



## Adorym: a multi-platform generic X-ray image reconstruction framework based on automatic differentiation: supplement

MING DU,<sup>1,\*</sup>  SAUGAT KANDEL,<sup>2</sup> JUNJING DENG,<sup>1</sup> XIAOJING HUANG,<sup>3</sup> ARNAUD DEMORTIERE,<sup>4</sup> TUAN TU NGUYEN,<sup>4</sup> REMI TUCOULOU,<sup>5</sup> VINCENT DE ANDRADE,<sup>1</sup> QIAOLING JIN,<sup>6,7</sup> AND CHRIS JACOBSEN<sup>1,6,7</sup> 

<sup>1</sup>Advanced Photon Source, Argonne National Laboratory, Lemont, Illinois 60439, USA

<sup>2</sup>Applied Physics Program, Northwestern University, Evanston, Illinois 60208, USA

<sup>3</sup>National Synchrotron Light Source II, Brookhaven National Laboratory, Upton, New York 11973, USA

<sup>4</sup>Laboratoire de Réactivité et Chimie des Solides (LRCs), CNRS UMR 7314, Université de Picardie Jules Verne, Hub de l'Energie, 15 Rue Baudelocque, 80039 Amiens Cedex, France

<sup>5</sup>European Synchrotron Radiation Facility, 71 Avenue des Martyrs, 38000 Grenoble, France

<sup>6</sup>Department of Physics & Astronomy, Northwestern University, Evanston, Illinois 60208, USA

<sup>7</sup>Chemistry of Life Processes Institute, Northwestern University, Evanston, Illinois 60208, USA

\*[mingdu@anl.gov](mailto:mingdu@anl.gov)

---

This supplement published with The Optical Society on 16 March 2021 by The Authors under the terms of the [Creative Commons Attribution 4.0 License](https://creativecommons.org/licenses/by/4.0/) in the format provided by the authors and unedited. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

Supplement DOI: <https://doi.org/10.6084/m9.figshare.13824143>

Parent Article DOI: <https://doi.org/10.1364/OE.418296>

# Adorym: A multi-platform generic x-ray image reconstruction framework based on automatic differentiation (supplemental document)

We describe and demonstrate an optimization-based x-ray image reconstruction framework called Adorym. Our framework provides a generic forward model, allowing one code framework to be used for a wide range of imaging methods ranging from near-field holography and fly-scan ptychographic tomography. By using automatic differentiation for optimization, Adorym has the flexibility to refine experimental parameters including probe positions, multiple hologram alignment, and object tilts. It is written with strong support for parallel processing, allowing large datasets to be processed on high-performance computing systems. We demonstrate its use on several experimental datasets to show improved image quality through parameter refinement.

## CONTENTS

<b>1</b>	<b>Modules and components</b>	<b>1</b>
A	Data format	2
B	Module LargeArray	3
C	Module Propagate	4
D	Module ForwardModel	6
D.1	Loss function	7
D.2	Parameter refinement	8
E	Module Wrapper and backends	11
F	Module Optimizer	11
G	Parallelization modes	13
G.1	Data parallelism (DP) mode	14
G.2	Distributed object (DO) mode	14
G.3	HDF5-file-mediated low-memory mode (H5)	15
G.4	User interface	16
<b>2</b>	<b>Definition of structural similarity index (SSIM)</b>	<b>17</b>

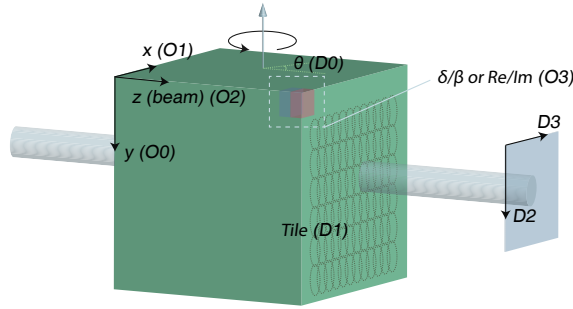
## 1. MODULES AND COMPONENTS

As noted in Section 2 of the main manuscript, Adorym is written in a modular and object-oriented programming style, consisting primarily of the following classes:

- The `ForwardModel` class discussed in Sec. D, which contains the forward model corresponding to a particular imaging method.
- The `Optimizer` class discussed in Sec. F, which provides options on what optimization method to use for minimizing the loss function  $L$ .
- The `LargeArray` class discussed in Sec. B, which manages the large array of the object, or  $x$ .
- The `propagate` class discussed in Sec. C, which handles wavefield propagation for those forward models that require it.
- The `wrapper` module that provides a common interface for functions in the two AD engines provided: *Autograd*, and *PyTorch*.

Modules like `ForwardModel` and `Optimizer` each contain several child classes, providing support for different types of imaging techniques or setups, numerical optimization algorithms, distributed computation methods, and automatic differentiation backends (and computation devices, namely CPU and GPU). We have therefore attempted to make it easy to combine varying





**Fig. S2.** Representation of experimental coordinates in Adorym’s readable dataset ( $D$ ) and object function array ( $O$ ). Directions and quantities are labeled with the index of dimension in the corresponding array; for example,  $O2$  means that the associated object axis is stored as the 2nd dimension of the object array.

interconnected tiles with size given by  $L_{\text{tile},y} = \lceil L_y / N_{\text{tile},y} \rceil$  and  $L_{\text{tile},x} = \lceil L_x / N_{\text{tile},x} \rceil$ , where  $\lceil x \rceil$  is the ceiling function that returns the smallest integer larger or equal to  $x$ . If MDH data are being tiled, all the tiles of images acquired at different distances are collectively regarded as “scan spots”, and are saved contiguous in the second dimension of the dataset following a “tile-then-distance” order.

It follows from our introduction above that a “tile” is the fundamental level of data organization scheme used in Adorym for all range of imaging techniques. In far-field ptychography, a tile refers to a diffraction pattern; in full-field imaging methods, a tile may refer to the image acquired at a certain viewing angle if the raw data are undivided, or a subblock of the image if they are divided. We shall thus use the word “tile” as a general term when referring to these data elements.

## B. Module LargeArray

The `LargeArray` class for large arrays is used to hold the object (when it is kept or partially kept in memory; more about this will be introduced in subsection G), or arrays that are comparable in size as the object (such as the object gradient and the finite support mask). The parent class of `LargeArray` contains common methods such as array rotation (important for processing tomographic data), HDF5 read/write (used in HDF5-mediated distribution mode), and MPI-based data synchronization (used in distributed object mode). The latter two methods mentioned above are specifically relevant to distributed reconstruction modes where the object function is either scattered over processes or nodes, or stored in a HDF5 file on the hard drive; distribution schemes for these cases are discussed in Section G. Three child classes, namely `ObjectFunction`, `Mask`, and `Gradient`, inherit properties from the parent class, and possesses additional methods developed for their specific roles and purposes. For example, the `ObjectFunction` child class contains methods to initialize the object function with Gaussian random values or a user-supplied initial guess, and the `Mask` class includes a `update_finite_support` function, which revises the mask by setting voxels associated with low object function values to zero – a technique known as shrink-wrapping in coherent diffraction imaging [4].

The unknown object to be solved is represented by an instance of the `ObjectFunction` class: a 4D array where the first 3 dimensions correspond to the  $y$  (vertical),  $x$  (horizontal), and  $z$  (beam axial) coordinates of the object (also see Fig. S2 for a illustration of the geometry). The last dimension depends on the mathematical representation of the object: Adorym allows the object function to be reconstructed as either a complex modulation function, where a wavefield  $\psi(\mathbf{r})$  is modulated by the object  $O(\mathbf{r})$  as

$$M[\psi(\mathbf{r}_{x,y})] = \psi(\mathbf{r}_{x,y}) \cdot (\Re[O(\mathbf{r}_{x,y})] + i\Im[O(\mathbf{r}_{x,y})]), \quad (\text{S1})$$

where  $\Re(\cdot)$  and  $\Im(\cdot)$  respectively take the real and imaginary part of the argument, or a distribution of refractive indices  $n(\mathbf{r}) = 1 - \delta(\mathbf{r}) - i\beta(\mathbf{r})$  [5], where the modulation to an input x-ray wavefield is given as

$$M[\psi(\mathbf{r}_{x,y})] = \psi(\mathbf{r}_{x,y}) \exp[-k\beta(\mathbf{r}_{x,y})\Delta z] \exp[ik\delta(\mathbf{r}_{x,y})\Delta z] \quad (\text{S2})$$

where  $k = 2\pi/\lambda$  is the wavenumber, and  $\Delta z$  is the thickness of the modulating object. Since the object is assumed to be a 3D function while the wavefield is 2D, we use  $\mathbf{r}_{x,y}$  to represent the lateral coordinates of a plane in the object. In the case of  $\Re[O(\mathbf{r}_{x,y})] + i\Im[O(\mathbf{r}_{x,y})]$ , the last dimension of the object array stores the real and imaginary part of the complex modulation function  $O(\mathbf{r})$ , while in the case of  $\delta(\mathbf{r}) + i\beta(\mathbf{r})$ , the last dimension stores  $\delta(\mathbf{r})$  and  $\beta(\mathbf{r})$ , respectively. The complex multiplicative modulation function representation used in Eq. S1 is more common in phase retrieval techniques such as ptychography [6], and we have found it sometimes shows better numerical robustness in optimization-based reconstruction because it avoids the ill-conditioned exponential function. However, since the phase angle  $\phi(\mathbf{r})$  can only be recovered from the reconstructed complex object function through  $\text{Arg}[O(\mathbf{r})]$ , phase wrapping is present in high phase contrast objects. In that case, post-processing is required to unwrap the phase [7], but this can be complicated when noise or strongly varying phases are present [8]. The situation is easier in the refractive index representation of the object (Eq. S2). When the refractive indices are solved without regularization, the phase part  $\delta(\mathbf{r})$  may still suffer from the  $2\pi$  ambiguity since it is contained in the periodic function  $\exp[ik\delta(\mathbf{r})\Delta z]$  as in Eq. S2. However,  $\delta(\mathbf{r})$  can be regularized to promote spatial smoothness. That is, if the values of  $\exp[ik\delta(\mathbf{r})\Delta z]$  at two adjacent object pixels are  $-1 + \epsilon_1$  and  $-1 - \epsilon_2$  (where  $\epsilon_1$  and  $\epsilon_2$  are complex numbers conjugate about the real axis), then the corresponding value of  $\delta(\mathbf{r})$  for the former is given by

