Integrating Materials and Manufacturing Innovation (2022) 11:450–466
https://doi.org/10.1007/s40192-022-00273-4

TECHNICAL ARTICLE

Microstructure Characterization and Reconstruction in Python: *MCRpy*

Paul Seibert[1] · Alexander Raßloff[1] · Karl Kalina[1] · Marreddy Ambati[1] · Markus Kästner[1,2,3]

Received: 1 July 2022 / Accepted: 10 August 2022 / Published online: 15 September 2022
© The Author(s) 2022

Abstract

Microstructure characterization and reconstruction (MCR) is an important prerequisite for empowering and accelerating integrated computational materials engineering. Much progress has been made in MCR recently; however, in the absence of a flexible software platform it is difficult to use ideas from other researchers and to develop them further. To address this issue, this work presents *MCRpy* as an easy-to-use, extensible and flexible open-source MCR software platform. *MCRpy* can be used as a program with graphical user interface, as a command line tool and as a Python library. The central idea is that microstructure reconstruction is formulated as a modular and extensible optimization problem. In this way, arbitrary descriptors can be used for characterization and arbitrary loss functions combining arbitrary descriptors can be minimized using arbitrary optimizers for reconstructing random heterogeneous media. With stochastic optimizers, this leads to variations of the well-known Yeong–Torquato algorithm. Furthermore, *MCRpy* features automatic differentiation, enabling the utilization of gradient-based optimizers. In this work, after a brief introduction to the underlying concepts, the capabilities of *MCRpy* are demonstrated by exemplarily applying it to typical MCR tasks. Finally, it is shown how to extend *MCRpy* by defining a new microstructure descriptor and readily using it for reconstruction without additional implementation effort.

Keywords Microstructure · Characterization · Reconstruction · Descriptor · Software · ICME

Introduction

Establishing and inverting process–structure–property (PSP) linkages is a central goal in integrated computational materials engineering (ICME) in order to accelerate the development of new materials. With increasing computational resources and much development in data processing and machine learning, data-centric workflows for microstructure design receive more and more attention [1]. These workflows rely on large databases that are created using numerical simulations. One central aspect to consider in this context is how to choose and create the microstructures to simulate from the extremely big set of possible structures. To avoid extremely time-consuming and cost-intensive experimental campaigns, an efficient microstructure characterization and reconstruction (MCR) tool is therefore a key ingredient to making large-scale ICME workflows feasible. A very brief introduction to MCR is given in the following, and the reader is kindly referred to [2] for an in-depth review.

Microstructure characterization, the first aspect of MCR, is required to handle the stochasticity of the microstructures: Two distinct image sections of the same microstructure are similar from a visual and statistical perspective, but completely different in terms of a pixel-based representation. Thus, for operations like quantitative comparisons, it is reasonable to map the pixel-based microstructure to a translation-invariant, stationary descriptor D that allows for these operations. In practice, D can range from simple volume fractions to advanced statistical descriptors such as spatial correlations. Therefore, D is a reasonable choice for representing structures in PSP linkages. Furthermore, it provides a possibility to explore the microstructure space in data-driven materials development workflows.

Microstructure reconstruction, the second aspect of MCR, can be regarded as the inverse operation to microstructure characterization: The goal is to find a microstructure such that the corresponding descriptor equals the given value.

✉ Markus Kästner
markus.kaestner@tu-dresden.de

1 Institute of Solid Mechanics, TU Dresden, 01062 Dresden, Germany

2 Dresden Center for Computational Materials Science, TU Dresden, 01062 Dresden, Germany

3 Dresden Center for Fatigue and Reliability, TU Dresden, 01062 Dresden, Germany

Microstructure reconstruction allows to *(i)* create a plausible 3D volume element from a 2D slice like a microscopy image, *(ii)* create a set of similar microstructures given one realization and *(iii)* interpolating between microstructures in terms of descriptors.

These two aspects of MCR, namely characterization and reconstruction, can be treated independently, for example using spatial correlations as descriptors and modern machine learning-based techniques for reconstruction. However, automatic ICME workflows for complex materials highly benefit from a principled exploration of the descriptor space, where microstructures are selected for reconstruction, simulation and homogenization in a way that maximizes the expected information gain for the PSP linkage [3]. Therefore, it is important to combine characterization and reconstruction so that given arbitrary combinations of descriptors and their values, the reconstruction can be triggered from these descriptors. Furthermore, recent research indicates that there is no single best descriptor for microstructure reconstruction [4] and for PSP linkages [5], but that it is reasonable to choose descriptors based on the structure at hand. For this purpose, we present *MCRpy*[1], a modular and extensible open-source tool that facilitates easy microstructure characterization and reconstruction based on arbitrary descriptors.

Free open-source platforms are a great way of harnessing the advantages of digitization and modern computational infrastructure. The free accessibility allows researchers to quickly test each others' ideas and to develop them further. The open-source nature of such a platform enables it to become a collaborative project, considerably leveraging its potential. Especially in complex scientific disciplines, such collaboration is indispensable. As an example, consider the field of machine learning, specifically neural networks [6]. In the beginning of research on neural networks, newcomers had to implement relatively complex procedures like backpropagation before being able to reproduce results from the literature, let alone to develop them further. Later, easy-to-use open-source libraries like *TensorFlow* and *PyTorch* have greatly lowered the hurdle, allowing more researchers to enter the field easily. This surely contributed to the rapid progress in the last decades and to the plethora of neural network architectures and applications that is observed today.

The digital infrastructure of the materials science community has grown considerably as a consequence of the materials genome initiative [7] and similar projects. Despite the rapidly growing number of tools for materials innovation in general, MCR specifically is in a comparable position now as machine learning was 20 years ago: A great variety of methods exists, but in the absence of a common platform and interface, every newcomer in the field has to implement fundamental technologies like the lineal path function and the Yeong–Torquato algorithm by hand. This is a big hurdle and thwarts rapid progress. Thus, the goal of this contribution is to accelerate MCR research by providing *MCRpy* as an easy-to-use, extensible and flexible software solution that aims at realizing a seamless workflow by providing various interfaces to new and established techniques.

The work starts with Sect. Current Digital Infrastructure, where the current digital infrastructure is reviewed and it is outlined how *MCRpy* integrates into it. Then, *MCRpy* is presented in Sect. Overview of MCRpy. Typical application workflows are presented in Sect. Typical MCRpy Workflows and finally, a conclusion is drawn in Sect. Conclusions and Outlook.

Current Digital Infrastructure

After President Barack Obama announced the US-American Materials Genome Initiative [7] that provided substantial funding for accelerated materials development, collaborative projects and digital frameworks were initiated all over the world. A non-exhaustive list includes the American *NanoMine* open data resource [8], the European *NOMAD-CoE* [9] and its platform described in [10] and the Swiss *NCCR MARVEL* [11] with its *AiiDA* platform [12] described in [13]. The extremely popular and often-cited *pymatgen* library [14] can be mentioned as an early contribution to open-source materials science software infrastructure. This trend continues, as can be seen with the recent example *radonpy* [15]. However, much of this research is focused on deriving material properties from considerations on the atomistic length scale.

On the continuum length scale, the *Python Materials Knowledge System (pyMKS)* [16] is a notable open-source framework. Its efficient FFT-based implementation of the spatial two-point correlation S_2 facilitates easy microstructure characterization. However, in *pyMKS*, microstructure characterization is limited to S_2 and no further descriptors are available. Moreover, *pyMKS* does not allow for microstructure reconstruction, only characterization. A strong focus lies on efficient homogenization [17] and direct coupling to an internal finite element solver, *SfePy* [18, 19]. This is very convenient for simple problems like elasticity. For advanced techniques like crystal plasticity, external software like the Düsseldorf Advanced Materials Simulation Kit (*DAMASK*) [20] can be used. Furthermore, *pyMKS* provides an easy interface for dimensionality reduction of the descriptor space and for establishing structure-property linkages based on the reduced descriptors and the corresponding homogenized properties. In summary, *pyMKS* acts as an overarching framework to implement ICME workflows.

[1] https://github.com/NEFM-TUDresden/MCRpy

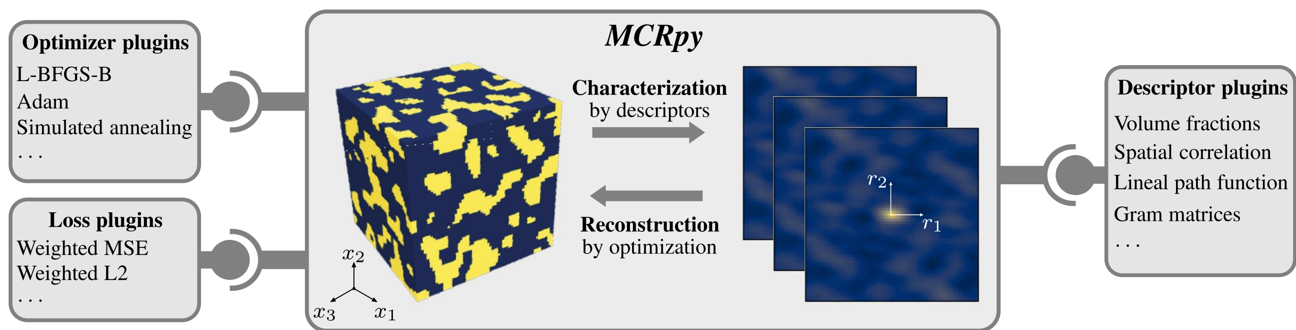


Fig. 1 Schematic overview of *MCRpy*: Microstructures can be characterized by descriptors and reconstructed by optimization. Herein, descriptors, losses and optimizers can be provided as flexible plugin modules

For numerical simulation of microstructures, many open-source tools are available, ranging from general and easy-to-use packages like *SfePy* [18, 19] to special-purpose software like *DAMASK* [20], which comes with a highly optimized Fourier-based crystal plasticity solver. Furthermore, current research on FFT-based homogenization [21] is making remarkable progress that might lead to an open-source tool soon. Thus, with *pyMKS* as an overarching framework and numerous tools and progress for numerical simulation and homogenization, an open-source MCR software package can be identified as a final component of ICME workflows.

