.getRow/getRows(i)`
Column Access*	$\boldsymbol{M}_{\bullet,j}$	`A[:, col]`	`A.getCol/getCols(j)`
Row Norms	$\frac{\boldsymbol{M}_{i,\bullet}}{\lVert M_{i,\bullet}\rVert}$		`A.rowNorms`
Column Norms	$\lVert M_{\bullet,j}\rVert$		`A.colNorms`
Row Normalization	$\lVert M_{i,\bullet}\rVert$		`A.rowNormalized`
Column Normalization	$\frac{\boldsymbol{M}_{\bullet,j}}{\lVert M_{\bullet,j}\rVert}$		`A.colNormalized`
Transpose	$\boldsymbol{M}^T$		`M.T`
Hermitian Transpose	$\boldsymbol{M}^H$		`M.H`
Complex Conjugate	$\mathrm{conj}(\boldsymbol{M})$		`M.conj`
Inverse	$\boldsymbol{M}^{-1}$		`M.inverse`
Pseudoinverse	$\boldsymbol{M}^{+}$		`A.pseudoInverse`
Gramian	$\boldsymbol{M}^H\boldsymbol{M}$		`A.gram`
Scipy Linear Operator			`A.scipyLinearOperator`

**Does not offer internal caching by the _-prefix syntax.*

the mathematical notation more closely in its implementation, especially since the heavy lifting required for *matrix-free* representations no longer clutters the algorithm's implementation. Most pitfalls associated with overly complicated code units are directly alleviated through the resulting code-tidying, provided by the `Matrix` and `Algorithm` class-API of `fastmat`.

3. Illustrative Examples

After establishing `fastmat`'s design philosophy, we will now consider how a user solves linear problems with `fastmat`, focusing on workflow and methodological implications arising from using *matrix-free* representations. Let us consider the case, where some linear mappings in simulation or measurement analysis are so complex and large in size that choosing *dense* representations regularly lead to runtime and/or memory exhaustion. Still, due to them being related to physical processes, their embedded structure gives rise to optimization potential from using *matrix-free* representations. Unfortunately, embedding such representations in scientific code often leads to the situation, where representation-level code optimizations clutter the higher-level application code, adding a readability and maintenance penalty. The resulting code of such solutions is highly specific, causing portability issues and impeding design reuse.

In applying the added abstraction of `fastmat`'s class-API, this crisis can be averted, freeing valuable resources for advancing the actual problem — while still being able to resolve acute bottleneck-problems (due to the underlying *matrix-free* implementations). The atomic class implementations encourage building shared libraries and the API also serves as a useful entry-point for numerical unit-tests, ensuring that the linear operators meet specification, independent from application-level testing.

In combination, these effects encourage the use of *matrix-free* methods over sticking with badly-performing *dense* representations out of fearing the opportunity cost of trying.

3.1. Defining your Custom Matrix-free Operators and Algorithms

Before diving into a full-blown example of higher-dimension physical modeling in Section 3.2, we will use `fastmat` for a quick matrix-free implementation of a **1** matrix of shape $N \times M$, defined such that every scalar element of its *dense* representation equals to 1.

Choosing a proper baseline class to inherit from may save much implementation effort for a new class, which very well may be any other `Matrix` class, with some parametrization. For our example, the `Matrix` baseclass is already well suited. The general behavior of the new class is defined by overriding the methods `_forward(•)` and `_backward(•)`, corresponding to $\boldsymbol{\phi}$ and $\boldsymbol{\beta}$. The `__init__(•)` method must be specified, which performs pre-computations (if required) and either finishes with calling `self.initProperties(•)` directly, or by passing on to its parent class' `__init__(•)`. Due to the general `Matrix` baseclass implementations, the class is now complete (see Listing 1). If known, further optimizations may be added through overloading specific class properties and methods (see Appendix C).