$$\begin{aligned} \exp[ik\delta(\mathbf{r}_1)\Delta z] &= -1 + \epsilon_1 \\ \exp(i\pi) + i \exp(i\pi)[ik\delta(\mathbf{r}_1)\Delta z - \pi] &\approx -1 + \epsilon_1 \\ k\delta(\mathbf{r}_1)\Delta z &= i\epsilon_1 + \pi + 2\pi m \end{aligned} \quad (\text{S3})$$

and similarly,  $k\delta(\mathbf{r}_2)\Delta z = i\epsilon_2 + \pi + 2\pi n$ , where  $m$  and  $n$  can be any arbitrary integers. It follows that  $|\delta(\mathbf{r}_2) - \delta(\mathbf{r}_1)| = \frac{1}{k\Delta z}[2\pi(n - m) + i(\epsilon_2 - \epsilon_1)]$  is unrestricted. Yet, if one applies TV regularization on  $\delta(\mathbf{r})$ , then a small  $|\delta(\mathbf{r}_2) - \delta(\mathbf{r}_1)|$  is favored. While  $i(\epsilon_2 - \epsilon_1)$  is preserved by the data fidelity term of the loss function, the TV term favors  $|m - n| = 0$ , which leads to a solution that is spatially continuous without the  $2\pi$  jumps in phase wrapping. More importantly, since the range of  $\delta(\mathbf{r})$  itself is continuous and unbounded, the solved values of  $\delta(\mathbf{r})$  can be ones that push the phase angle  $k\delta(\mathbf{r})\Delta z$  out of the bound of  $(-\pi, \pi]$ , which would be impossible when one solves for the real and imaginary parts of  $O(\mathbf{r})$  instead: in the latter case, even if one regularizes  $\text{Arg}[O(\mathbf{r})]$ , the computed phase is still bounded between  $(-\pi, \pi]$ , and TV only puts a  $\pi$ -ceiling or  $-\pi$ -floor for slightly out-of-bound pixels aside from smoothing the reconstructed image. By solving for  $\delta(\mathbf{r})$ , one may directly obtain a phase-unwrapped solution without any post-processing. Furthermore, since  $\delta(\mathbf{r})$  of natural samples is positive except near an absorption edge, this physical property can be used to further regularize  $\delta(\mathbf{r})$  through a non-negativity constraint. The  $\delta(\mathbf{r})$ -representation is common in direct phase retrieval methods, and one example is the algorithm based on the contrast transfer function (CTF) [3, 9], which derives a linear relation between the phase and the Fourier transform of the detected intensity from Eq. S2, based on the assumption that the object is homogeneous and weak in phase. Moreover, solving the refractive indices of the object is also proven to work with iterative reconstruction algorithms in [10] and by us in [11].

### C. Module Propagate

The propagate module is essentially a toolbox for simulating wave propagation, the central component in building the forward model. For a wavefield propagating in vacuum (or air, without any substantial loss of accuracy), two scenarios are considered: when the Fresnel number of propagation is small as in the case of far-field imaging techniques, Fraunhofer diffraction can be used to calculate the wavefield at the destination plane, which is simply done by Fourier transforming the source wavefield:

$$\begin{aligned} P_{\text{far}}[\psi(\mathbf{r})] &= \int \psi(\mathbf{r}') \exp[i2\pi(\mathbf{r} \cdot \mathbf{v}')] d\mathbf{r}' \\ &= \mathcal{F}[\psi(\mathbf{r})]. \end{aligned} \quad (\text{S4})$$

One thing to note here is that there exist two sign conventions in formulating wave propagation equations. Most equations in this paper are written assuming a “negative phase convention” – that is, the phase factor of a wavefield evolves with propagation distance as  $\exp(-ikz)$ , and accordingly, the refractive index is expressed as  $n(\mathbf{r}) = 1 - \delta(\mathbf{r}) - i\beta(\mathbf{r})$  [12]. In another often-used sign convention, phase evolves as  $\exp(ikz)$  and refractive index takes the form of  $n(\mathbf{r}) = 1 - \delta(\mathbf{r}) +$

$i\beta(\mathbf{r})$  [13]. Eq. S5 is consistent with the negative phase convention, which results in a positive argument in the exponential phase factor of the Fourier transform's integrand. Using the positive phase convention, on the other hand, is associated with a negative sign in the Fourier phase factor's argument. Some scientific computation packages used in Adorym including *Autograd* and *PyTorch* implement Fourier transform (*i.e.*, the `fft` function) with a negative exponential argument for forward Fourier transform, and a positive exponential argument for inverse Fourier transform. Therefore, Fraunhofer diffraction in the negative phase convention should in fact be implemented using the `ifft` functions of these packages, and vice versa for the positive phase convention. While both conventions should lead to the same intensity prediction, it is important stay consistent with the sign convention throughout the forward model. In Adorym, most optical propagation and modulation functions come with a `sign_convention` argument, so that users who attempt to create new forward models can switch the convention according to their preference.

When not satisfying the requirements of the Fraunhofer approximation (such as in holography, or cases where the multislice method needs to be applied), Fresnel propagation is used. For propagation over a short distance  $d$  where the Fresnel number is large [14], Fresnel propagation can be expressed as

$$P_{\text{Fr},d}[\psi(\mathbf{r})] = \mathcal{F}^{-1}\{\mathcal{F}[\psi(\mathbf{r})]H(\mathbf{v},d)\} \quad (\text{S6})$$

where the Fresnel propagator kernel  $H$  in Fourier space can be either from the paraxial approximation of

$$H_1(\mathbf{v},d) = \exp(i\pi\lambda d|\mathbf{v}|^2) \quad (\text{S7})$$

or the Sommerfeld-Rayleigh formulation [13] of

$$H_2(\mathbf{v},d) = \begin{cases} \exp\left[-ikd\sqrt{1-\lambda^2|\mathbf{v}|^2}\right], & 1-\lambda^2|\mathbf{v}|^2 > 0 \\ 0, & 1-\lambda^2|\mathbf{v}|^2 \leq 0 \end{cases} \quad (\text{S8})$$

where the frequency components with  $1-\lambda^2|\mathbf{v}|^2 \leq 0$  result in evanescent waves that die down quickly.

The process through which the sample interacts with the wavefield is modeled using the `multislice_propagate_batch` function. The multislice method was first used in electron microscopy [15, 16], and is also known as the beam propagation method in optics [17]. Multislice propagation treats a 3D object as a stack of thin slices along the beam axis, assuming the refractive index at each lateral ( $x$ - $y$ ) point in a slice is axially constant. At each slice, the wavefield is modulated using either Eq. S1 or S2 depending on how the object is represented, and then propagated to the next slice using Fresnel propagation (Eq. S6). The steps above are repeated until the wavefield reaches the last slice of the object. The entire process can be compactly represented in matrix form as

$$\boldsymbol{\psi}_{\text{exit}} = \prod_j^{L_z} \left( P_{\Delta z} M_{j,\Delta z} \right) \boldsymbol{\psi} \quad (\text{S9})$$

where  $M_{n,\theta,\Delta z}$  extracts the  $j$ -th slice of the object with thickness (or spacing)  $\Delta z$ , and uses it to modulate wavefield  $\boldsymbol{\psi}$ ;  $P_d$  is the linear operator that implements Fresnel propagation over slice spacing  $\Delta z$ .

The implementation of multislice propagation in Adorym is again generalizable: a 2D object is interpreted to contain just 1 slice, and the wavefield is modulated by the only slice before it is propagated to the detector plane. On the other hand, when the object to be reconstructed is within the depth of focus, the `pure_projection` option can be turned on: in this case, the modulation of the wavefield by the entire object is computed as

$$M_{\text{obj}}[\psi(\mathbf{r}_{x,y})] = \psi(\mathbf{r}_{x,y}) \prod_j^{L_z} O(\mathbf{r}_{x,y},j) \quad (\text{S10})$$

if the object is represented as complex modulation function, or

$$M_{\text{obj}}[\psi(\mathbf{r}_{x,y})] = \psi(\mathbf{r}_{x,y}) \exp\left[-k\sum_j^{L_z}\beta(\mathbf{r}_{x,y})\Delta z\right] \exp\left[ik\sum_j^{L_z}\delta(\mathbf{r}_{x,y})\Delta z\right] \quad (\text{S11})$$

if it is represented as refractive indices.