To the authors' best knowledge, the only widely used software tool for MCR is *DREAM.3D* [22], a long developed and full-fledged program. Its roots date back around 20 years to the early works of Michael Groeber and the Carnegie Mellon University microstructure builder. Despite this long history, *DREAM.3D* still enables numerous current research activities in materials innovation and ICME, see for example [23]. This success is empowered by the many features, robustness, efficiency and easy user interface of *DREAM.3D*, which may be partially attributable to its open-source core. Thus, *DREAM.3D* can be highly recommended for the workflows it implements. However, the internal microstructure representation and the available pipelines in *DREAM.3D* are mainly intended for certain material systems and microstructure descriptors. The internal data format as well as the provided characterization and reconstruction algorithms are centered around classical descriptors like grain size distribution functions and orientation distribution functions. This makes *DREAM.3D* excellent at reconstructing geometric inclusions like ellipses and texture as in metallic materials, but multi-phase materials with complex morphology as shown in Fig. 8 cannot be realized. Furthermore, *DREAM.3D* is written in C++, which is not common among engineering researchers due to its complexity. In recent research, new microstructure descriptors or reconstruction algorithms are sometimes provided as Python or Matlab code in a GitHub repository, but are hardly ever implemented in C++ as a *DREAM.3D* pipeline[2]. Even if that was the case, then these descriptors could not be readily used for reconstruction since the *DREAM.3D* reconstruction pipelines are tailored toward specific descriptors and would need to be re-implemented. Thus, *DREAM.3D* is an excellent and robust program, but it is mainly suited for specific practical applications and for certain materials.

In contrast, the present work aims at creating a flexible research platform for multiphase materials of high morphological complexity. Thus, *MCRpy* clearly differs from *DREAM.3D* regarding the targeted audience and the scope of materials systems. As a Python package, it integrates naturally with numerous tools for numerical simulation, machine learning or materials science workflows. Especially *pyMKS* can act as an overarching ICME framework, where the present work provides an MCR solution. In summary, *MCRpy* attempts to fill a striking gap in the ICME software landscape. A theoretical understanding of *MCRpy* is provided in Sect. Overview of MCRpy, followed by an illustration of typical workflows in Sect. Typical MCRpy Workflows.

Overview of MCRpy

Microstructure characterization and reconstruction in Python (*MCRpy*) is an open-source software tool accessible under https://github.com/NEFM-TUDresden/MCRpy. It is released under the *Apache 2.0* license and can be used

(i) as a program with graphical user interface (GUI), intended for non-programmers and as an easy introduction to MCR,

[2] Note, however, ongoing developments with the eventual goal of allowing *DREAM.3D* pipelines to be coded purely in Python: https://github.com/BlueQuartzSoftware/dream3d-conda-feedstock.

Table 1 Functions

Function	Explanation
`characterize`	characterize a microstructure, see Sect. Characterization
`reconstruct`	reconstruct a microstructure given the descriptors, see Sect. Reconstruction
`match`	characterize and reconstruct immediately, for validation and for 2D-to-3D workflows
`view`	plot microstructures, descriptors and convergence data interactively or save to a file
`smooth`	smooth a microstructure
`merge`	merge different descriptors to prescribe them on orthogonal sections for reconstructing anisotropic structures
`interpolate`	interpolate between given descriptors

(ii) as a command line tool, intended for automated and large-scale application on high-performance computers without graphical interface, and
(iii) as a regular PIP-installable Python module, intended for performing advanced and custom operations in the descriptor space.

A schematic overview is given in Fig. 1: The main functionalities of *MCRpy*, characterization and reconstruction, are explained in Sect. Characterization and Reconstruction, respectively. Furthermore, additional functions are provided to manipulate the microstructures and descriptors and to visualize data. A complete set of the available operations is given in Table 1, and supported inputs and outputs for selected functions are summarized in Table 2. The core idea of *MCRpy* is its extensibility in that arbitrary descriptors can be used for characterization and arbitrary loss functions combining arbitrary descriptors can be minimized using arbitrary optimizers for reconstructing random heterogeneous media. This is outlined in Sect. Extensibility.

Characterization

The characterization function

$$f_C : M \mapsto \{D_i\}_{i=1}^{n_D} \quad (1)$$

assigns a given pixel-based microstructure M to a set of n_D corresponding descriptors D_i. These descriptors, sometimes referred to as statistical descriptors, quantify the microstructural morphology in a statistical and translation-invariant manner[3]. Hereby, a microstructure with n_p different phases is represented as a set of n_p indicator functions

$$I_p(x) = \begin{cases} 1, & \text{if } x \text{ in phase } p \\ 0, & \text{else.} \end{cases} \quad (2)$$

For example, the volume fraction v_f of a microstructure is a very simple descriptor. Of course, the volume fraction captures some but not all information needed to describe the microstructure. Several other quantities matter, for example the size and shape of inclusions and the degree to which distinct phases, are spatially clustered. Besides these classical descriptors, in the light of increasing computational resources, recent research has been focused on more universal high-dimensional descriptors that are less dense in information, but have higher descriptive capabilities in total. As an early example for high-dimensional descriptors, spatial correlations [24] have proven to be a versatile tool that is still used today [2]. A good introduction can be found in [25]. A differentiable generalization of spatial correlation is presented in [26] and used in this work. Spatial correlations have inspired a range of conceptually similar descriptors like the lineal path function [27], cluster correlation function [28] and polytope function [29]. The reader is referred to [2] for a comprehensive overview. Finally, the Gram matrices of the feature maps of pre-trained convolutional neural networks have been shown to contain relevant microstructural information [30]. Remarkable results in microstructure reconstruction have been achieved using

Table 2 Possible inputs and outputs for selected *MCRpy* functions

Function	Input		Output
`Characterize`	2D M	→	$\{D_i\}$
	3D M	→	$\{D_i\}$
	3D M	→	$\{D_i^x\ D_i^y\ D_i^z\}$
`merge`	2 or 3 orthogonal $\{D_i\}$	→	$\{D_i^x\ D_i^y\ D_i^z\}$
`Reconstruct`	$\{D_i\}$	→	2D M
	$\{D_i\}$	→	3D M
	$\{D_i^x\ D_i^y\ D_i^z\}$	→	3D M
	2D M	→	2D M
`Match`	2D M	→	3D M
	3D M	→	3D M

[3] While some descriptors, such as the volume fractions, are also rotation-invariant, this property does not apply in the general case. For example, the spatial correlations in *MCRpy* are not rotation-invariant.

Table 3 Microstructure descriptors that are implemented in *MCRpy*

	Descriptor	Differentiable	Comment
v_f	`VolumeFractions`	✓	-
$\tilde{S}$	`Correlations`	✓	$\tilde{S}_2$ and $\tilde{S}_3$; see [26]
S	`FFTCorrelations`	✗	only S_2; FFT-based; from *pyMKS* [16]
G	`GramMatrices`	✓	using VGG19 [35]; see [31]
$\mathcal{V}$	`Variation`	✓	normalized total variation; see [4]
$\tilde{L}$	`LinealPath`	✓	see Appendix A

such Gram matrices alone [31, 32] and in combination with other descriptors [4, 33].

Finding a microstructure description that is both dense and contains all relevant information is an active field of research [2]. Examples are the recently developed entropic descriptors [34] or polytope functions [29]. Thus, besides the currently available descriptors listed in Table 3, users can add a descriptor plugin to *MCRpy*. If the descriptor plugin is defined with an indicator function as input, it is applied to the indicator function of each phase separately. Furthermore, a 2D descriptor is automatically applied on and averaged over 2D slices of a 3D structure. The only requirement posed on new descriptors is that they must be computable on a pixel or voxel geometry. More details on extensibility can be found in Sect. Extensibility. All of the available and added descriptors can be used for microstructure reconstruction, which is discussed in the following section.

Reconstruction

In *MCRpy*, microstructure reconstruction is fundamentally regarded as an optimization problem

$$M^{\text{rec}} = \underset{M}{\text{argmin}}\ \mathcal{L}\left(\{(D_i(M),\ D_i^{\text{des}})\}_{i=1}^{n_D}\right) \quad , \tag{3}$$

where the reconstructed microstructure M^{rec} minimizes a loss function $\mathcal{L}$. The loss function depends on n_D different descriptors $\{D_i\}_{i=1}^{n_D}$ and quantifies the distance between their current and desired values. Herein, $D_i(M)$ denotes the value of the *i-th* descriptor associated with the current microstructure and its desired value D_i^{des}. Naturally, as in the characterization step, arbitrary descriptors can be used, for example the volume fractions v_f, the spatial correlations S or the Gram matrices G. For the loss function $\mathcal{L}$, a simple choice is a weighted sum over the mean squared error norm. Different loss functions are available in *MCRpy* and the user can implement additional ones. Finally, given a set of descriptors and a loss function, an optimization problem emerges as a special case of Equation 3. This optimization problem can be solved using an optimizer, which is provided as a plugin module. If all descriptors are differentiable, then a gradient-based optimizer like L-BFGS-B [36] can be used, leading to the very efficient *differentiable MCR* [4, 26]. Otherwise, the choice is limited to gradient-free optimizers like simulated annealing.