The remainder of this section will shortly outline how to make use of the `fastmat` testing-, benchmark- and calibration systems for a new class. The testing subsystem, designed to unit-test a particular class for numerical accuracy, requires the definition of two methods: The `reference()` method is supposed to return a dense reference array to compare against, while `_getTest()` must return a valid configuration for the test case generator. For the calibration feature to be available, `_getComplexity()` must implement a model for returning relative complexity estimates for $\boldsymbol{\phi}$ and $\boldsymbol{\beta}$ and finally, the run-time benchmark must be configured by declaring `_getBenchmark()` such that a valid benchmark configuration is returned. With those four methods declared, all features of `fastmat` are now enabled for the newly defined class.

To make use of run-time execution-path optimization (see Appendix B), either load pre-existing calibration data from disk, or generate it at run-time (prior to instantiating •) by invoking `fastmat.core.calibrateClass(•)`.

```python
import numpy, fastmat

class Ones(fastmat.Matrix):
  '''A simple 1-Matrix implementation'''

  def __init__(self, numRows, numCols):
    '''Initialize fastmat class-API and define shapes'''
    self._initProperties(numRows, numCols, numpy.int8)

  def _forward(self, arrX):
    '''Define forward transform'''
    return numpy.resize(
      numpy.sum(arrX, axis=0, keepdims=True),
      (self.numRows, arrX.shape[1])
    )

  def _backward(self, arrX):
    '''Define backward transform'''
    return numpy.resize(
      numpy.sum(arrX, axis=0, keepdims=True),
      (self.numCols, arrX.shape[1])
    )

  ### Additional methods required to be defined for...
  ###  - Testing: reference(), _getTest()
  ###  - Calibration: _getComplexity(), _getBenchmark()

# #################### Instantiate and Use the new class
N, M = 4, 5            ; O = Ones(N, M)
xf = numpy.random.randn(M) ; xb = numpy.random.randn(N)
print(xf, '\n', O * xf)    ; print(xb, '\n', O.H * xb)
```

Listing 1: Minimal Working Example of a matrix-free implementation of our **1**-Matrix.

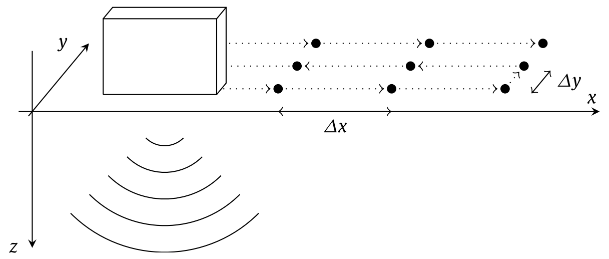

Fig. 2. Transducer path and the measurement positions.

3.2. Application to Ultrasonic Nondestructive Testing

Ultrasound is used in a wide range of fields such as manufacturing, medicine or maintenance as a nondestructive modality to characterize the interior of an object under test. In non-destructive testing one is further interested in detecting and characterizing possible defects within a specimen without afflicting any damage or alterations to the object. Following, we summarize an example of recent research carried out on the 3D reconstruction of defects using ultrasonic non-destructive testing [11] with special focus on its implementation using matrix-free methods and the proposed package `fastmat`.

A transducer, emitting ultrasonic pulses into the specimen, is moved along a predefined trajectory to collect the measurement data. At each position, we take snapshots by inserting a known waveform and recording the echoes originating at material boundaries, due to the change in acoustic impedance. This process, as depicted in Fig. 2, creates a synthetic aperture, which allows us to recover the (previously unknown) locations of such rapid impedance changes in the investigated medium. These usually indicate interesting areas, which we aim to extract from the measurements.