The `multislice_propagate_batch` function is intended for reconstructing a continuous 3D object with isotropic voxel size – that is, it assumes that the spacing  $\Delta z$  between slices is equal to the lateral pixel size  $\Delta x$ . By binning the slices, the function can work with  $\Delta z$  being multiples of  $\Delta x$ , but the slice spacings are assumed to be constant. This setting works well with common 2D ptychography (which does not need slice spacing at all) and beyond-depth-of-focus ptychotomography [11]. On the other hand, in multislice ptychography [18–20], multiple slices of the object are reconstructed using ptychography data from one or just a few tilt angles. The slices in these cases are generally much sparser, and the slice spacing can be variable. Thus, we also implemented a sparse-and-variable version of multislice propagation in Adorym, where the positions of each slice are saved as an array. The array is optimizable in the AD framework if slice positions need to be refined.

#### D. Module ForwardModel

The `ForwardModel` class and its child classes define the prediction models corresponding to various types of imaging techniques, which take as inputs the object function to be solved, the probe function that may be known or unknown, and a collection of experimental parameters. They predict (generally the magnitude of) the wavefields at the detector plane according to given experimental setups and optical theories. Although Adorym can be easily adapted to work with complex-valued measurements (for example, obtained using interferometry [21, 22]), intensity-only measurements are more common and easier to setup. If multiple probe modes are used, the intensity will be the incoherent sum of all probe modes:

$$I_{\text{pred}} = \sum_i^{n_{\text{modes}}} |\psi_{i,\text{detector}}|^2 \quad (\text{S12})$$

and magnitude is calculated as the square root of the summed intensity.

**Algorithm 1:** An example `predict` function in the forward model of far-field ptychotomography. This is for data parallelism mode, with probe position refinement and multiple probe modes.

```

Input:
detector size  $[l_y, l_x]$ , object size  $[L_y, L_x, L_z]$ , full object function  $O$ , a list of probe modes  $\Psi$ ,
probe positions  $R$ , probe position corrections  $\Delta R$ , wavelength  $\lambda$ , pixel size  $\delta$ , current
rotation angle  $\theta$ 
Initialization
|  $o \leftarrow []$  // List of object chunks each in shape of  $[l_y, l_x, L_z]$ 
|  $\Psi_{\text{detector}} \leftarrow []$  // List of detector-plane waves
|  $\Psi_{\text{shifted}} \leftarrow []$  // Shifted probe functions
end
Procedure
|  $O' \leftarrow \text{Rotate}(O)$ 
| for  $r$  in  $R$  do
| | /* Get local object chunks corresponding to diffraction patterns
| | (tiles). */
| |  $o.append(\text{TakeObjectChunk}(O', r))$ 
| | /* Shift probes according to the list of position corrections. */
| |  $\Psi_{\text{shifted}}.append(\text{FourierShift}(\Psi, \Delta R))$ 
| end
| /* Do propagation for all probe modes. */
| for  $\psi_i$  in  $\Psi_{\text{shifted}}$  do
| |  $\psi_i \leftarrow \text{MultislicePropagate}(\psi_i, o, \lambda, \delta)$ 
| |  $\psi_i \leftarrow \text{FarFieldPropagate}(\psi_i)$ 
| |  $\Psi_{\text{detector}}.append(\psi_i)$ 
| end
end
Output:  $\Psi_{\text{detector}}$ 

```

Any experimental variables that are to be refined, such as probe positions, geometric transformation of projection images, and propagation distances, should also be incorporated as a part of the model. Collectively, these procedures, or the prediction function, are given in the `predict`



method inside each child class. An example of the `predict` method used for far-field ptychography with probe position correction is shown in Algorithm 1. The example assumes simple data parallelism, where each rank possesses its own copy of the whole object function. In this case, the 3D object is rotated to the current tilt angle  $\theta$ , and sub-chunks of the object corresponding to the beam path of each diffraction pattern are extracted. If probe position correction is enabled, the prediction function takes in an optimizable array of probe position corrections,  $\Delta\mathbf{R}$ , which has a length equal to the number of probe positions. Each element in the list is a 2D vector  $\Delta\mathbf{r}$  which represents the correction that needs to be made to the user-given position  $\mathbf{r}$ , so that the corrected position is  $\mathbf{r}' = \mathbf{r} + \Delta\mathbf{r}$ ; additionally, when scan positions are not integer pixel, the array  $\Delta\mathbf{R}$  can also be used to hold floating point residuals of probe positions that should be added to the integer pixel positions regardless whether probe position refinement is enabled or not. The correction shift  $\Delta\mathbf{r}$  is then applied to the probe function using the Fourier shift theorem:

$$\psi_{\text{shifted}}(\mathbf{r}) = \mathcal{F}^{-1} \{ \mathcal{F}[\psi(\mathbf{r})] \exp[-i2\pi(\Delta\mathbf{r} \cdot \mathbf{v})] \}. \quad (\text{S13})$$

Now this is already enough for the refinement: the gradient of the loss function with regards to the position correction vector  $\Delta\mathbf{r}$  is left for the AD engine to calculate behind the scene, and no explicit derivation or implementation is needed. More information about parameter refinement will be introduced in Section D.2.

When the size of the raw data is large, Adorym can be set to process only a subset, or a minibatch, of the measured data containing `batch_size` tiles at a time. Thus, we hereby define the terms we will use for describing the workload of reconstruction: an “iteration” is used to refer to the process of finishing a minibatch of data by all parallelized processes (the inner loop), and an “epoch” refers to the work done to cover the entire dataset (all tiles on all viewing angles; the outer loop). Propagation through the object is done using multislice propagation which is capable of simulating multiple scattering within the object volume and is useful when the object thickness is beyond the depth of focus [11, 15]. On the other hand, when projection approximation is valid, the multislice simulation can be reduced to a line-integration of the object function. The prediction function is concluded by propagating the wavefield onto the detector plane, and the final complex wavefields are returned.

The above illustrated routine can be easily generalized beyond the settings of far-field ptychography. Full-field CDI data, for instance, is interpreted by Adorym as a special type of ptychography with 1 tile per angle, and with detector size equal to the  $y$ - and  $x$ -dimensions of the object array. Furthermore, propagation from the object’s exiting plane to the detector plane is modeled using far-field diffraction in the example of Algorithm 1, but when working with near-field ptychography or tiled holography data (described in Section A), it can be replaced with a Fresnel propagation function. In addition, when processing tiled data with bright illumination (in contrast to far-field ptychography, where the illumination at each tile or diffraction spot is localized and has most of its energy concentrated to an area much smaller than the detector size), Fresnel propagation in object (when multislice modeling is enabled) and in free space of a wavefield tile may cause diffraction fringes to wrap around, due to the non-zero edges in the modulated wavefield. In this case, Adorym can select a larger area for each object chunk and wavefield tile, leaving a “buffer zone” around the wavefield for wrapping-around fringes to spread, and discard the buffer zones after the wavefield reaches the detector plane [23, 24].

When the illumination is only partially coherent [25] or when fly-scan is used in data acquisition [26–28], multiple mutually incoherent probe modes can be used for image reconstruction to account for the limited coherence of illumination wavefield. In this case, the prediction function gives the outputs of all probe modes individually, assuming each of them does not interact with others before arriving at the detector plane. As a general convention, the prediction functions in virtually all forward models return the detector-plane wavefields as two separate arrays that respectively stores the real and imaginary parts of the complex magnitude. Each of these arrays has a shape of `[batch_size, n_probe_modes, len_detector_y, len_detector_x]`. In this way, multi-mode reconstruction can be realized for various types of imaging experiments in addition to far-field ptychography.

#### D.1. Loss function

Each child class of `ForwardModel` contains also a `loss` method that takes in the predicted detector-plane magnitude  $\sqrt{I_{\text{pred}}}$  and the associated experimental measurement  $\sqrt{I_{\text{meas}}}$ , and calculates the loss from them (*i.e.*, the last-layer loss function). Combining `predict` and `loss`, another method `get_loss_function`, constructs an end-to-end loss function mapping the input variables



to the final loss, optionally with added regularization terms. The data mismatch term in the last-layer loss function can take different forms, each of them has specific advantages in terms of numerical robustness and noise resistance. The expressions of these data mismatch terms are defined in the parent `ForwardModel` class which are inherited by all its children classes the loss method, so users can easily select from existing types of the data mismatch term, modify them, or plug in new types, and then apply the change to all forward model classes. Adorym has two built-in types of data mismatch term, namely the least-square type (LSQ) and the Poisson maximum likelihood type.

The LSQ data mismatch term measures the Euclidian distance between the prediction and the measurement. As pointed out by Godard *et al.* [29], when the measured intensity  $I_{\text{meas}}$  follows Poisson statistics whose standard deviation is the square root of its mean, then the variance of  $\sqrt{I_{\text{meas}}}$  is insensitive to the expectation of  $I_{\text{meas}}$ . That is, using the square root of intensities, instead of intensities themselves in the LSQ loss function, may lead to better numerical robustness. As such, Adorym uses an LSQ-type loss function expressed as

$$\mathcal{D}_{\text{LSQ}} = \frac{1}{N_d} \sum_m^{N_d} \left\| \sqrt{I_{\text{pred},m}} - \sqrt{I_{\text{meas},m}} \right\|^2 \quad (\text{S14})$$

where  $N_d$  is the number of detector (or tile) pixels. If  $I_{\text{meas}}$  and  $I_{\text{pred}}$  are spatially sparse, the intensity terms  $I_k$  in Eq. S14 can be implemented as  $I_k + \epsilon$ , where  $\epsilon$  is a small positive number, to improve numerical robustness, since  $I_k$  appears in the denominator of the square root function's derivative.