As a simple example, if only the spatial two-point correlation S_2 is used as a descriptor and the loss function is formulated as a mean squared error norm of the descriptor difference, the following optimization problem emerges:

$$M^{\text{rec}} = \underset{M}{\text{argmin}}\ ||S_2(M) - S_2^{\text{des}}||_{\text{MSE}} \quad . \tag{4}$$

If simulated annealing is chosen as an optimizer, *MCRpy* effectively performs the well-known Yeong–Torquato algorithm as used in [37].

As a more recent example, if the Gram matrices G of the feature maps of the VGG-19 convolutional neural network are chosen as a descriptor [30] for the same loss function, the emerging optimization problem

$$M^{\text{rec}} = \underset{M}{\text{argmin}}\ ||G(M) - G^{\text{des}}||_{\text{MSE}} \tag{5}$$

allows for a gradient-based optimizer. If L-BFGS-B [36] is chosen for this purpose, *MCRpy* effectively performs the approach of Li et al. [31], which is a special case of differentiable MCR [26].

As a final example, the differentiable three-point correlations S_3, the above-mentioned Gram matrices G and the normalized total variation $\mathcal{V}$ are combined. The loss function accumulates the weighted mean squared error norm, where λ_{D_i} denotes the weight of the *i-th* descriptor. If the resulting optimization problem

$$\begin{aligned} M^{\text{rec}} = \underset{M}{\text{argmin}}\ & \lambda_S||S_3(M) - S_3^{\text{des}}||_{\text{MSE}} \\ & +\lambda_G||G(M) - G^{\text{des}}||_{\text{MSE}} \\ & +\lambda_{\mathcal{V}}||\mathcal{V}(M) - \mathcal{V}^{\text{des}}||_{\text{MSE}} \end{aligned} \tag{6}$$

is solved using the gradient-based L-BFGS-B optimizer, *MCRpy* effectively performs the differentiable MCR algorithm as used in [4].

As can be seen, different parameter settings allow to recreate well-known reconstruction algorithms as well as to try out new ones by simply changing the arguments. As an overview, all descriptors, optimizers and loss functions are listed in Table 4.

Table 4 Microstructure descriptors, optimizers and loss functions that are implemented in *MCRpy*. Simulated annealing is the only optimizer in the list that is not gradient-based. More details on the descriptors are given in Table 3.

Descriptors	Optimizers	Loss functions
Volume fractions	L-BFGS-B [36]	2D/3D weighted MSE
Correlations	TNC [38]	2D/3D weighted RMS error
Lineal path	Adam [39], Adagrad [40], Adadelta [41]	2D/3D weighted L1 distance
Gram matrices	RMSprop [42]	2D/3D weighted L2 distance
Variation	SGD [42]	
	Simulated Annealing [24]	

Extensibility

The central advantage of *MCRpy* is its extensibility in that descriptors, loss functions and optimizers can be easily provided by anyone. For example, new optimization-based reconstruction algorithms like the work of Cecen at al. [43] can be implemented as an optimizer plugin to combine them with all the available microstructure descriptors. This is achieved by a plugin architecture, which is sketched in Fig. 2. In this section, we explain the underlying software pattern, whereas exact instructions and an example on how to write a plugin are given in Sect. Defining a custom descriptor. In the following, the plugin architecture is explained for the case of descriptors. The same idea is employed for loss functions and optimizers.

A descriptor plugin can be written by simply inheriting from the abstract `Descriptor` class. Consequently, the available descriptor plugins are not known at the time of writing the *MCRpy* core code, so they must be loaded dynamically as soon as the characterization or reconstruction module demands the plugin. This is done by means of a loader module based on `importlib`. Upon import, a descriptor plugin registers itself at a descriptor factory. After that, the descriptor factory can be queried to create descriptor instances from the plugin. The descriptor factory then returns a callable which computes the descriptor value given a microstructure. This callable can now be used to characterize microstructures, compose loss functions, compute gradients using automatic differentiation and to reconstruct microstructures.

Thus, adding a descriptor plugin to *MCRpy* merely consists of adding a file with the plugin definition to the right directory, while the rest of the code does not need to be changed. The descriptor immediately becomes available for characterization and for reconstruction in combination with arbitrary other descriptors, arbitrary loss functions and arbitrary optimizers.

Typical MCRpy Workflows

Typical use-cases and workflows of *MCRpy* are illustrated in this section by means of three representative examples. First, in Sect. Obtaining a 3D domain from a 2D microstructure slice, a plausible 3D volume element is reconstructed from a 2D microstructure slice. This very relevant, since 3D information can be very time- and cost-intensive to obtain experimentally. Secondly, in Sect. Obtaining a set of similar volume elements, a statistically similar set of small volume elements is generated from a single example. This greatly reduces the computational effort for numerical homogenization. Thirdly, in Sect. Manipulating the descriptor space, descriptor values are directly manipulated and used for reconstructing novel structures. Techniques like this may be explored in the future to augment data sets and explore PSP linkages. These three examples are demonstrated in the three modes of operating *MCRpy*, namely via a GUI, as a command line tool and as a Python library, respectively. Note that this order is chosen for demonstration purposes only and it is possible to execute all three workflows with all three modes of operation. Finally, in Sect. Defining a custom descriptor, it is demonstrated how to add a custom descriptor to *MCRpy* and how to use it for characterization and reconstruction. The original structures are taken from *pyMKS* [16] for Sect. Obtaining a 3D domain from a 2D microstructure slice to Manipulating the descriptor space and from [31] for Sect. Defining a custom descriptor.

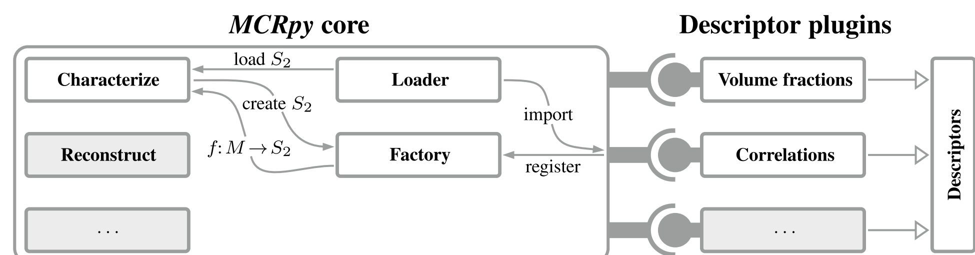


Fig. 2 Schematic overview of the plugin architecture in *MCRpy*

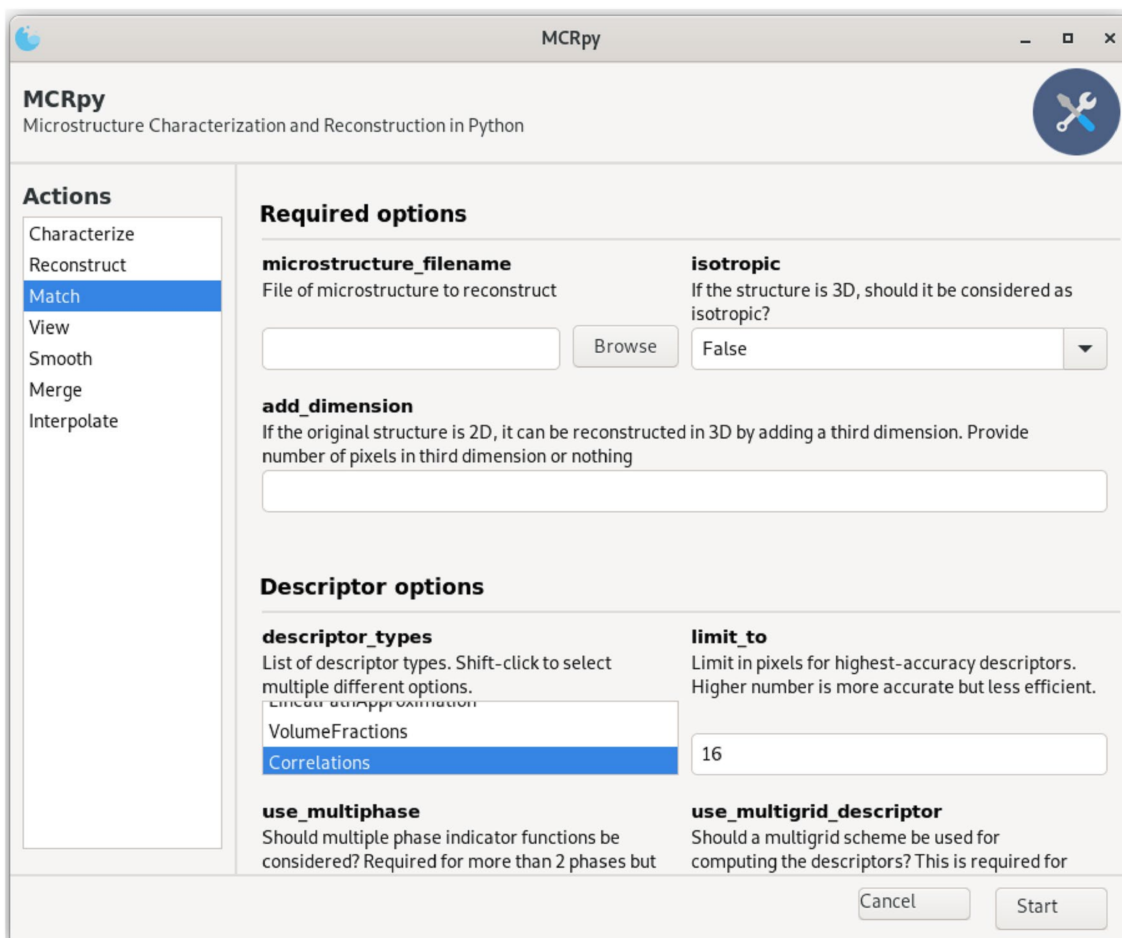
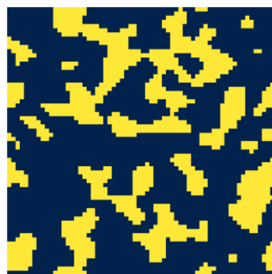