In order to formulate a tractable parameter estimation problem, we first introduce a linear model, which maps a vector encoding possible defect locations onto a possible observation captured by the transducer. Very generally, this reads as

$$\boldsymbol{b} = \boldsymbol{H} \cdot \boldsymbol{x} + \boldsymbol{n}, \tag{6}$$

where $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{C}^n$ and $\boldsymbol{H} \in \mathbb{C}^{n \times n}$. In this equation, $\boldsymbol{b}$ is the obtained measurement from the transducer as in Fig. 2. The matrix $\boldsymbol{H}$ models our assumptions of the physical process taking place in the medium: if we insert pulsed waveforms, these pulses get reflected by possible defects and can be seen in the recorded echos. Based on this model, the entries in $\boldsymbol{x}$ correspond to certain positions in the object. A nonzero entry indicates a reflector or defect at said position. Consequently, we wish to recover $\boldsymbol{x}$ from the measurements $\boldsymbol{b}$ in order to infer these positions.

The transducer moves over points that are aligned on a regular 2*D* grid located in a hyperplane of $\mathbb{R}^3$. In case when we introduce a well chosen grid for the possible defect locations, one can show that the matrix

$$\boldsymbol{H} = \left[\boldsymbol{H}_{i,j}\right]_{i,j=1}^{N_1},$$

is a block matrix, where each $\boldsymbol{H}_{i,j} \in \mathbb{C}^{(2N_2-1)(2N_3-1)\times(2N_2-1)(2N_3-1)}$ is a 2-level Toeplitz matrix. Since N_1, N_2 and N_3 scale with the spatial resolution during measurement and reconstruction of the defects locations, as they constitute the number of samples in each of the spatial directions. The dense matrix $\boldsymbol{H}$ quickly assumes an insane amount of elements $(N_1^2(2N_2-1)^2(2N_3-1)^2)$. Since this is simply too much for current system memory and computation resources we have to employ a matrix-free implementation of the transforms involving the dense matrix $\boldsymbol{H}$ to stay within the realms of practical feasibility.

Now, consider a single $\boldsymbol{H}_{i,j}$, which due to its structure represents a 2D non-cyclic convolution. Such a convolution can be implemented by means of a larger cyclic convolution and appropriate zero-padding of the input. Leaving out technical details, one can show that

$$\boldsymbol{H}_{i,j} = \mathcal{P}\left((\boldsymbol{F}_{2N_2-1} \otimes \boldsymbol{F}_{2N_3-1}) \cdot \operatorname{diag}(\boldsymbol{h}_{i,j}) \cdot (\boldsymbol{F}^H_{2N_3-1} \otimes \boldsymbol{F}^H_{2N_2-1})\right), \tag{7}$$

where $\otimes$ denotes the matrix Kronecker product, $\operatorname{diag}(\boldsymbol{x})$ is a diagonal matrix with $\boldsymbol{x}$ as its diagonal and $\mathcal{P}$ is the operator in charge of the correct zero-padding. To implement this matrix, one would use a suitable combination of `Partial`, `Kron`, `Diag`, `Product`, `Hermitian` and `Fourier` classes of `fastmat`.

In order to carry out the estimation of $\boldsymbol{x}$ based on $\boldsymbol{H}$ and $\boldsymbol{b}$, one popular approach is the following optimization problem:

$$\min_{\boldsymbol{x}} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{b}\|_2^2 + \tau\|\boldsymbol{x}\|_1, \tag{8}$$

where $\|\boldsymbol{x}\|_p = (\sum_i |x_i|^p)^{(1/p)}$ and $\tau > 0$ being some suitably chosen hyper parameter. Interestingly, there are algorithms to approximately solve problems like in (8) efficiently, solely based on matrix–vector products. It has been discovered in recent years, that the regularizing term $\tau\|\boldsymbol{x}\|_1$ is able to enforce *sparsity* in $\boldsymbol{x}$, which corresponds to the assumption that there are only a few defects within the specimen under test. One way of solving the problem in (8) is the Iterative Shrinkage-Thresholding Algorithm (ISTA) [12], which is the basic inspiration for a whole array of similar algorithms with varying performance. However, for the sake of simplicity, we present the simple and basic version. For a given initial estimate $\boldsymbol{x}_0 \in \mathbb{C}^n$ we simply iterate the following expressions until we have satisfied some stopping criterion:

$$\boldsymbol{x}_{k+1} = \boldsymbol{\tau}_s\left(\boldsymbol{H}^H \cdot (\boldsymbol{H} \cdot \boldsymbol{x}_k - \boldsymbol{b})\right),$$

where $\boldsymbol{\tau}_s : \mathbb{C}^n \to \mathbb{C}^n$ is a non-linear function (called the soft thresholding operator) to the level $s > 0$, which exhibits linear runtime over the size of $\boldsymbol{x}$. This means the largest computational effort for ISTA resides in the computation of the $\boldsymbol{\phi_H}$ and $\boldsymbol{\beta_H}$ projections respectively, rendering it a perfect application for a matrix-free implementation. For ISTA we simply end up with

$$\boldsymbol{x}_{k+1} = \boldsymbol{\tau}_s\left(\boldsymbol{\beta_H}(\boldsymbol{\phi_H}(\boldsymbol{x}_k) - \boldsymbol{b})\right).$$

The fact, that the modeling stage and the algorithm design stage remain completely separate from each other, renders the

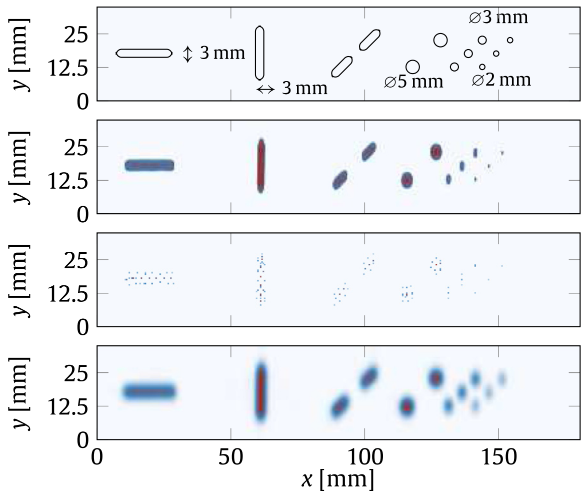

Fig. 3. *Sparse recovery results for a steel specimen (top view)* – Top to bottom: Sketch of the specimen (Top View), Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) reconstruction, Orthogonal Matching Pursuit (OMP) reconstruction, beamforming via Synthetic Aperture Focusing Technique (SAFT).

proposed toolbox very advantageous as a research tool. The same matrix-free representation can easily be plugged into other suitable algorithms that use the described class-API.

On the other hand, we can also easily adapt for a different measurement scenario. Compressed Sensing (CS) uses the techniques developed in Sparse Signal Recovery (SSR) for estimating signal parameters from compressed measurements. If we apply this approach to the problem of defect detection we get a different observation model, which reads as

$$\hat{\boldsymbol{b}} = \boldsymbol{\Psi} \cdot \boldsymbol{H} \cdot \boldsymbol{x} + \boldsymbol{n}, \tag{9}$$

where the matrix $\boldsymbol{\Psi} \in \mathbb{C}^{m \times n}$ is such that $m \ll n$, resulting in a $\hat{\boldsymbol{b}}$, containing far fewer measurements than $\boldsymbol{b}$.

In research on ultrasonic non-destructive testing, a common procedure is to compare imaging algorithms using a test specimen with artificially introduced target defects. Such defects are commonly represented by flat holes that are drilled into the bottom of the specimen. In Fig. 3 we show some examples for possible reconstructions of such defects in a steel block. In this case we use the transducer to localize these defects by measuring into the specimen from the top side. See Fig. 3 for results of three different algorithms, namely FISTA [12], a slight variant of the described ISTA, OMP [13] (another sparsity-enforcing algorithm well suited for matrix-free methods), and SAFT, a beamforming method that simply computes $\hat{\boldsymbol{x}} = \boldsymbol{\beta}_{\boldsymbol{H}}(\boldsymbol{b})$.