Alternatively, when the measured data are noisy, one may also explicitly introduce Poisson statistics to account for the noise in  $I_{\text{meas}}$ , which leads to the Poisson maximum likelihood mismatch [29]:

$$\mathcal{D}_{\text{Poisson}} = \frac{1}{N_d} \sum_m^{N_d} I_{\text{pred},m} - I_{\text{meas},m} \log(I_{\text{pred},m}). \quad (\text{S15})$$

In a previous publication [30], we have shown with numerical simulations that the Poisson loss function of Eq. S15 can provide better resolution and contrast under low-dose conditions compared to the LSQ loss function of Eq. S14, but it takes longer to converge, and may result in indeterministic artifacts for less noisy images. Adorym allows users to conveniently add new loss function types (for example, mixed Gaussian-Poisson) into the parent class of `ForwardModel`.

When processing data subject to photon noise or information deficiency, it is a common practice to supply the reconstruction algorithm with additional prior knowledge about the sample. The finite support constraint, which is commonly used in full-field coherent diffraction imaging [4] and is also made available in Adorym, is one type of such prior knowledge used to ensure the algorithm converges to a unique solution. Alternatively, some other categories involve assumptions on one or more quantitative indicators of the object function, such as its smoothness or sparsity. The  $l_1$ -norm, for example, is known to enhance the sparsity of the solution, encouraging more voxels to have near-zero absolute values [31, 32]. If such a quantity  $\mathcal{R}[O(\mathbf{r})]$  constrained to be below a certain value, the constrained optimization can be equivalently solved by adding  $\mathcal{R}[O(\mathbf{r})]$  to the original loss function of Eq. S14 or Eq. S15 as a Tikhonov regularizer [33]. The full form of the loss function to be minimized is then

$$\mathcal{L} = \mathcal{D}(I_{\text{pred}}, I_{\text{meas}}) + \sum_i \alpha_i \mathcal{R}_i[O(\mathbf{r})] \quad (\text{S16})$$

where  $\alpha_i$  is the ‘‘weighting coefficient’’ applied to regularizer  $i$ . The regularizers already available in Adorym include  $l_1$ -norm [31, 32], reweighted  $l_1$ -norm, and total variation. The expressions, purposes, and example scenarios of application, are shown in Table S1. Moreover, these regularizer functions are defined as individual `Regularizer` classes, and users can conveniently add their own regularizers.

## D.2. Parameter refinement

In Section D, we have shown the incorporation of probe position correction in the forward model, which is done using Fourier shift theorem. In addition to that, Adorym also supports the refinement of a range of experimental variables, and the list can be readily expanded by users upon needed. At present, the following parameters and variables can be refined:

- Probe defocus: in imaging techniques like ptychography, it is often the case where the probe function retrieved from a previous experiment is used to reconstruct a new dataset.

Name	Expression of $\mathcal{R}$	Purpose	Application
$l_1$ -norm	$\frac{1}{N_o} \sum_{i_r} \alpha_1  \delta_{i_r}  + \alpha_2  \beta_{i_r} $ <p>if object is represented by refractive indices, or</p> $\frac{1}{N_o} \sum_{i_r} \alpha_1 \left   O_{i_r}  -  \overline{O}  \right  + \alpha_2  \arg(O_{i_r}) $ <p>if object is represented by complex modulation function.</p>	Promotes sparsity.	Reduce artifacts in objects with finite extent (smaller than the object array being solved) or porous objects.
Reweighted $l_1$ -norm [32]	$\frac{1}{N_o} \sum_{i_r} \alpha_1 W_1  \delta_{i_r}  + \alpha_2 W_2  \beta_{i_r} $ <p>if object is represented by refractive indices, or</p> $\frac{1}{N_o} \sum_{i_r} \alpha_1 W_1 \left   O_{i_r}  -  \overline{O}  \right  + \alpha_2 W_2  \arg(O_{i_r}) $ <p>if object is represented by complex modulation function. In both equations, the quantity <math>W_i</math> as in <math>W_i C </math> is the adaptive weight defined as <math>W_i = \max( C ) / ( C  + \epsilon)</math>. Values of <math>W_i</math> are calculated outside the loss function using “snapshots” of the object function, and are treated as no-gradient constants by AD. Their values are updated once upon a fixed number of minibatches are done.</p>	Promotes sparsity adaptively, so that near-zero voxels are penalized more.	Similar to $l_1$ -norm, and more desirable if a good initial guess is available.
Total variation (TV) [34]	$\frac{1}{N_o} \gamma \sum_{i_r} \ \nabla \delta_{i_r}\ _1 + \ \nabla \beta_{i_r}\ _1$ <p>if object is represented by refractive indices, or</p> $\frac{1}{N_o} \gamma \sum_{i_r} \left( \left   O_{i_r}  -  \overline{O}  \right  + \ \arg(O_{i_r})\ _1 \right)$ <p>if object is represented by complex modulation function. <math>\ \nabla C\ _1</math> returns the <math>l_1</math>-norm of the spatial gradient of <math>C</math>, given as <math> \nabla_x C  +  \nabla_y C  +  \nabla_z C </math>.</p>	Promotes smoothness.	Reconstructing the object from noisy data.

**Table S1.** A list of regularizers available in Adorym. In all equations,  $N_o = L_x L_y L_z$  is the total number of voxels in the object function, and  $i_r$  indexes the object voxels;  $\overline{O}$  is the mean of object modulus  $\arg(\cdot)$  represents the complex argument function which returns the phase of the complex input.

However, as the axial position of the sample may differ, the previously recovered probe might be deviated from the incident plane of the new sample. In this case, Adorym can Fresnel propagate (Eq. S6) the probe function according to a list of defocusing distances,  $z_{\text{defoc}}$ , which stores the defocusing that should be applied for each tilt angle. The gradient of  $z_{\text{defoc}}$  can be automatically calculated by AD for optimization.

- Probe position offset: Section D presented an example where all probe positions are refined individually. In the case of ptychotomography, it is often the case that the relative positions of probe positions on a viewing angle are accurate, but there exist offsets among positions on different angles in the sample frame, such as in the case of a not-well-centered rotation axis [35]. Individually refining all probe positions from all angles might still be feasible, but would introduce too many degrees of freedom, making the reconstruction problem badly underconstrained. Adorym can instead refine a list of position offsets across different tilt angles, where each correction vector as an element of the list applies to all probe positions in a certain angle.
- Slice position: in sparse multislice imaging, slice positions enter the forward model through Fresnel propagation from a slice to the next, and the gradient of slice positions is thus obtainable through AD.
- Free propagation distance: the distance between the exiting plane of the sample to the detector in the case of near-field imaging can be refined as well. In the case of multi-distance holography, the distances from the sample to all projection images can be included in the framework.
- Object orientation: the sample stage in nanotomography may suffer from random wobbling, which causes the angular orientation of the sample to fluctuate in all three axes at different viewing angles. Refinement of orientations has already been demonstrated in combined x-ray ptychography and fluorescence microscopy [36], so orientation refinement has been implemented in Adorym as well. When enabled, the object is rotated in its three axes according to an optimizable orientation correction array of shape [3, num\_angles]. For each axis, the pixel coordinates on each plane of the rotated object perpendicular to the axis are mapped back to the original object function at  $0^\circ$ , using the linear transform

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x_\theta \\ y_\theta \end{pmatrix}. \quad (\text{S17})$$

Subsequently, the rotated object is computed by taking voxel values according to the mapped coordinates from the  $0^\circ$  object array using bilinear interpolation. A way to understand how the loss function can be differentiable with regards to rotation angle  $\theta$  through this interpolation operation is to regard the interpolation as an  $(L_p L_q) \times (L_p L_q)$  matrix  $\mathbf{R}_\theta$  parameterized by  $\theta$ , where  $L_p$  and  $L_q$  are the numbers of pixels along an object plane perpendicular to the rotation axis. It follows that the interpolation is in fact a linear operation of evaluating  $\bar{\sigma}_\theta = \mathbf{R}_\theta \bar{\sigma}_0$ , where  $\bar{\sigma}$  is a vectorized “ $pq$ -plane” of the object, formed by concatenating each row of it. The matrix  $\mathbf{R}_\theta$  is sparse, containing at most 4 non-zero elements on each row in the case of bilinear interpolation; therefore, calculating the derivative of  $\mathbf{R}_\theta$  to  $\theta$  with a gradient vector can be done efficiently by exploiting this sparsity [37]. In Adorym, we implement differentiable rotation by using the `grid_sample` function of *PyTorch* [38, 39].

- Affine transform of projection images: this feature is particularly useful for near-field imaging and especially multi-distance holography. The misalignment of detector or sample stage may cause the acquired projection image to suffer from one or more of three types of distortions, namely translation, tilting, and scaling. These distortions are collectively refined to as the affine transformation. We denote the translation, scaling, and shearing in  $x$  and  $y$  as  $\delta x$ ,  $\delta y$ ,  $S_x$ ,  $S_y$ ,  $c_x$ ,  $c_y$ , respectively, and also represent the tilt angle as  $\phi$ . This leads to an affine transformation matrix  $\mathbf{A}$  which is the composite of 4 matrices responsible for translation, scaling, shearing, and tilt:

$$\mathbf{A} = \begin{pmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & c_x & 0 \\ c_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \delta x \\ 0 & 1 & \delta y \\ 0 & 0 & 1 \end{pmatrix}. \quad (\text{S18})$$

Eq. S18 is apparently an expanded form of the square matrix in Eq. S17 for situations beyond rotation. Adorym can refine an affine transformation matrix for the projection image at each defocus distance to minimize the loss function, and the transformation is differentiable based on the same reason as in the case of sample rotation refinement. The forward model of affine transformation and the associated bilinear interpolation is also implemented using *PyTorch*'s `grid_sample` function.