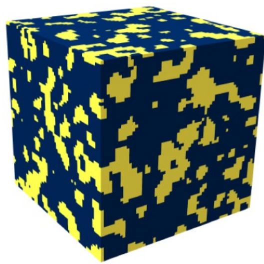
Fig. 3 Screenshot of the *MCRpy* graphical user interface. After selection an action on the left, all options can be set in the center and performed upon clicking *start*. The options are identical to the command line interface and the Python library

Obtaining a 3D Domain from a 2D Microstructure Slice

As a first example, *MCRpy* is used to reconstruct a plausible 3D volume element given a segmented 2D slice. This is a common task since experimental observations are often available only in 2D. The 3D volume element can be used for example for numerical simulations. From an algorithmic perspective, this goal is achieved by computing the descriptor on the given slice and prescribing it on every slice of the microstructure, details cf. [4].



(a) Original slice



(b) Reconstructed structure

Fig. 4 Results for the example in Section Obtaining a 3D domain from a 2D microstructure slice

This task is solved using the *MCRpy* GUI as shown in Fig. 3. A simple approach would be characterization and immediate reconstruction, but as mentioned in Table 1, *MCRpy* provides a shortcut for this in the `match` function. After selecting the *match*-action on the left, the relevant options can be set in the center. The name of each option is identical to the command line and the Python library, allowing users to easily switch interfaces. By default, a 2D structure is reconstructed in 2D. However, by using the option `add_dimension`, the extent of the reconstructed structure in z-direction is set to the desired value. The differentiable three-point correlations $\tilde{S}_3$ as proposed in [26] are chosen as descriptor. Furthermore, as discussed in [4], the variation $\mathcal{V}$ is employed as a descriptor in order to suppress noise in the 3D reconstruction. The weights of $\tilde{S}_3$ and $\mathcal{V}$ are empirically

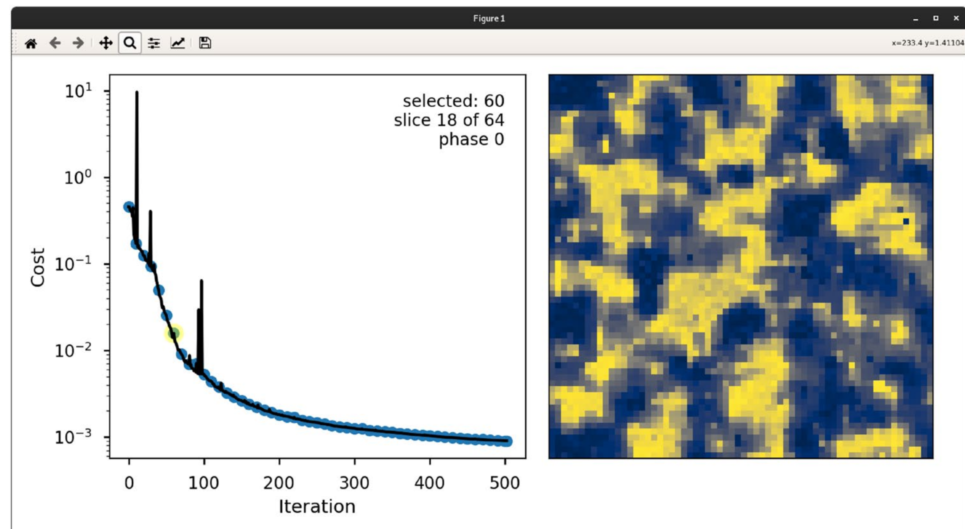


Fig. 5 Interactive window for inspecting convergence data. By selecting the highlighted dot on the left at iteration 60, a slice of the intermediate result is displayed on the right. In this example, the indicator function of phase 1 is displayed for slice 18 of 64

set to 1 and 100, respectively[4]. Finally, the role of the setting `limit_to` needs to be discussed. The parameter is introduced in [26] as P and Q and quantifies the length in pixels up to which spatial correlations are computed with the highest-possible precision. All longer-ranged correlations are computed on a lower-resolution version of the structure in order to save computational resources, cf. [26]. With a default of 16, it allows a flexible trade-off between accuracy and efficiency. A quantitative analysis of wallclock time and memory requirements is given in Appendix D. In this example, it is lowered to 8 in order to accelerate the computations.

After setting all options, the reconstruction can be started and the results can be viewed from the GUI by selecting the *view*-action on the left. 2D microstructures are plotted directly, whereas 3D structures are exported to and opened in *ParaView* [44]. The original 2D slice and the reconstructed 3D volume are shown in Fig. 4. Note that for 2D-to-3D reconstruction using multiple orthogonal 2D slices, an additional descriptor merging step is required as discussed in Sect. Manipulating the descriptor space and carried out in Appendix C.

In addition to the final microstructure, a convergence data file is written, which can be viewed interactively with *MCRpy* as shown in Fig. 5. On the left, the loss is plotted over iterations along with blue dots indicating intermediate results. The user can click on these dots to have the corresponding microstructure displayed on the right. For 3D structures, only one slice is plotted and the user can scroll through the microstructure using the mouse wheel. For displaying the raw phase indicator functions of multiphase structures and other functionalities, the user is referred to the documentation. In summary, the *MCRpy* GUI constitutes an easily accessible solution for microstructure reconstruction.

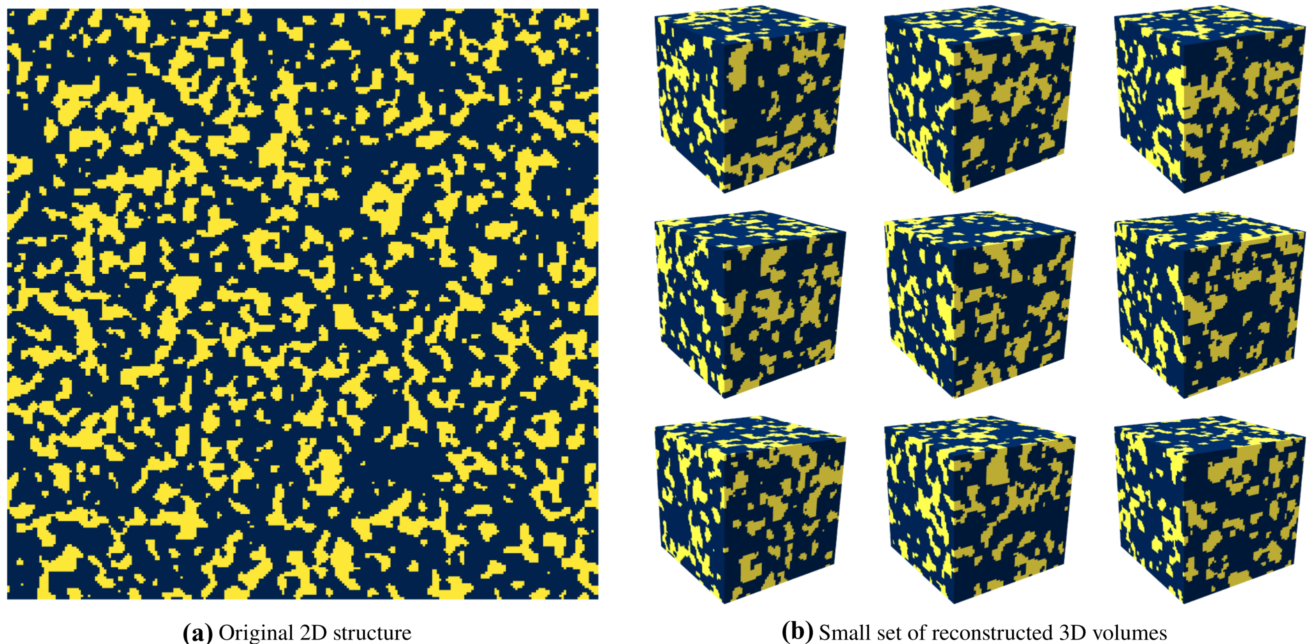
Obtaining a Set of Similar Volume Elements

As a second example, a statistically similar set of volume elements is created from a single microstructure example. In numerical homogenization, a volume element can only be called representative if it is large enough for the stochasticity of the microstructure to have no effect on the effective properties. In practice, this requirement can imply unfeasible computational effort. If smaller volume elements are used, it is still possible to quantify the effective behavior by using sufficiently many smaller volume elements and statistically aggregating the results. An example for structure-property linkages based on this idea can be found in [45]. From an MCR perspective, this requires characterizing the given structure and reconstructing different random realizations from it[5].

This task is solved using *MCRpy* as a command line tool as shown in Listing 1. First, the original 2D

[4] The role of the weights is discussed in [4]. If the weight of the variation is too small, the noise is not suppressed well enough. If it is too large, the optimization problem becomes harder to solve and more iterations are needed for convergence.