In our case we use $\boldsymbol{\Psi}$ to compress the $N_2 \cdot N_3$ many single-pulse-echo-measurements independently in frequency domain. This way we get

$$\boldsymbol{\Psi} = \mathrm{blkdiag}\{\boldsymbol{\Phi}_{1,1}, \ldots, \boldsymbol{\Phi}_{N_1,N_2}\},$$

where each $\boldsymbol{\Phi}_{i,j}$ is a matrix containing a random selection of $n_f \in \mathbb{N}$ rows of the Fourier matrix and the blkdiag operator arranges the matrices in a block-diagonal structure. Clearly, the measurement matrix $\boldsymbol{\Psi}$ has a matrix-free representation and as such also does the product $\boldsymbol{\Psi H}$. If we simply plug the new measurements $\hat{\boldsymbol{b}}$ and the product $\boldsymbol{\Psi H}$ into the already discussed algorithms like OMP, FISTA or ISTA, we get results as depicted in Fig. 4. There, we only used a single Fourier coefficient from each of the originally collected pulse-echo-measurements.

To proof our claim, that `fastmat` tightens code-notation relationship, we show a simplified example implementation of (7) and (8) in Listing 2. The example not only shows that modeling and algorithmic design is fully separated, is also emphasizes the value of properly abstracted models, by easily adding a compressive data acquisition model to the design. In then swapping models (reflecting different sampling schemes or physical models) and algorithms in-place, a wide range of research results is now quickly attained, while reusing most of the easy-to-follow programming code, that also resembles our mathematical formulation closely.

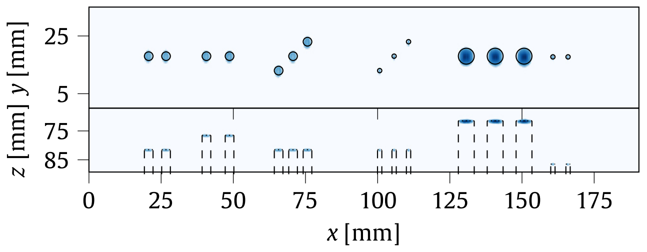

Fig. 4. Top and side view of a FISTA reconstruction using 20 steps, $n_f = 1$ Fourier coefficients and $\boldsymbol{\Psi H}$ as the system matrix.

```python
# either you could go by coding Eq. (7) literally,
FK = fastmat.Kron(
  fastmat.Fourier(N2), fastmat.Fourier(N3)
)
H = fastmat.Blocks([
  [ fastmat.Partial(
    FK * fastmat.Diag(h[i, j]) * FK.H, cols=vec_zeropad
  )
    for j in range(N1)
  ] for i in range(N1)
])

# or you could just use the Multi-Level Toeplitz class
H = fastmat.Blocks([
  [ fastmat.Toeplitz(h[i, j])
    for j in range(N1)
  ] for i in range(N1)
])

# Using H in an algorithm (8) works as follows:
import fastmat.algorithms
alg = fastmat.algorithms.ISTA(H, numLambda=lambda)
x = alg.process(b)
```

Listing 2: The code implementation for Eq. (7) (two variants given) and Eq. (8).

4. Impact

By design of the API presented in Section 2, a research programmer can abstract low-level optimization away from high-level algorithm design without sacrificing general performance optimizations, resulting in greatly improved readability, extensibility and portability of the produced code. Due to the OOP approach resembling a construction-kit, linear mappings can be implemented very rapidly, even for complex *matrix-free* representations with high performance optimization potential. The integrated testing functionality greatly reduces maintenance time by focussing debugging effort at the precise abstraction level required, even for complex implementations. Consequent abstraction establishes the isolation of individual efficiency-boosting tweaks to their particular architecture level. This improves on code maintenance, interface clarity and early bug-detection all the way up to the algorithmic design level, accelerating development cycles.