- Linear relation coefficient between phase and absorption. The homogeneous assumption about the object, which assumes that the  $\delta(\mathbf{r})$  and  $\beta(\mathbf{r})$  terms in its refractive index is related through  $\delta(\mathbf{r}) = (1/\kappa)\beta(\mathbf{r})$  [9], is sometimes desirable when the measured data do not provide enough information to recover both the phase and magnitude of the object function. By enforcing a linear relation between the phase shifting part and the absorption part of the object, the number of unknowns is effectively reduced to half, which could prevent the reconstruction problem from being too loosely constrained. However, since most real samples have multiple materials present, it is hard to accurately calculate the value of  $\kappa$  using prior knowledge about the object. Adorym on the other hand can include  $\kappa$  into the automatic differentiation loop, so that its value can be adjusted constantly to minimize the loss function.

### E. Module Wrapper and backends

The AD engine is the cornerstone of Adorym: it provides the functionality to calculate the derivative of the loss function with regards to the object, the probe, and other parameters. Because the AD engine needs to keep track of all variables (or “nodes” [40]), and the relationship between them and the loss function (this relationship is known as the “graph”), most AD libraries have their own data types and functions for the parent and child nodes in the forward model. Many AD libraries are quite similar: they come with a comprehensive look-up table that logs the derivatives for many common mathematical and array-manipulation operators (like stacking and reshaping), and some provide an easy interface for users to expand the package by defining derivatives for new functions [41]. On the other hand, packages may differ in performance depending on the environment in which they run, in the ease of setting up the environment they require, and in the size of their user communities. *Autograd* builds its data type and functions on the basis of the popular scientific computation package NumPy [42] and SciPy [43]; these in turn can make use of hardware-tuned libraries such as the Intel Math Kernel library [44]. However, *Autograd* does not have built-in support for graphical processing units (GPUs). Another AD package which provides GPU support is *PyTorch*, which has perhaps a larger user community. *Autograd* and *PyTorch* are dynamic graph tools which allow the computational graph or the forward model to be altered at runtime [45]. This means the forward model structure can be modified on the fly using Python’s native workflow control clauses (such as an `if...else...` block), which is very helpful for implementing a multi-stage optimization strategy. In this way one can use one type of loss function for the first several iterations, and switch to another after that. Moreover, a dynamic graph tool means the runtime values of any intermediate variables can be easily retrieved by inserting a print function inside the forward model code. With a static graph tool, this can only be done outside the forward model, which makes debugging less intuitive and straightforward. Therefore, Adorym is designed to specifically work with dynamic graph tools.

As we aim to incorporate both *Autograd* and *PyTorch* (and more dynamic graph AD libraries in the future) in Adorym, we have built a common front end for them in the `wrapper` module. This module provides a unified set of APIs to create optimizable or constant variables (arrays in *Autograd*, or tensors in *PyTorch*), call functions, and compute gradients. In order to make sure all frontend APIs know the right backend to use, we keep the setting parameter `backend` in a specific module named `global_settings`. This common module is imported by reference (instead of by value) in every relevant module of Adorym, and once its value is changed in any module or script, it is changed for all modules after that point. For example, the line `import adorym.global_settings` is used in both `ptychography` and `wrapper`; once it is changed in `ptychography` by `adorym.global_settings.backend = "pytorch"` upon user setting, the value of `adorym.global_settings.backend` is changed in `wrapper` immediately as well, so all frontend functions in `wrapper` referring to the backend setting will use functions from *PyTorch* in this case.

### F. Module Optimizer

Optimizers define how gradients should be used to update the object function or other optimizable quantities; these choices are provided by the class `Optimizer`. Adorym currently provides 4

built-in optimizers: gradient descent (GD), momentum gradient descent, adaptive momentum estimation (Adam), and conjugate gradient (CG). Moreover, Adorym also has a `ScipyOptimizer` class that wraps the `optimize.minimize` module of the Scipy library [43], so that users may access more optimization algorithms coming with Scipy when using the *Autograd* backend on CPU. The simplest GD algorithm updates an optimizable vector  $\mathbf{x}$  using the gradient of the loss function with regards to it, or

$$\mathbf{x} \leftarrow \mathbf{x} - \rho \nabla_{\mathbf{x}} \mathcal{L} \quad (\text{S19})$$

where  $\rho$  is the step size. GD is very low in overhead in that it does not require any additional parameters to be stored and updated. However, when the loss function is ill-conditioned (so that the loss is steep at the vicinity of its minimum) and convex, update iterations can cause it to overshoot the global minimum. As a consequence, the solution may oscillate around the minimum point, taking many iterations to finally converge. One viable workaround is to make  $\rho$  variable. If  $\rho$  is reduced to  $\rho/2$  each time one reduces by half the suboptimality gap  $|\mathcal{L}_j - \mathcal{L}^*|$  between the current loss  $\mathcal{L}_j$ , and the theoretically asymptotic loss value  $\mathcal{L}^*$ , then GD will monotonically converge to the minimum [46]. In practice, we found the actual halving of the suboptimality gap can be tricky to decide as the loss curve sometimes exhibits a periodically oscillating pattern even if the unknowns are still far away from the global minimum. Nevertheless, if  $\rho$  is reduced according to the above mentioned strategy, then the number of iterations needed for the suboptimality gap to drop from  $|\mathcal{L}_j - \mathcal{L}^*|$  to  $|\mathcal{L}_j - \mathcal{L}^*|/2$  is twice the number needed for it to drop from  $2|\mathcal{L}_j - \mathcal{L}^*|$  to  $|\mathcal{L}_j - \mathcal{L}^*|$ . In other words, the number of iterations before which  $\rho$  should be halved doubles each time that halving occurs. Therefore, in our implementation of GD in the `GDoptimizer` child class, users are allowed to define a “base iteration number”  $N_{\text{bi}}$ , so that  $\rho$  is halved each time the iteration index hits  $\{\sum_{s=0}^S 2^s N_{\text{bi}} | S = 0, 1, \dots\}$ .

Another known problem of GD is that it can also oscillate excessively when it travels through a ravine in the solution space. As the gradient vector can sometimes point to the “side walls” of the ravine, so that GD may dangle between the side walls instead of going straight down along the ravine. The momentum algorithm [47] augments GD by adding the history of past gradients into the update vector, so that Eq. S19 becomes

$$\begin{aligned} \mathbf{v} &\leftarrow \gamma \mathbf{v} + \rho \nabla_{\mathbf{x}} \mathcal{L} \\ \mathbf{x} &\leftarrow \mathbf{x} - \mathbf{v}. \end{aligned} \quad (\text{S20})$$

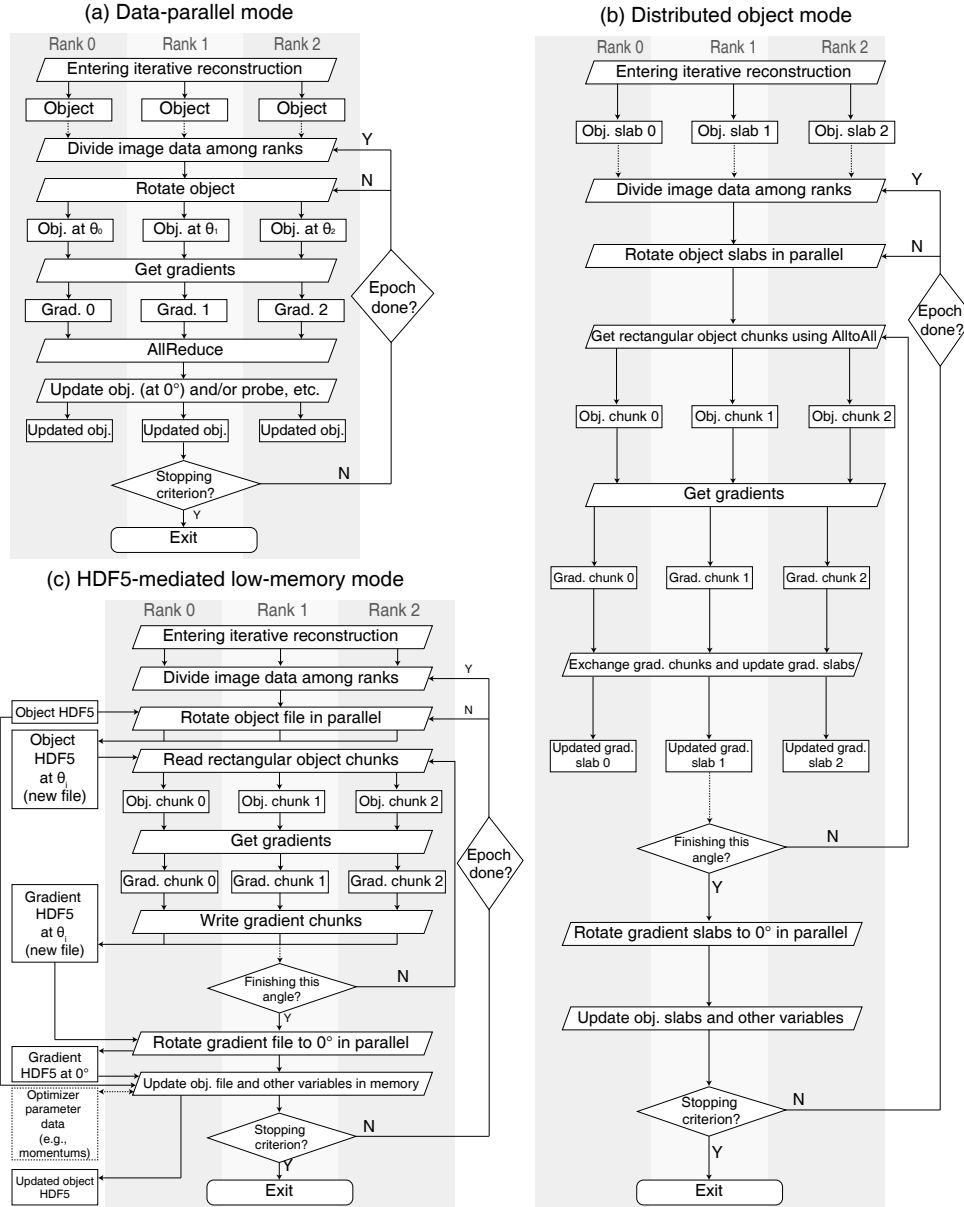
This algorithm is implemented as the `MomentumOptimizer` class of Adorym.