[5] In [26], the reconstruction was shown to converge to the exact same microstructure that was used for the characterization. The same can be seen for some cases in Fig. 8. However, this only happens in 2D reconstruction (not in 3D) and only for certain descriptors and microstructures. In these cases, the desired descriptor prescribed for reconstruction needs to be varied statistically in order to create a diverse set of microstructure realizations. This aspect is not considered in the following because in application, it is most useful to reconstruct 3D structures.



(a) Original 2D structure

(b) Small set of reconstructed 3D volumes

Fig. 6 Input and results generated from the code in Listing 1

microstructure stored as `ms_slice.npy` is characterized using the same parameters as in Sect. Obtaining a 3D domain from a 2D microstructure slice (line 1). Then, nine different 3D structures are generated by a simple loop over the reconstruction script (lines 2-7). Note that the extent of the reconstructed structures in voxels is set independently of the original slice (line 5). Furthermore, the loop index is passed to the reconstruction script in order to have it added to all result filenames and prevent to overwrite previous results (line 5). Because the chosen descriptors are differentiable, the standard optimizer L-BFGS-B [36] can be used, allowing to harness the computational efficiency of DMCR [4, 26]. On an *Nvidia A100* GPU, the reconstructions take around 25 minutes per structure for 500 iterations. The original structure and the results can be seen in Fig. 6. In summary, the command line interface is analogous to the GUI and allows for easy automation and large-scale application.

Listing 1: Simple bash script for automating reconstruction using the *MCRpy* command line interface. The results are shown in Figure 6

```
python characterize.py ms_slice.npy --limit_to 8 --descriptor_types Correlations Variation
for i in {1..9}
do
    python reconstruct.py --descriptor_filename results/ms_slice_characterization.pickle \
        --extent_x 64 --extent_y 64 --extent_z 64 --limit_to 8 --information ${i} \
        --descriptor_types Correlations Variation --descriptor_weights 1 100
done
```

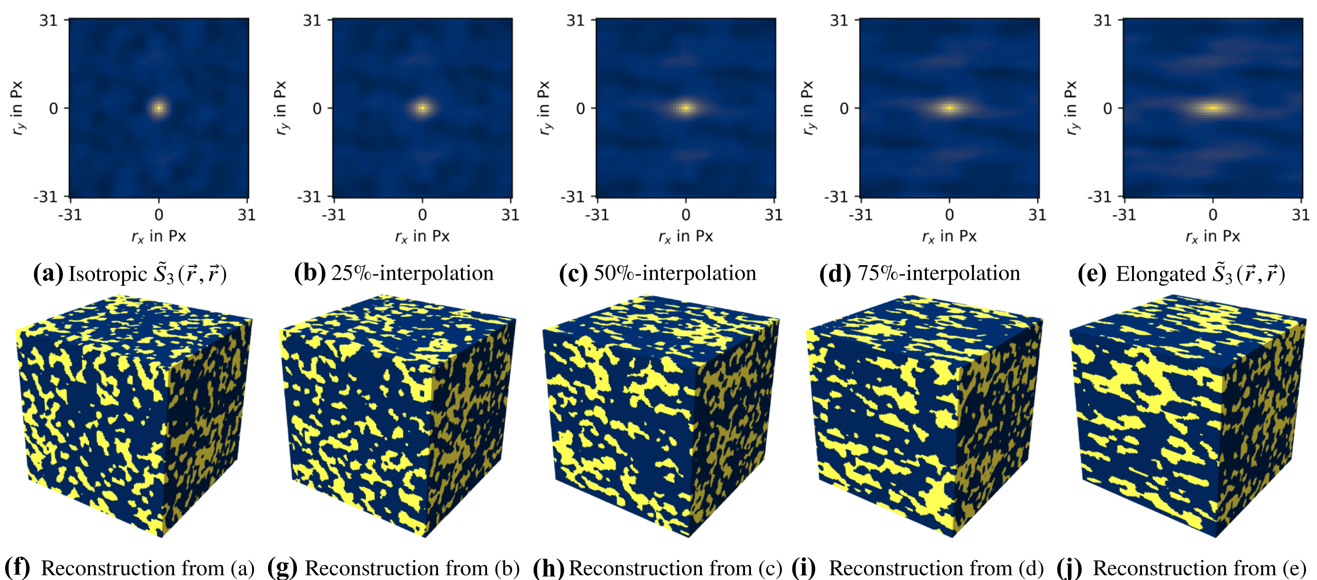


Fig. 7 The original descriptors (**a**, **e**) are linearly interpolated (**b-d**) and used for reconstruction to create a smooth transition between an isotropic and an elongated microstructure (**f-j**). For the descriptor $\tilde{S}_3(\mathbf{r}_a, \mathbf{r}_b)$, only the case that $\mathbf{r}_a = \mathbf{r}_b = \mathbf{r}$ is plotted for clarity

Manipulating the Descriptor Space

As a third example, the explicit availability of the descriptor is exploited by directly manipulating it. Specifically, *MCRpy* is used to interpolate between two given microstructures in a morphologically meaningful way. Consider two microstructures that could stem from different sets of process parameters. It can be interesting to create a morphology that is a mix between these two structures. For example, if numerical simulations and homogenization of the interpolated structure predict favorable effective properties, it might be worth to fine-tune the process parameters or try to establish a PSP linkage to manufacture these structures. Direct interpolation of the microstructures in terms of pixel values is meaningless. As a simple alternative, we interpolate linearly in the descriptor space and reconstruct microstructures from the interpolated descriptors.

This task is solved using *MCRpy* as a Python library as shown in Listing 2. After defining the settings (lines 4-10), the original 2D microstructure slices are loaded (lines 13-14) and characterized (lines 17-18). For reconstructing elongated 3D structures, the 2D descriptors need to be combined such that different descriptors are used in different directions. The order thereby matters and mistakes can lead to geometrically unrealizable descriptors[6]. In order to avoid confusion and mistakes, *MCRpy* provides the function `merge` for this task. The merged descriptors (lines 21-22) are then interpolated in 5 steps including start and end (line 25). Each descriptor is used for a 3D reconstruction, which returns the convergence data and the final microstructure (line 29). The convergence data is viewed in an interactive window as shown in Fig. 5 (line 31). Finally, the microstructures are smoothed by a Gaussian filter (line 32) and saved to a file (line 33). The results are shown in Fig. 7. It can be confirmed that linear interpolation in the descriptor space leads to a visually reasonable transition between the corresponding microstructures.

[6] For example, consider the three planes $x-y$, $x-z$ and $y-z$. Structures that are elongated in x-direction can be created by prescribing horizontally elongated descriptors in planes 1 and 2 and an isotropic descriptor in plane 3. However, if the same horizontally elongated descriptors are prescribed in planes 1 and 3, the structure cannot be realized: Plane 1 requires elongations in the y-direction, whereas plane 3 requires the elongations to be in the z-direction and not in y.