All those features result in `fastmat` being a tool to boost the work efficiency of researchers, engineers or teams thereof, that regularly have to deal with large-scale structural models and algorithms that apply those. Since team communication is limited (even already by available time), this puts a direct limit on the sustainable problem complexity and team efficiency – especially when groups grow in size [14] – introducing a "specialization trap", where at some point every colleague in a design team maintains some different, highly specialized piece of a large common project, that is infeasible to communicate fully to others (due to overall complexity). Our package helps breaking this situation by enabling individual team members to better understand their peers problems, exchange ideas quicker and find solutions, improving their overall team collaboration efficiency to prepare for solving even more complex problems.

On top of the team-dynamic aspects, the resulting implementation efficiency is also sufficiently close to the optimum, such that the opportunity cost of living with the remaining efficiency gap (that stems from the implementation generality) is low. Therefore, the best of both worlds can be achieved in that convenience no longer must be sacrificed for speed (and vice-versa).

Finally we would like to reference original research that applied `fastmat`. In Section 3.2 we showed its application to the problem of CS-based 3D reconstruction of defects from ultrasound measurements, involving dense, yet highly structured multi-dimensional linear models. `fastmat` enabled this research by reducing the memory footprint of these models from many petabytes down to the gigabyte-range [11,15,16]. For a novel CS-based ultra-wideband radar architecture, `fastmat` was used to model the whole signal-acquisition hardware frontend to assess the impact of circuit design and signaling choices to the resulting reconstruction performance [17]. In the field of machine learning, our package is utilized to perform Structured Kernel Interpolation (SKI) for scaling Gaussian Processes (GP) to massive datasets on the example of three-dimensional weather radar datasets with over 100 million points [18]. One mentionable application of our algorithmic subsystem, being used for its high-performing efficient rank-1 update OMP implementation, is in `pygpc`, a sensitivity and uncertainty analysis toolbox for Python [19].

5. Conclusions

We have shown that with a carefully designed API one can design an architecture that unifies dense and matrix-free linear mappings, while leveraging the strengths of both approaches, and keeping the complications between them at a minimum. As we demonstrate this results in simplifications both in terms of the written code as well as the design process that is needed to construct a specific implementation. In essence, the described package allows to focus more on the actual problems at hand, when dealing with large-scale structured linear models, without being subjected to the obstacles of realizing them from scratch. Future versions of the package will include support for tensor unfoldings in order to realize higher-order linear mappings more rigorously. Further, improved support for the specification of more complex block structures aims to make better use of the redundancy occurring from repeating structures within a matrix-presentation.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to express thanks for the support received during the development of `fastmat` by the Carl-Zeiss Foundation under the project “PRIME”, by the DFG under the projects “CoSMoS” and “HoPaDyn” as well as the Free State of Thuringia and the European Social Fund supporting the HyLoC research group under the reference “2019 FGR 0100”. We acknowledge support for the publication costs by the Open Access Publication Fund of the Technische Universität Ilmenau.

Appendix A. Prevent Python performance penalties: Cython

Partly due to its interpreted, abstract and permissive (duck-typing) nature, Python is not particularly known for rapid code execution. To mitigate this we make extensive use of Cython throughout the architecture to reduce the additional call overhead (introduced with our object oriented model) by reducing Python interpreter interaction where possible. Especially for matrix-free operators that combine thousands small (N)-operators, curbing Python’s flexibility where it is not of benefit (e.g. under the hood of `fastmat`) becomes imperative, as (superfluous) interpreter interaction quickly begins to dominate runtime.

To minimize call overhead, the complete class model architecture and all built-in class implementations utilize Cython to compile the code and statically-link it back into the Python API. This way the Python interpreter can be evaded not only during intra-package execution flow (e.g. method calls), but also for most interactions with Numpy (by means of its provided C-API), resulting in a significant performance boost over pure (interpreted) Python code. Dodging the runtime penalty that comes with runtime dynamic typing, `fastmat` internally makes heavy use of static typing and supports the data types integer (8, 16, 32 and 64 bit), float and complex (both for single and double precision). Another runtime penalty arises from the creation of vast amounts of Views into `ndarray`‘s during internal array slicing operations. This is mitigated by the use of a dedicated mutable striding operator that works on arrays, which greatly relieves the stress of creating and garbage-collecting many short-lived Python objects, such as array views.