The Adam algorithm [48] makes use of the history of gradients in past iterations, and is among the most popular optimization algorithms in the machine learning community [46]. What distinguishes Adam from many other algorithms in the same category is that it stores both the first-order moment (the mean) and the second-order moment (the variance) of past gradients, and that it involves a correction step for countering the bias of the momenta towards zero. These features provide an advantage in comparison to similar momentum-based algorithms [46]. Another merit of Adam in our specific application is that the magnitude of its update vector is relatively insensitive to the scale of gradient [48], so it can work well with raw data that have various numerical scalings. As an example, full-field tomography and holography data are usually normalized using “white field” images, while far-field ptychography data generally have a high dynamic range and are often used without normalization. Because the update rate in Adam can be imperfectly tuned even by several orders of magnitude, Adam can also work with different types of loss functions which may have very different values. This aligns well with the generalizability concept of Adorym, so we have implemented the algorithm in the `AdamOptimizer` class.

Our implementations of GD, Momentum, and Adam are designed for stochastic minibatch optimization – in the literature, these are the algorithms of choice for when we have large data sets [46]. For small or medium-sized problems, when the raw data and computational graph fit into the memory, more sophisticated (and hyperparameter-free) conjugate gradient algorithms or higher order algorithms are typically the methods of choice. In Adorym, we implement the non-linear conjugate gradient (CG) method [49, 50], which is implemented as the `CGoptimizer` in Adorym. Our implementation uses the Polak–Ribière method [51, 52] to find the conjugate gradient vectors, followed by an adaptive line search [53] to determine the update step size. We also implement a `ScipyOptimizer` wrapper that can be used to access a variety of sophisticated algorithms (such as BFGS, Newton-CG, and so on) from *SciPy* within Adorym. One distinction is that while our GD, Momentum, Adam, and CG implementations can be used in both CPUs

and GPUs, the Scipy optimizers can only be used in the CPU. However, for small-sized convex problems, this can often be balanced by the reduced number of iterations that some advanced algorithms require [54].

### G. Parallelization modes



**Fig. S3.** Workflow diagram of Adorym in DP mode, DO mode, and H5 mode.

Adorym supports parallelized processing based on the message passing interface (MPI) [55], a multiprocessing programming model that is compatible with most modern computation platforms ranging from laptops to HPCs. Specifically, in Adorym we use MPI4py Python API [56]. The MPI interface allows one to conveniently communicate data or synchronize program workflow among multiple processes (or ranks) on CPU cores. Because each spawned MPI rank is independent from others on the Python level, it can individually send data back and forth between RAM and a specified GPU, and run GPU operations without interfering with other ranks as long as the GPU memory does not overflow. Therefore, our MPI-based parallelization is



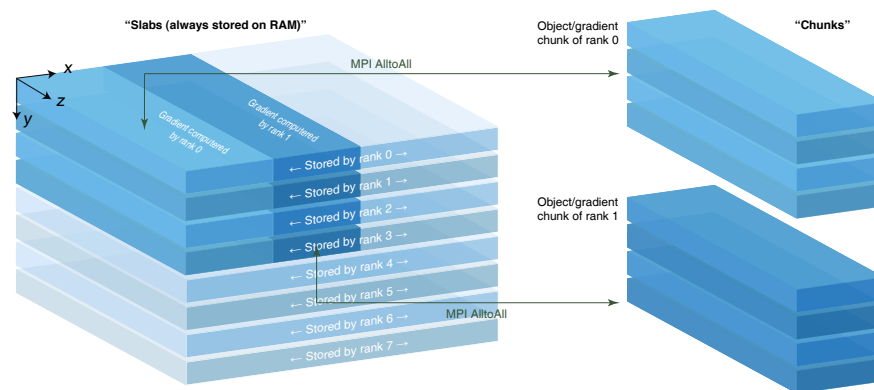
also inherently capable of working with an arbitrary number of GPUs. When it is necessary to communicate between all ranks, each rank needs to retrieve the data back from the GPU to the CPU, and use collective MPI operations such as Allreduce. This is inefficient compared to direct GPU-to-GPU communications using tools such as NVLink [57], but it works on a wider range of devices.

Parallel processing based on MPI can be implemented in several different ways, each of which differs from others in terms of computational overhead and memory consumption. Here we introduce the three types of parallelization modes used in Adorym: data parallelism (DP) mode, distributed object (DO) mode, and the HDF5-file-mediated low-memory (H5) mode.

### G.1. Data parallelism (DP) mode

In machine learning, “data parallelism” refers to distributed neural network training where each rank has a copy of the full model stored in memory, and process a subset of training data that is distinct from that on other ranks [58]. The model parameters are periodically synchronized by taking their averages over all ranks. In Adorym, the equivalent of a neural network is the forward optical model, where the parameters are the object function and all optimizable variables. When working in DP mode, each rank is allocated with a batch of `batch_size` tiles, which is used to calculate the gradient of the loss function with regards to the entire object volume and other parameters (probes, probe positions, *etc.*). Once all ranks have finished their gradient computation, their gradient arrays are synchronized to the same average values through an MPI Allreduce operation, after which they are used to update the optimizable variables using a chosen optimization algorithm. Since gradient averaging is the only major inter-rank communication, the DP mode is low in overhead, but it has high memory consumption due to the need of keeping a copy of the whole object function for every rank. This in turn limits the ability to reconstruct objects with limited resources. A illustrative workflow diagram for the DP mode with 3 MPI ranks is shown in Fig. S3(a).

### G.2. Distributed object (DO) mode



**Fig. S4.** Illustration of the distributed object (DO) scheme. With multiple MPI ranks, the object is divided into a vertical stack of slabs, which are kept separately by all ranks. When a rank processes a diffraction image, it gets data from ranks that possess the parts of the object function that it needs, after which it assembles these partial slabs into the object chunk corresponding to the beam path of the diffraction image that it is processing. After gradient of the chunk is calculated, it is scattered back into the same positions of the gradient slabs kept by relevant ranks. Object update is done by each rank individually using the gathered gradient array.

In spite of its simplicity and low overhead, keeping a copy of the object function for every rank as in the DP mode can be impractical with limited memory per rank. We have therefore implemented the distributed object (DO) mode as an alternative option. In DO mode, only 1 object function is jointly kept by all ranks. This is done by severing the object along the vertical axis into several slabs, and letting each rank store one of them in its available RAM as 32-bit floating point numbers. In this way, standard tomographic rotation can be done independently

by all ranks in parallel. The stored object slab does not enter the device memory as a whole when GPU acceleration is enabled; instead, object chunks are extracted from the object slabs in the CPU, and only these partial chunks are sent to GPU memory for gradient computation. Since the “thickness” of a slab is usually different from the vertical size of an object chunk (whose  $x$ - $y$  cross sections should match the size of a tile), we use MPI’s AlltoAll communication for a rank to gather the voxels it needs to assemble an object chunk from other ranks.

This is illustrated in Fig. S4, where we assume the MPI command spawns 8 ranks, and the object function is evenly distributed among these ranks. For simplicity, we assume a batch size of 1, meaning a rank processes only 1 tile at a time. Before entering the differentiation loop, the object slab kept by each rank is duplicated as a new array, which is then rotated to the currently processed viewing angle. In our demonstrative scenario, rank 0 processes the top left tile in the first iteration, and the vertical size of the tile spans four object slabs. As ranks 0–3 have knowledge about rank 0’s job assignment, they extract the part needed by rank 0 from their own slabs, and send them to rank 0 through AlltoAll operation. Rank 0 then assembles the object chunk from the partial slabs by concatenating them along  $y$  in order. Besides, since rank 0 itself contains the object slab needed by other ranks, it also needs to send information to them. The collective send/receive can be done using MPI’s AlltoAll communication in a single step. In reality, the vertical size of an object chunk might not be divisible by the number of slices in each slab, so certain ranks may send a partial object that is “thinner” than the slab it keeps. The ranks then send the assembled object chunks to their assigned GPUs for gradient computation, yielding gradient chunks on each rank that have exactly the same shape as the object chunks. These gradient chunks are scattered back to relevant ranks, but this time to their “gradient slab” arrays that identically match the object slabs in position and shape. After all tiles on this certain viewing angle are processed, the gradient slabs kept by all ranks are rotated back to  $0^\circ$ . Following that, the object slab is updated by the optimizer independently in each rank. When reconstruction finishes, the object slabs stored by all ranks are dumped to a RAM-buffered hard drive, so that they can be stacked to form the full object. The optimization workflow of the DO mode is shown in Fig. S3(b).