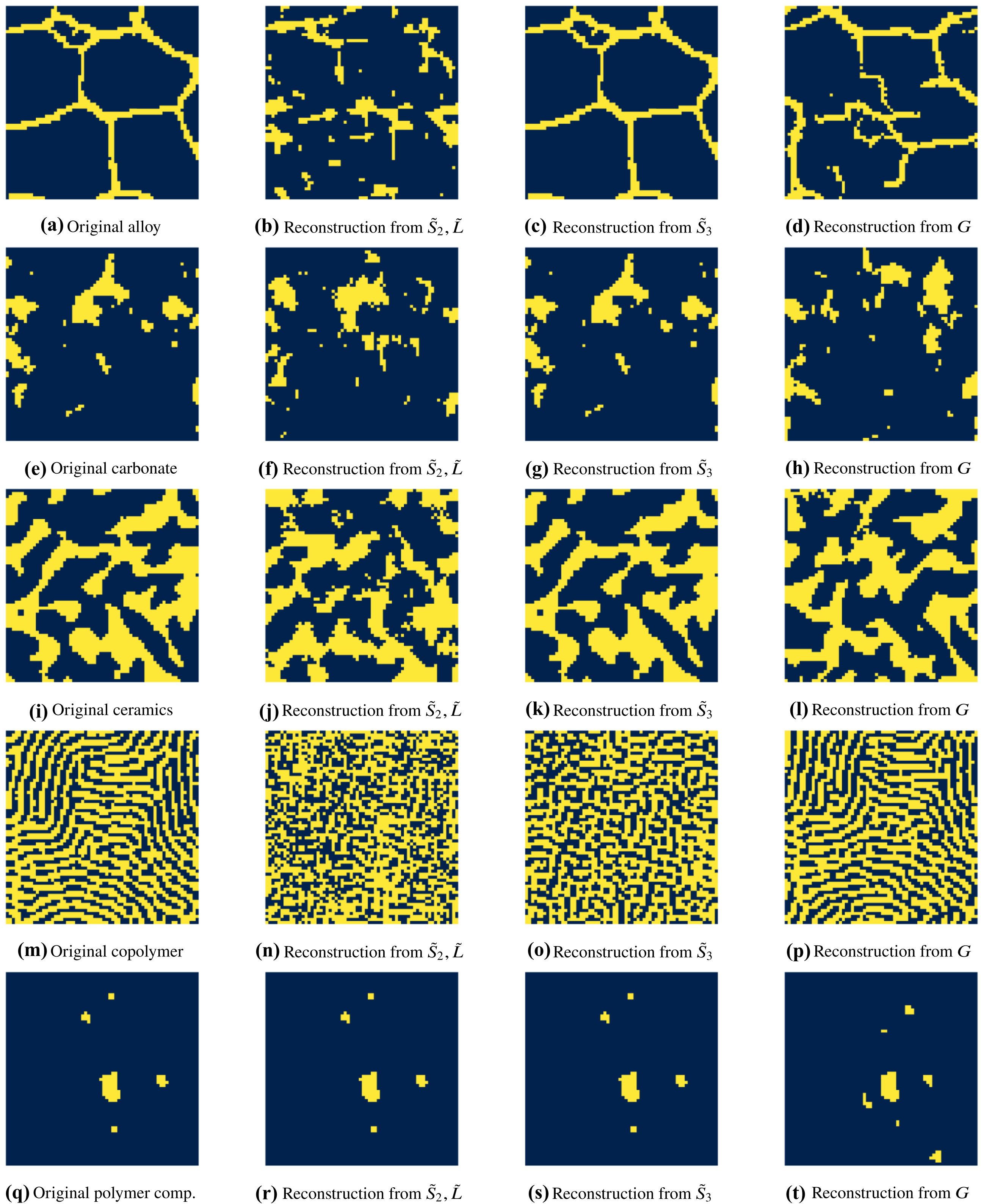


Fig. 8 Comparison between original microstructures and reconstruction results from different descriptors. For a clearer visualization, the reconstructed microstructures are shifted periodically to match the original structure as closely as possible

Listing 2: Simple Python script that uses *MCRpy* to characterize two microstructure slices, interpolate between them in the descriptor space and reconstruct the corresponding 3D structures.

```
import mcrpy

# define settings
limit_to = 8
descriptor_types = ['Correlations', 'Variation']
descriptor_weights = [1.0, 10.0]
characterization_settings = mcrpy.CharacterizationSettings(descriptor_types=descriptor_types,
   limit_to=limit_to)
reconstruction_settings = mcrpy.ReconstructionSettings(descriptor_types=descriptor_types,
   descriptor_weights=descriptor_weights, limit_to=limit_to, use_multigrid_reconstruction=True)

# load microstructures
ms_from = mcrpy.load('microstructures/ms_slice_isotropic.npy')
ms_to = mcrpy.load('microstructures/ms_slice_elongated.npy')

# characterize microstructures
descriptor_isotropic = mcrpy.characterize(ms_from, characterization_settings)
descriptor_elongated = mcrpy.characterize(ms_to, characterization_settings)

# merge descriptors
descriptor_from = mcrpy.merge([descriptor_isotropic])
descriptor_to = mcrpy.merge([descriptor_elongated, descriptor_isotropic])

# interpolate in descriptor space
d_inter = mcrpy.interpolate(descriptor_from, descriptor_to, 5)

# reconstruct from interpolated descriptors and save results
for i, interpolated_descriptor in enumerate(d_inter):
   convergence_data, ms = mcrpy.reconstruct(interpolated_descriptor, (128, 128, 128),
      settings=reconstruction_settings)
   mcrpy.view(convergence_data)
   smoothed_ms = mcrpy.smooth(ms)
   mcrpy.save_microstructure(f'ms_interpolated_{i}.npy', smoothed_ms)
```

Defining a Custom Descriptor

MCRpy can be easily extended by adding custom plugin modules. In this section, the procedure is demonstrated by means of a descriptor plugin. Similar concepts apply to loss functions and optimizers. First, the implementation of a descriptor plugin is discussed for the volume fraction v_{f}. Secondly, a differentiable approximation to the lineal path function is developed and tested.

Listing 3 shows the plugin source code for the volume fraction v_{f}. Like all descriptors in *MCRpy*, the volume fraction must inherit from the abstract `Descriptor` class (line 5). This base class provides

(i) a wrapper that applies descriptors defined for single phases to the indicator function of each phase,
(ii) a wrapper to compute multigrid descriptors as discussed in [26] and
(iii) default functions for visualizing descriptors[7].

In this case, it is reasonable to define the descriptor for a single phase and let the superclass handle the generalization to multiple phases. For this purpose, the subclass function `make_singlephase_descriptor` is defined (line 9). This function receives information about the microstructure, like the resolution, which is not needed in this case and therefore summarized via `**kwargs`. It returns a `callable` which computes the descriptor given the indicator function of a phase (line 12). In order to allow for automatic differentiation of the descriptor with respect to the microstructure, this `callable` needs to be implemented in *TensorFlow*. In contrast, a non-differentiable descriptor would be implemented in *Numpy* and integrated into the computation graph by *MCRpy* using the *TensorFlow* function

[7] By default, low-dimensional descriptors are visualized via bar plots and high-dimensional descriptors are reshaped to an approximately quadratic array and plotted as a heatmap. This behavior can be overwritten for each descriptor subclass separately.

`tf.py_function`. Finally, the plugin is required to register itself at the descriptor factory using its class name (lines 16-17).

In the following, the same procedure is applied to a differentiable approximation $\tilde{L}$ to the lineal path function L, which is developed in Appendix A. Naturally, the code for defining $\tilde{L}$ is much longer than Listing 3 and is not given in this paper. Instead, the reader is referred to the *GitHub* repository for viewing the code. After adding the descriptor definition to the `mcrpy/descriptors` directory, it is accessible for characterization and reconstruction via the *MCRpy* GUI, the command line interface and the Python library.

Listing 3: Simple definition of a descriptor plugin for computing the volume fraction v_{f}. This descriptor is differentiable with respect to each pixel in the microstructure, so it is implemented in *TensorFlow* to allow for automatic differentiation. Non-differentiable descriptors can be implemented in *Numpy*.

```
import tensorflow as tf
from mcrpy.src import descriptor_factory
from mcrpy.descriptors.Descriptor import Descriptor

class VolumeFractions(Descriptor):
    is_differentiable = True

    @staticmethod
    def make_singlephase_descriptor(**kwargs) -> callable:

        @tf.function
        def compute_descriptor(indicator_function: tf.Tensor) -> tf.Tensor:
            return tf.math.reduce_mean(indicator_function)
        return compute_descriptor

def register() -> None:
    descriptor_factory.register("VolumeFractions", VolumeFractions)
```

In the Yeong–Torquato algorithm, the lineal path function is often employed to compensate for the shortcomings of the two-point correlation S_2 alone [2, 24]. As an alternative approach to enriching S_2, the differentiable three-point correlations $\tilde{S}_3$ are used in [26]. Furthermore, Gram matrices G have become a common descriptor recently [4, 31–33]. In order to determine a best-practice for gradient-based reconstruction, $\tilde{S}_3$ is compared to G and a combination of $\tilde{S}_2$ and $\tilde{L}$ in Fig. 8. It can be seen that $\tilde{S}_3$ yields perfect reconstructions except for the copolymer, which can only be reconstructed well from G. In contrast, G yields acceptable results for all structures. The combination of $\tilde{S}_2$ and $\tilde{L}$ performs very poorly for the alloy and the copolymer and is relatively noisy for the carbonate and ceramics. However, it outperforms G for the polymer composite. In summary, the results in Fig. 8 indicate that including higher-order information to $\tilde{S}_2$ via $\tilde{S}_3$ is more promising for gradient-based reconstruction than via the newly proposed differentiable approximation to the lineal path function $\tilde{L}$.

The more relevant aspect, however, is how easily new descriptors can be assessed using *MCRpy*. After defining a plugin as shown in Listing 3, it can be used for characterization and reconstruction and evaluated seamlessly. This extensibility facilitates quick and easy experimentation, allowing researchers to assess new MCR techniques easily and provide them to their colleagues.

Conclusions and Outlook

MCRpy is a powerful and extensible open-source Python library and toolkit for microstructure characterization and reconstruction. Besides these core features, *MCRpy* provides a plethora of convenient tools for inspecting and comparing descriptors, analyzing reconstruction results and controlling the descriptor space. It is easily applied via a GUI and brought to automated large-scale application on high-performance computers through a command line interface. For advanced and custom operations in the descriptor space, *MCRpy* can be imported and used as a Python module with direct access to the structures and descriptors. Typical workflows for these interfaces are presented in this work by means of different MCR tasks.

A central design aspect in *MCRpy* is its extensibility in that descriptors, loss functions and optimizers can be provided by the community as simple plugin modules. An example for a simple plugin is given in this work. We hope that the open source nature of the code and the plugin

architecture will make *MCRpy* an international collaborative project with contributions from numerous researchers. This growth can leverage the potential of the presented tool to facilitate faster and easier MCR research and ultimately help accelerating materials development.

Appendix 1: Differentiable Approximation to Lineal Path Function

The lineal path function L is a well-established microstructure descriptor [27] that is often used in the Yeong–Torquato algorithm to compensate for the shortcomings of the two-point correlation S_2 alone [2, 24]. Given a vector $\mathbf{r} = (r_x, r_y)$, it yields the probability that $\mathbf{r}$ lies entirely within a single phase if it is placed randomly in the structure. In contrast to S_2, which considers only the start and end point of the vector, L incorporates information about the connectedness of the phases. In this section $\tilde{L}$ is presented as a differentiable approximation to L.