Appendix B. Class calibration and execution path optimization

Since a modern CPU is highly optimized for crunching numbers, the *dense* representation is likely to compute faster than the *matrix-free* for very small transform sizes — even if complexity analysis predicts the opposite. This is mostly due to the scaling-insensitivity of complexity models, but is also promoted by the impact of processor caches and memory speeds. In this case, using the conventional *dense* matrix–vector “dot” product as a shortcut might be more efficient than using the *matrix-free* $\boldsymbol{\phi}$ or $\boldsymbol{\beta}$ representation. However, determining the threshold problem size, from below which the *dense* representation should be preferred, is highly individual to any particular problem and processing system and must be determined at run-time.

In `fastmat` this process is called *calibration* and is available for any `Matrix` class (also user-defined ones) as part of the normal user interface. For this, `fastmat` offers a simple interface for the one-time generation of problem-size-agnostic runtime estimation models (which also consider simultaneous processing of multiple vectors) for $\boldsymbol{\phi}$ and $\boldsymbol{\beta}$. During the instantiation of a `Matrix` instance its runtime performance and that of a comparable dense product are estimated, given calibration data is available in the context of the current session. When $\boldsymbol{\phi}$ or $\boldsymbol{\beta}$ is finally used, the optimal path for a particular number of simultaneous vectors can immediately be chosen. This way, `fastmat` features integrated run-time computation path optimization.

Appendix C. Overloading classes from a user’s perspective

To promote usability, a major challenge for `fastmat` is to simultaneous achieve the following goals: First, a `Matrix` instance must behave the same regardless of context as long it makes sense mathematically. Therefore, in embracing the *duck typing* mantra of Python, input and output sanitizing is required for all user-interface methods — regardless of whether they are subject to overloading or not. Secondly, it must be easily possible to define new or extend existing classes, regardless of them being built-in (Cython) or user defined (Python), or to override particular portions of any classes’ take on the user interface. And finally, readability, use- and reusability must be maintained to establish a user experience of lowest possible complexity. The user must be able to enjoy the comfortable position of not having to care about breaking the internal API or functionality when overloading classes.

Adding an extra input preparation layer directly at the class-API entry methods of the `Matrix` baseclass achieves all goals. Every class-API method takes care of input compliance and applies more advanced features, such as pre-computation, result caching or runtime execution path optimization (*calibration*, see Appendix B). The actual implementation code is provided in a separate private method. Since the user only overrides these private methods when implementing custom structured matrices, interface consistency is always guaranteed.

For properties that apply pre-computation caching, the overall code execution (and implementation overloading) mechanism is as follows: Assume one wishes to access the property `A.foo`. There exists a shadowed attribute `A._foo`, which caches the computation result for immediate return in all subsequent queries. During the first query, `A.getFoo()` is invoked as the direct API entry point for the computation of 'foo'. This entry point handles interface integrity (similar to any other class-API method) and may not be overloaded directly. Instead, it redirects the execution flow to `A._getFoo()` for the actual computation, which is the lean method that a user would overload. Therefore, if a user wants to alter the behavior of `A.foo` or `A.getFoo()`, overloading `A._getFoo()` with a new implementation is all that needs to be done.

References

[1] Knabner P, Angermann L. Numerical methods for elliptic and parabolic partial differential equations. New York, NY, New York: Springer; 2003.

[2] Deans SR. The radon transform and some of its applications. reprint of the 1993 original. Mineola, NY: Dover Publications. xii; 2007, p. 295.

[3] Ludwig D. The radon transform on euclidean space. Comm Pure Appl Math 1966;19(1):49–81. http://dx.doi.org/10.1002/cpa.3160190105.

[4] Quinto ET. An introduction to *X*-ray tomography and radon transforms. Providence, RI: American Mathematical Society (AMS); 2006.