While it has the advantage of requiring less memory per rank, a limitation of the DO mode is that tilt refinement about the  $x$ - and  $z$ -axis is hard to implement; tilting about these axes requires a rank to get voxel values from other slabs, which not only induces excessive MPI communication, but also demands AD to differentiate through MPI operations. The latter is not impossible, but existing AD packages may need to be modified in order to add that feature. Another possible approach among the slabs kept by different MPI ranks is to introduce overlapping regions along  $y$ ; in this case, tilting about  $x$  and  $z$  can be done individually by each rank, though the degree of tilt is limited by the length of overlap. This is not yet implemented in Adorym, but could be added in the future.

### **G.3. HDF5-file-mediated low-memory mode (H5)**

When running on an HPC with hundreds or thousands of computational nodes, the DO mode can significantly reduce the memory needed by each node to store the object function if one uses many nodes (and only a few ranks per node) to distribute the object. On a single workstation or laptop, however, the total volume of RAM is fixed, and very large-scale problems are still difficult to solve even if one uses DO.

For such cases, Adorym comes with an alternative option of storing the full object function in a parallel HDF5 file [1], which is referred to as the H5 mode. The HDF5 library works with the MPI-IO driver to allow a file to be read or written by several MPI ranks, where any modifications to the file’s metadata are done collectively by all ranks to avoid potential conflicts. This feature provides a viable scheme of further reducing the RAM usage of Adorym. At the beginning of the reconstruction job, the object function is initialized and saved as an HDF5 file. Before entering the differentiation loop, the object function is duplicated as a new dataset in the same HDF5 file, and rotated to the viewing angle being processed in a similar fashion as the DO mode: the rotation is done in parallel by all ranks, with each rank handling several  $y$ -slices. Following that, object chunks are read from the rotated object by each rank, and sent to GPU if GPU acceleration is enabled. The calculated gradient chunks are written by each rank into a separate gradient dataset, which, after all tiles on the current angle are processed, is rotated back to  $0^\circ$  collectively. To update the object function, each rank reads in a slice of the object and the corresponding gradient at a time, performs update using the selected optimizer, then writes the updated object slice back to the HDF5 file. The next slice is then processed, until the entire object is updated jointly by all ranks. The workflow [Fig. S3(c)] in fact resembles that of the DO mode, except the distribution of

object chunks and synchronization of gradient chunks is done through HDF5 file reading and writing instead of using MPI AlltoAll.

While the H5 mode allows the reconstruction of large objects on limited memory machines, I/O with a hard drive (even with a solid state drive) is slower than memory access. Additionally, writing into an HDF5 with multiple ranks is subject to contention, which may be mitigated if a parallel file system with multiple object storage targets (OSTs) is available [59]. This is more likely to be available at an HPC facility than on a smaller, locally managed system, and precise adjustment of striping size and HDF5 chunking are needed to optimize OST performance [60]. Despite these challenges, the HDF5 mode is valuable because it enables reconstructions of large objects and/or complex forward models on limited memory machines. It also provides future-proofing because, as fourth-generation synchrotron facilities deliver higher brightness and enable one to image very thick samples [61], we may eventually encounter extra-large objects which might be difficult to reconstruct even in existing HPC machines.

With all the above descriptions, we summarize in Fig. S1 the inter-relations among different modules as parts of Adorym's software architecture.

#### G.4. User interface

If a reconstruction job can be done using the provided `ForwardModel` classes, users generally just need to call the `reconstruct_ptychography` function inside the `ptychography`. Optimizers for different variables can either be explicitly declared using the child classes of `Optimizer`, or be specified by providing the optimizer name (for object function) and step size while using default values for other parameters. Below is a code example that can be used to generate the fly-scan ptychography results shown in the main text:

```
import adorym
from adorym.ptychography import reconstruct_ptychography

output_folder = "recon"
distribution_mode = None
optimizer_obj = adorym.AdamOptimizer("obj", output_folder=output_folder,
                                     distribution_mode=distribution_mode,
                                     options_dict={"step_size": 1e-3})
optimizer_probe = adorym.AdamOptimizer("probe", output_folder=output_folder,
                                       distribution_mode=distribution_mode,
                                       options_dict={"step_size": 1e-3, "eps": 1e-7})
optimizer_all_probe_pos = adorym.AdamOptimizer("probe_pos_correction",
                                               output_folder=output_folder,
                                               distribution_mode=distribution_mode,
                                               options_dict={"step_size": 1e-2})

params_ptych = {"fname": "data.h5",
               "theta_st": 0,
               "theta_end": 0,
               "n_epochs": 1000,
               "obj_size": (618, 606, 1),
               "two_d_mode": True,
               "energy_ev": 8801.121930115722,
               "psize_cm": 1.32789376566526e-06,
               "minibatch_size": 35,
               "output_folder": output_folder,
               "cpu_only": False,
               "save_path": ".",
               "initial_guess": None,
               "random_guess_means_sigmas": (1., 0., 0.001, 0.002),
               "probe_type": "aperture_defocus",
               "forward_model": adorym.PtychographyModel,
               "n_probe_modes": 5,
               "aperture_radius": 10,
               "beamstop_radius": 5,
               "probe_defocus_cm": 0.0069,
               "rescale_probe_intensity": True,
               "free_prop_cm": "inf",
               "backend": "pytorch",
               "raw_data_type": "intensity",
               "optimizer": optimizer_obj,
```

```

        "optimize_probe": True,
        "optimizer_probe": optimizer_probe,
        "optimize_all_probe_pos": True,
        "optimizer_all_probe_pos": optimizer_all_probe_pos,
        "save_history": True,
        "unknown_type": "real_imag",
        "loss_function_type": "lsq",
    }

reconstruct_ptychography(**params_ptych)

```

In the code example, we passed `adorym.PtychographyModel` to `"forward_model"`. This argument can be replaced by a user-defined `ForwardModel` class whenever desirable. Instructions on creating new forward models and defining new refinable parameters can be found in the documentation of Adorym, which is currently hosted on the GitHub repository (<https://github.com/mdw771/adorym/>).

## 2. DEFINITION OF STRUCTURAL SIMILARITY INDEX (SSIM)

The SSIM [62] is a metric that measures the “degree of match” between two images, and, unlike pixel-to-pixel error metrics such as the mean squared error, it takes into account the interdependence among pixels lying in their local neighborhoods. For reconstructed image  $I_a$  and reference image  $I_r$ , the SSIM is computed as

$$\text{SSIM}(I_a, I_r) = l(I_a, I_r) \cdot c(I_a, I_r) \cdot s(I_a, I_r). \quad (\text{S21})$$

where  $l$ ,  $c$ , and  $s$  respectively gauge the similarity of the images in terms of luminance, contrast, and structure. These factors are defined as

$$l(I_a, I_r) = \frac{2\mu_a\mu_r + c_1}{\mu_a^2 + \mu_r^2 + c_1} \quad (\text{S22})$$

$$c(I_a, I_r) = \frac{2\sigma_a\sigma_r + c_2}{\sigma_a^2 + \sigma_r^2 + c_2} \quad (\text{S23})$$

$$s(I_a, I_r) = \frac{\sigma_{a,r} + c_3}{\sigma_a\sigma_r + c_3} \quad (\text{S24})$$

where  $\mu_a$  and  $\mu_r$  are the mean value of the reconstructed image and the reference image,  $\sigma_a$  and  $\sigma_r$  are their standard deviations, and  $\sigma_{a,r}$  is their correlation coefficient. The parameters appearing in the equations above are given by

$$c_1 = (k_1 L)^2 \quad (\text{S25})$$

$$c_2 = (k_2 L)^2 \quad (\text{S26})$$

$$c_3 = c_2/2 \quad (\text{S27})$$

where  $k_1$  and  $k_2$  are set to 0.01 and 0.03 in our case, and  $L$  is the dynamic range of the grayscale images.

## REFERENCES

1. The HDF Group, “Hierarchical Data Format, version 5,” [Http://www.hdfgroup.org/HDF5/](http://www.hdfgroup.org/HDF5/).
2. F. De Carlo, D. Gürsoy, F. Marone, M. Rivers, D. Y. Parkinson, F. Khan, N. Schwarz, D. J. Vine, S. Vogt, S.-C. Gleber, S. Narayanan, M. Newville, T. Lanzirotti, Y. Sun, Y. P. Hong, and C. Jacobsen, “Scientific Data Exchange: a schema for HDF5-based storage of raw and analyzed data,” *J. Synchrotron Radiat.* **21**, 1224–1230 (2014).
3. P. Cloetens, W. Ludwig, J. Baruchel, J. Baruchel, D. Van Dyck, J. Van Landuyt, J. P. Guigay, and M. Schlenker, “Holotomography: Quantitative phase tomography with micrometer resolution using hard synchrotron radiation x rays,” *Appl. Phys. Lett.* **75**, 2912–2914 (1999).
4. S. Marchesini, H. He, H. Chapman, S. Hau-Riege, A. Noy, M. Howells, U. Weierstall, and J. Spence, “X-ray image reconstruction from a diffraction pattern alone,” *Phys. Rev. B* **68**, 140101 (2003).