The lineal path function is approximated using a *convolve-threshold-reduce* pipeline similarly to [26]. For this purpose, the vector or line $\mathbf{r}$ is discretized to a pixel grid as shown in Fig. 9 and divided by the length of the line. In this work, the Bresenham line algorithm [46] is used, but alternative approaches like the Xiaolin Wu line algorithm [47] might be equally viable options that can be investigated in future works. In the *convolve* step, the discretized line from Fig. 9 is used as a mask for a convolution with periodic boundary conditions. The output of the convolution is an image where each pixel corresponds to the discretized line being placed at the pixel's location. The pixel value is 0 if no part of the line lies in phase 1 and 1 if the line lies completely in phase 1. If only parts of the line are in phase 1, the value is between 0 and 1. These pixels can be set to zero by thresholding the image with a value t, where $1/(1-||\mathbf{r}||_\infty) < t < 1$. The *threshold* step thus yields an image which can be interpreted as an ensemble of realizations, where each pixel takes the value 0 or 1. In the *reduce* step, this ensemble is averaged to obtain the probability of a randomly placed line being entirely in phase 1.

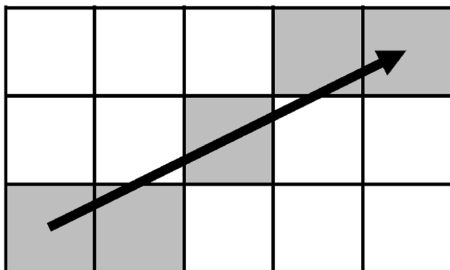


Fig. 9 A discretization of the vector $\mathbf{r}$ to a discrete pixel grid using Bresenham's method [46]

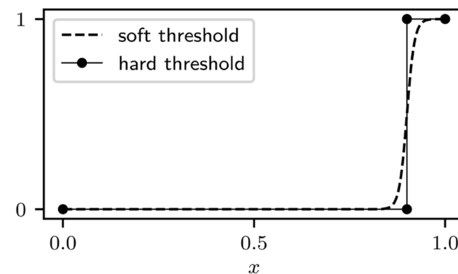


Fig. 10 Approximation of a hard threshold by a scaled and shifted differentiable sigmoid function

To make the *convolve-threshold-reduce* pipeline differentiable, only the thresholding needs to be modified. The hard thresholding is therefore approximated using a scaled and shifted differentiable sigmoid function[8] as shown in Fig. 10. This introduces errors, because the ensemble to average does not contain only ones and zeros but also intermediate values. Unlike in [26], where a similar error for $\tilde{S}$ could be eliminated by deriving a correction step, the difference between L and $\tilde{L}$ cannot be quantified easily. Thus, it is clear that $\tilde{L}$ is only an approximation to L, not a generalization. This is not problematic if the same descriptor is used for characterization and reconstruction.

To the authors' best knowledge, this concludes the first differentiable approximation to the lineal path function.

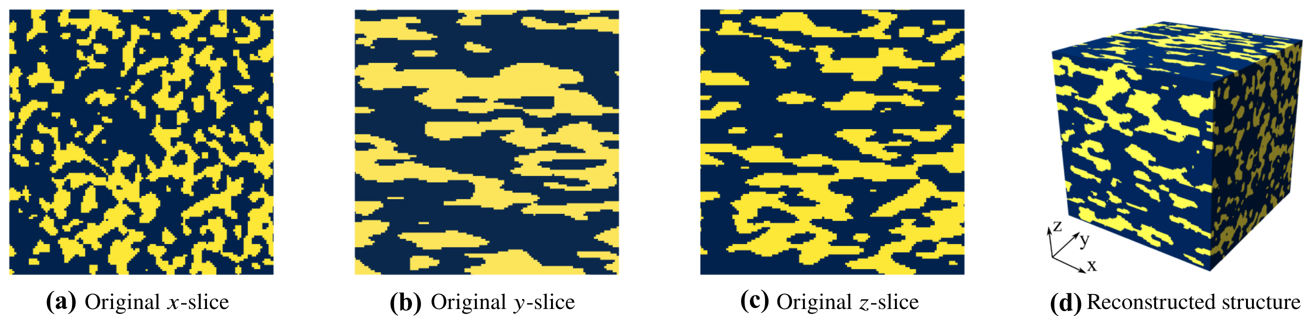
Appendix 2: Underlying Technologies

MCRpy is programmed in Python and based on very common packages like *Numpy*, *Scipy*, *Matplotlib* and *TensorFlow*. While simulated annealing is implemented from scratch, the gradient-based optimizers are taken from *Scipy* and *TensorFlow*. Furthermore, *TensorFlow* is used for automatic differentiation of the loss function and the descriptors as well as for just-in-time compilation via *AutoGraph*. This allows *MCRpy* to run highly optimized code on GPUs despite being written entirely in Python. As optional dependencies, *Gooey* is required to run the *MCRpy* GUI and *pyMKS* is used in a descriptor plugin for FFT-based 2-point correlations. A summary of required and optional dependencies and their versions is given in Table 5.

Table 5 Software dependencies for the current version of *MCRpy*

Package	Required	Version
numpy	yes	≥ 1.20.1
matplotlib	yes	≥ 3.3.4
scipy	yes	≥ 1.6.2
tensorflow	yes	≥ 2.3.1
pymks	for FFT-based correlations	≥ 0.4.1
gooey	for GUI	≥ 1.0.8.1

[8] We use the same function as in [26].



(a) Original x-slice **(b)** Original y-slice **(c)** Original z-slice **(d)** Reconstructed structure

Fig. 11 Results for reconstructing a 3D microstructure from 3 orthogonal 2D slices

Appendix 3: 2D-to-3D Reconstruction from Multiple Images

Reconstructing a 3D microstructure from multiple orthogonal 2D slices requires one step more than direct isotropic 2D-to-3D reconstruction. As discussed in Sect. Manipulating the descriptor space, after characterization, the descriptors need to be merged using the `merge` function before reconstruction. Like any operation, this can be done in the graphical interface, on the command line or via Python as shown in Listing 2. An exemplary result is shown in Fig. 11.

Appendix 4: Computational Efficiency

The computational efficiency of *MCRpy* is measured in terms of wallclock time and memory requirements on an *Nvidia A100* GPU. Note that both quantities depend on the chosen descriptor and its accuracy controlled by the `limit_to` parameter as well as the microstructure resolution.

The wallclock time and memory requirements are reported in Table 6 for varying `limit_to` and in Table 7 for varying microstructure resolution. While the descriptor-based reconstruction in *MCRpy* is not as fast as machine learning-based methods such as [48, 49] it lies in a similar order of magnitude as a numerical simulation of the generated structures. Regarding the memory requirements, it should be mentioned that the just-in-time compilation of *TensorFlow* trades off available memory for improved wallclock time if possible. Furthermore, it should be noted that the memory requirements depend on the chosen optimizer. While L-BFGS-B [36] is chosen here due to its fast convergence, the Adam optimizer [39] can be used to save memory by increasing the number of iterations.

Finally, the scalability for a number of indicator functions $p > 2$ is discussed. The wallclock time per iteration as well as the memory consumption both grow linearly in p because the reconstruction cost is governed by the computation of the descriptors and their gradients. The requirement that the sum of all indicator functions should be 1 at each position causes negligible computational overhead per iteration. However, depending on how it is enforced during optimization, it can influence the wallclock time by an increased number of required iterations. While Simulated Annealing fulfills this requirement by construction, all currently implemented gradient-based optimizers use a penalty method. In practice, the authors have observed a slightly increased number of iterations for $p = 3$, but have no experience with $p \geq 4$.

Table 6 Wallclock time and memory requirements for reconstructing the structure from Sect. Obtaining a 3D domain from a 2D microstructure slice with different values of `limit_to`. Note that *TensorFlow* trades off memory for speed and lower memory consumption is possible.

`Limit_to` in px	Time in min	RAM in GB
4	5	4.2
8	7	4.2
16	11	4.4
32	51	5.1

Note that *TensorFlow* trades off memory for speed and lower memory consumption is possible.

Table 7 Wallclock time and memory requirements for reconstructing the structure from Sect. Obtaining a 3D domain from a 2D microstructure slice with different microstructure resolutions and `limit_to = 8`. Note that *TensorFlow* trades off memory for speed and lower memory consumption is possible.

Resolution in px	Time in min	RAM in GB
64	7	4.2
128	20	5.9
256	102	18.1

Acknowledgements The group of M. Kästner thanks the German Research Foundation DFG which supported this work under Grant number KA 3309/18-1. Furthermore, this work is partially funded by the European Regional Development Fund (ERDF) and co-financed by

tax funds based on the budget approved by the members of the Saxon State Parliament under Grants 100373334. All presented computations were performed on a HPC-Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden. The authors thus thank the ZIH for generous allocations of computer time.

Author Contributions **P. S** was involved in the conceptualization, formal analysis, investigation, methodology, software, validation, visualization, writing—original draft, writing—review and editing. **A. R** contributed to the conceptualization, software, writing—review and editing. **K. K** helped in the supervision, visualization, writing—review and editing. **M. A** was involved in the supervision, writing—review and editing. **M. K** contributed to the funding acquisition, resources, supervision, writing—review and editing.

Funding Open Access funding enabled and organized by Projekt DEAL.