[5] Davis TA, Hu Y. The University of Florida Sparse Matrix Collection. ACM Trans Math Software 2011;38(1). http://dx.doi.org/10.1145/2049662.2049663.

[6] Shepp LA, Logan BF. The Fourier reconstruction of a head section. IEEE Trans Nucl Sci 1974;21:21–43.

[7] Johnson SG, Frigo M. Implementing FFTs in Practice. In: Burrus CS, editor. Fast Fourier transforms. Rice University, Houston TX: Connexions; 2008, URL http://cnx.org/content/m16336/.

[8] Frigo M, Johnson S. The Design and Implementation of FFTW3. Proc IEEE 2005;93(2):216–31. http://dx.doi.org/10.1109/JPROC.2004.840301.

[9] Frigo M. A Fast Fourier Transform Compiler. In: Proceedings of the ACM SIGPLAN 1999 conference on programming language design and implementation. New York, NY, USA: Association for Computing Machinery; 1999, p. 169–80. http://dx.doi.org/10.1145/301618.301661.

[10] Frigo M, Johnson S. FFTW: an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE international conference on acoustics, speech and signal processing (Cat. No.98CH36181). vol. 3, 1998, p. 1381–4 vol.3. http://dx.doi.org/10.1109/ICASSP.1998.681704.

[11] Kirchhof J, Semper S, Wagner CW, P'erez E, Römer F, Del Galdo G. Frequency Sub-Sampling of Ultrasound Non-Destructive Measurements: Acquisition, Reconstruction and Performance. IEEE Trans Ultrason Ferroelectr Freq Control 2021;68(10):3174–91. http://dx.doi.org/10.1109/TUFFC.2021.3085007.

[12] Beck A, Teboulle M. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. SIAM J Imaging Sci 2009;2(1):183–202. http://dx.doi.org/10.1137/080716542.

[13] Tropp JA, Gilbert AC. Signal Recovery From Random Measurements Via Orthogonal Matching Pursuit. IEEE Trans Inform Theory 2007;53(12):4655–66. http://dx.doi.org/10.1109/TIT.2007.909108.

[14] Brooks FP. The mythical man-month. Anniversary ed., reprinted with corr., [with 4 new chapters]. Addison-Wesley; 1995.

[15] Semper S, Kirchhof J, Wagner C, Krieg F, Römer F, Osman A, Del Galdo G. Defect Detection from 3D Ultrasonic Measurements Using Matrix-free Sparse Recovery Algorithms. In: Proceedings of the 26th European signal processing conference. Rome, Italy; 2018, http://dx.doi.org/10.23919/EUSIPCO.2018.8553074.

[16] Semper S, Kirchhof J, Wagner C, Krieg F, Römer F, Del Galdo G. Defect Detection From Compressed 3-D Ultrasonic Frequency Measurements. In: 2019 27th European signal processing conference. p. 1–5. http://dx.doi.org/10.23919/EUSIPCO.2019.8903133.

[17] Wagner CW, Semper S, Römer F, Schönfeld A, Del Galdo G. Hardware Architecture for Ultra-Wideband Channel Impulse Response Measurements Using Compressed Sensing. In: 2020 28th European signal processing conference. 2021, p. 1663–7. http://dx.doi.org/10.23919/Eusipco47968.2020.9287454.

[18] Yadav M, Sheldon D, Musco C. Faster Kernel Interpolation for Gaussian Processes. In: Banerjee A, Fukumizu K, editors. Proceedings of the 24th international conference on artificial intelligence and statistics. Proceedings of machine learning research, vol. 130, PMLR; 2021, p. 2971–9, URL https://proceedings.mlr.press/v130/yadav21a.html.

[19] Weise K, Poßner L, Müller E, Gast R, Knösche TR. Pygpc: A sensitivity and uncertainty analysis toolbox for Python. SoftwareX 2020;11:100450. http://dx.doi.org/10.1016/j.softx.2020.100450.