Code availability *MCRpy* is available on the *GitHub* repository https://github.com/NEFM-TUDresden/MCRpy.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Chen W, Iyer A, Bostanabad R (2022) Data-centric design: a new approach to design of microstructural materials systems. Engineering. https://doi.org/10.1016/j.eng.2021.05.022
2. Bostanabad R, Zhang Y, Li X, Kearney T, Brinson LC, Apley DW, Liu WK, Chen W (2018) Computational microstructure characterization and reconstruction: review of the state-of-the-art techniques. Prog Mater Sci 95:1–41. https://doi.org/10.1016/j.pmatsci.2018.01.005
3. Khatamsaz D, Molkeri A, Couperthwaite R, James J, Arróyave R, Srivastava A, Allaire D (2021) Adaptive active subspace-based efficient multifidelity materials design. Mater Des 209:110001. https://doi.org/10.1016/j.matdes.2021.110001
4. Seibert P, Raßloff A, Ambati M, Kästner M (2022) Descriptor-based reconstruction of three-dimensional microstructures through gradient-based optimization. Acta Mater 227:117667. https://doi.org/10.1016/j.actamat.2022.117667
5. Liu H, Yucel B, Wheeler D, Ganapathysubramanian B, Kalidindi SR, Wodo O (2022) How important is microstructural feature selection for data-driven structure-property mapping? MRS Commun. https://doi.org/10.1557/s43579-021-00147-4
6. Sonnenburg S, Braun ML, Ong CS, Bengio S, Bottou L, Holmes G, LeCun Y (2007) The need for open source software in machine learning. J Mach Learn Res, p. 25. http://jmlr.org/papers/v8/sonnenburg07a.html
7. de Pablo JJ, Jones B, Kovacs CL, Ozolins V, Ramirez AP (2014) The materials genome initiative, the interplay of experiment, theory and computation. Curr Opin Solid State Mater Sci 18(2):99–117. https://doi.org/10.1016/j.cossms.2014.02.003
8. Nanomine: Ontology-enabled polymer nanocomposite open community data resource (2022). https://tw.rpi.edu/project/nanomine/
9. European center of excellence for novel materials discovery (NOMAD-CoE) (2021). https://nomad-lab.eu/
10. Ghiringhelli LM, Carbogno C, Levchenko S, Mohamed F, Lüders M, Oliveira M, Scheer M (2016) Towards a common format for computational materials science data. arXiv:1607.04738 pp. 1–16
11. Computational design and discovery of novel materials (NCCR MARVEL) (2021). https://www.nccr-marvel.ch/
12. Automated interactive infrastructure and database for computational science (2021). https://www.aiida.net/
13. Pizzi G, Cepellotti A, Sabatini R, Marzari N, Kozinsky B (2016) AiiDA: automated interactive infrastructure and database for computational science. Comput Mater Sci 111:218–230. https://doi.org/10.1016/j.commatsci.2015.09.013
14. Ong SP, Richards WD, Jain A, Hautier G, Kocher M, Cholia S, Gunter D, Chevrier VL, Persson KA, Ceder G (2013) Python materials genomics (pymatgen): a robust, open-source python library for materials analysis. Comput Mater Sci 68:314–319. https://doi.org/10.1016/j.commatsci.2012.10.028
15. Hayashi Y, Shiomi J, Morikawa J, Yoshida R (2022) RadonPy: automated physical property calculation using all-atom classical molecular dynamics simulations for polymer informatics. arXiv:2203.14090 p. 42
16. Brough DB, Wheeler D, Kalidindi SR (2017) Materials knowledge systems in python–a data science framework for accelerated development of hierarchical materials. Int Mater Manuf Innov 6(1):36–53. https://doi.org/10.1007/s40192-017-0089-0
17. Fast T, Kalidindi SR (2011) Formulation and calibration of higher-order elastic localization relationships using the MKS approach. Acta Mater 59(11):4595–4605. https://doi.org/10.1016/j.actamat.2011.04.005
18. Cimrman R (2014) SfePy - write your own FE application. Proc. of the 6th Eur. Conf. on Python in Science (Euroscipy 2013) pp. 69–69
19. Cimrman R, Lukeš V, Rohan E (2019) Multiscale finite element calculations in python using SfePy. Adv Comput Math 45(4):1897–1921. https://doi.org/10.1007/s10444-019-09666-0
20. Roters F, Diehl M, Shanthraj P, Eisenlohr P, Reuber C, Wong S, Maiti T, Ebrahimi A, Hochrainer T, Fabritius HO, Nikolov S, Friák M, Fujita N, Grilli N, Janssens K, Jia N, Kok P, Ma D, Meier F, Werner E, Stricker M, Weygand D, Raabe D (2019) DAMASK - the düsseldorf advanced material simulation kit for modeling multi-physics crystal plasticity, thermal, and damage phenomena from the single crystal up to the component scale. Comput Mater Sci 158:420–478. https://doi.org/10.1016/j.commatsci.2018.04.030
21. Keshav S, Fritzen F, Kabel M (2022) FFT-based homogenization at finite strains using composite boxels (ComBo). arXiv:2204.13624 [cs, math]
22. Groeber MA, Jackson MA (2014) DREAM.3D: A digital representation environment for the analysis of microstructure in 3D. Int Mater Manuf Innov 3(1):56–72. https://doi.org/10.1186/2193-9772-3-5
23. Azhari F, Davids W, Chen H, Ringer SP, Wallbrink C, Sterjovski Z, Crawford BR, Agius D, Wang CH, Schaffer G (2022) A comparison of statistically equivalent and realistic microstructural

representative volume elements for crystal plasticity models. Int Mater Manuf Innov. https://doi.org/10.1007/s40192-022-00257-4
24. Yeong CLY, Torquato S (1998) Reconstructing random media. Phys Rev E 57(1):495–506. https://doi.org/10.1103/PhysRevE.57.495
25. Jiao Y, Stillinger FH, Torquato S (2007) Modeling heterogeneous materials via two-point correlation functions: basic principles. Phys Rev E 76(3):031110. https://doi.org/10.1103/PhysRevE.76.031110
26. Seibert P, Ambati M, Raßloff A, Kästner M (2021) Reconstructing random heterogeneous media through differentiable optimization. Comput Mater Sci. https://doi.org/10.1016/j.commatsci.2021.110455
27. Lu B, Torquato S (1992) Lineal-path function for random heterogeneous materials. Phys Rev A 45(2):922–929. https://doi.org/10.1103/PhysRevA.45.922
28. Jiao Y, Stillinger FH, Torquato S (2009) A superior descriptor of random textures and its predictive capacity. Proceed Natl Acad Sci 106(42):17634–17639. https://doi.org/10.1073/pnas.0905919106
29. Chen PE, Xu W, Chawla N, Ren Y, Jiao Y (2019) Novel hierarchical correlation functions for quantitative representation of complex heterogeneous materials and microstructural evolution. SSRN Electron J. https://doi.org/10.2139/ssrn.3397269
30. Lubbers N, Lookman T, Barros K (2017) Inferring low-dimensional microstructure representations using convolutional neural networks. Phys Rev E 96:052111. https://doi.org/10.1103/PhysRevE.96.052111
31. Li X, Zhang Y, Zhao H, Burkhart C, Brinson LC, Chen W (2018) A transfer learning approach for microstructure reconstruction and structure-property predictions. Sci Rep 8(1):13461. https://doi.org/10.1038/s41598-018-31571-7
32. Bostanabad R (2020) Reconstruction of 3D microstructures from 2D images via transfer learning. Computer-Aided Des 128:102906. https://doi.org/10.1016/j.cad.2020.102906
33. Bhaduri A, Gupta A, Olivier A, Graham-Brady L (2021) An efficient optimization based microstructure reconstruction approach with multiple loss functions. arXiv:2102.02407 [cond-mat]
34. Piasecki R, Plastino A (2010) Entropic descriptor of a complex behaviour. Phys A: Stat Mech Appl 389(3):397–407. https://doi.org/10.1016/j.physa.2009.10.013
35. Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556
36. Byrd RH, Hansen SL, Nocedal J, Singer Y (2015) A stochastic quasi-newton method for large-scale optimization. arXiv:1401.7020 [cs, math, stat]
37. Cule D, Torquato S (1999) Generating random media from limited microstructural information via stochastic optimization. J Appl Phys 86(6):3428–3437. https://doi.org/10.1063/1.371225
38. Nash SG (1984) Newton-type minimization via the lanczos method. SIAM J Numer Anal 21(4):770–788. https://doi.org/10.1137/0721052
39. Kingma DP, Ba J (2017) Adam: A method for stochastic optimization. arXiv:1412.6980 [cs] pp. 1–15
40. Duchi J, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12(7):39
41. Zeiler MD (2012) ADADELTA: An adaptive learning rate method. arXiv:1212.5701 [cs]
42. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX conference on operating systems design and implementation, OSDI'16, p. 265-283. USENIX Association, USA
43. Cecen A, Yucel B, Kalidindi SR (2021) A generalized and modular framework for digital generation of composite microstructures. J Compos Sci 5(8):211. https://doi.org/10.3390/jcs5080211
44. Ahrens J, Geveci B, Law C (2005) Paraview: An end-user tool for large data visualization. The visualization handbook. Elsevier, p 717(8)
45. Raßloff A, Schulz P, Kühne R, Ambati M, Koch I, Zeuner AT, Gude M, Zimmermann M, Kästner M (2021) Accessing pore microstructure-property relationships for additively manufactured materials. GAMM-Mitt. https://doi.org/10.1002/gamm.202100012
46. Bresenham JE (1965) Algorithm for computer control of a digital plotter. IBM Syst J 4(1):25
47. Wu X (1991) An efficient antialiasing technique. Comput Gr 25(4):143–152
48. Bostanabad R, Chen W, Apley D (2016) Characterization and reconstruction of 3D stochastic microstructures via supervised learning. J Microsc 264(3):282–297. https://doi.org/10.1111/jmi.12441
49. Kench S, Cooper SJ (2021) Generating 3D structures from a 2D slice with GAN-based dimensionality expansion. Nat Mach Intell 3:299–305. https://doi.org/10.1038/s42256-021-00322-1