5. B. L. Henke, E. M. Gullikson, and J. C. Davis, "X-ray interactions: Photoabsorption, scattering, transmission, and reflection at  $E=50\text{--}30,000$  eV,  $Z=1\text{--}92$ ," *At. Data Nucl. Data Tables* **54**, 181–342 (1993).
6. J. M. Rodenburg, "Ptychography and related diffractive imaging methods," *Adv. Imaging Electron Phys.* **150**, 87–184 (2008).
7. R. M. Goldstein, H. A. Zebker, and C. L. Werner, "Satellite radar interferometry: Two-dimensional phase unwrapping," *Radio Sci.* **23**, 713–720 (1988).
8. H. Xia, S. Montresor, R. Guo, J. Li, F. Yan, H. Cheng, and P. Picart, "Phase calibration unwrapping algorithm for phase data corrupted by strong decorrelation speckle noise," *Opt. Express* **24**, 28713–28730 (2016).
9. L. D. Turner, B. B. Dhal, J. P. Hayes, A. P. Mancuso, K. A. Nugent, D. Paterson, R. E. Scholten, C. Q. Tran, and A. G. Peele, "X-ray phase imaging: Demonstration of extended conditions with homogeneous objects," *Opt. Express* **12**, 2960–2965 (2004).
10. Z. Fabian, J. Haldar, R. Leahy, and M. Soltanolkotabi, "3D phase retrieval at nano-scale via accelerated Wirtinger flow," *arXiv*, arXiv:2002.11785 (2020).
11. M. Du, Y. S. G. Nashed, S. Kandel, D. Gürsoy, and C. Jacobsen, "Three dimensions, two microscopes, one code: automatic differentiation for x-ray nanotomography beyond the depth of focus limit," *Sci. Adv.* **6**, eaay3700 (2020).
12. C. Jacobsen, *X-ray Microscopy* (Cambridge University, 2020).
13. J. W. Goodman, *Introduction to Fourier Optics* (W.H. Freeman, 2017).
14. D. Voelz, *Computational Fourier optics: a Matlab tutorial*, Tutorial text (Society of Photo-Optical Instrumentation Engineers, 2011).
15. J. M. Cowley and A. F. Moodie, "The scattering of electrons by atoms and crystals. I. a new theoretical approach," *Acta Crystallogr.* **10**, 609–619 (1957).
16. K. Ishizuka and N. Uyeda, "A new theoretical and practical approach to the multislice method," *Acta Crystallogr. A* **33**, 740–749 (1977).
17. J. Van Roey, J. van der Donk, and P. E. Lagasse, "Beam-propagation method: Analysis and assessment," *J. Opt. Soc. Am.* **71**, 803–810 (1981).
18. A. M. Maiden, M. J. Humphry, and J. M. Rodenburg, "Ptychographic transmission microscopy in three dimensions using a multi-slice approach," *J. Opt. Soc. Am. A* **29**, 1606–1614 (2012).
19. A. Suzuki, S. Furutaku, K. Shimomura, K. Yamauchi, Y. Kohmura, T. Ishikawa, and Y. Takahashi, "High-resolution multislice x-ray ptychography of extended thick objects," *Phys. Rev. Lett.* **112**, 053903 (2014).
20. E. H. R. Tsai, I. Usov, A. Diaz, A. Menzel, and M. Guizar-Sicairos, "X-ray ptychography with extended depth of field," *Opt. Express* **24**, 29089–29108 (2016).
21. U. S. Kamilov, I. N. Papadopoulos, M. H. Shoreh, A. Goy, C. Vonesch, M. Unser, and D. Psaltis, "Learning approach to optical tomography," *Optica* **2**, 517–522 (2015).
22. U. S. Kamilov, I. N. Papadopoulos, M. H. Shoreh, A. Goy, C. Vonesch, M. Unser, and D. Psaltis, "Optical tomographic image reconstruction based on beam propagation and sparse regularization," *IEEE Transactions on Comput. Imaging* **2**, 59–70 (2016).
23. D. Blinder and T. Shimobaba, "Efficient algorithms for the accurate propagation of extreme-resolution holograms," *Opt. Express* **27**, 29905–29915 (2019).
24. S. Ali, M. Du, M. Adams, B. Smith, and C. Jacobsen, "Comparison of distributed memory algorithms for x-ray wave propagation in inhomogeneous media," *Opt. Express* **28**, 29590–29618 (2020).
25. P. Thibault and A. Menzel, "Reconstructing state mixtures from diffraction measurements," *Nature* **494**, 68–71 (2013).
26. P. M. Pelz, M. Guizar-Sicairos, P. Thibault, I. Johnson, M. Holler, and A. Menzel, "On-the-fly scans for x-ray ptychography," *Appl. Phys. Lett.* **105**, 251101 (2014).
27. J. Deng, Y. S. G. Nashed, S. Chen, N. W. Phillips, T. Peterka, R. Ross, S. Vogt, C. Jacobsen, and D. J. Vine, "Continuous motion scan ptychography: characterization for increased speed in coherent x-ray imaging," *Opt. Express* **23**, 5438–5451 (2015).
28. X. Huang, K. Lauer, J. N. Clark, W. Xu, E. Nazaretski, R. Harder, I. K. Robinson, and Y. S. Chu, "Fly-scan ptychography," *Sci. Reports* **5**, 9074 (2015).
29. P. Godard, M. Allain, V. Chamard, and J. M. Rodenburg, "Noise models for low counting rate coherent diffraction imaging," *Opt. Express* **20**, 25914–25934 (2012).
30. M. Du, D. Gürsoy, and C. Jacobsen, "Near, far, wherever you are: simulations on the dose efficiency of holographic and ptychographic coherent imaging," *J. Appl. Crystallogr.* **53**,

- 748–759 (2020).
31. R. Tibshirani, “Regression shrinkage and selection via the lasso,” *J. Royal Stat. Soc. B* **59**, 267–288 (1996).
  32. E. J. Candès, M. B. Wakin, and S. P. Boyd, “Enhancing sparsity by reweighted  $\ell_1$  minimization,” *J. Fourier Analysis Appl.* **14**, 877–905 (2008).
  33. M. T. McCann and M. Unser, “Biomedical image reconstruction: From the foundations to deep neural networks,” *arXiv*, arXiv:1901.03565 (2019).
  34. Grasmair, Markus and Lenzen, Frank, “Anisotropic total variation filtering,” *Appl. Math. & Optim.* **62**, 323–339 (2010).
  35. M. Holler, A. Diaz, M. Guizar-Sicairos, P. Karvinen, E. Färm, E. Härkönen, M. Ritala, A. Menzel, J. Raabe, and O. Bunk, “X-ray ptychographic computed tomography at 16 nm isotropic 3D resolution,” *Sci. Reports* **4**, 3857 (2014).
  36. J. Deng, Y. H. Lo, M. Gallagher-Jones, S. Chen, A. Pryor, Q. Jin, Y. P. Hong, Y. S. G. Nashed, S. Vogt, J. Miao, and C. Jacobsen, “Correlative 3D x-ray fluorescence and ptychographic tomography of frozen-hydrated green algae,” *Sci. Adv.* **4**, eaau4548 (2018).
  37. W. Jiang, W. Sun, A. Tagliasacchi, E. Trulls, and K. M. Yi, “Linearized multi-sampling for differentiable image transformation,” *arXiv*, arXiv:1901.07124 (2019).
  38. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems* **32**, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds. (Curran Associates, 2019), pp. 8024–8035.
  39. “PyTorch documentation,” (PyTorch, 2020). <https://pytorch.org/docs>.
  40. C. C. Margossian, “A review of automatic differentiation and its efficient implementation,” *arXiv*, arXiv:1811.05031 (2018).
  41. “Autograd tutorial,” (2020). <https://github.com/HIPS/autograd/blob/master/docs/tutorial.md>.
  42. S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy array: A structure for efficient numerical computation,” *Comput. Sci. & Eng.* **13**(2), 22–30 (2011).
  43. P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “SciPy 1.0: Fundamental algorithms for scientific computing in Python,” *Nat. Methods* **17**, 261–272 (2020).
  44. E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *High-Performance Computing on the Intel Xeon Phi* (Springer International Publishing, 2014).
  45. M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, “Deep learning with dynamic computation graphs,” *arXiv*, arXiv:1702.02181 (2017).
  46. S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv*, arXiv:1609.04747 (2016).
  47. N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural Networks* **12**, 145–151 (1999).
  48. D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv*, arXiv:1412.6980 (2014).
  49. M. R. Hestenes and E. Stiefel, “Methods of conjugate gradients for solving linear systems,” *J. Res. Natl. Bureau Standards* **49**, 409–436 (1952).
  50. R. Fletcher and C. M. Reeves, “Function minimization by conjugate gradients,” *The Comput. J.* **7**, 149–154 (1964).
  51. E. Polak and G. Ribiere, “Note sur la convergence de méthodes de directions conjuguées,” *Modélisation Mathématique et Analyse Numérique* **3**, 35–43 (1969).
  52. B. Polyak, “The conjugate gradient method in extremal problems,” *USSR Comput. Math. Math. Phys.* **9**, 94–112 (1969).
  53. J. Nocedal and S. J. Wright, *Numerical Optimization* (Springer, 2006).
  54. L. Bottou, F. E. Curtis, and J. Nocedal, “Optimization methods for large-scale machine learning,” *arXiv*, arXiv:1606.04838 (2016).
  55. Message Passing Interface Forum, “A message-passing interface standard, version 3.1” (2015). <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.



56. L. Dalcín, R. Paz, and M. Storti, "MPI for Python," *J. Parallel Distributed Comput.* **65**, 1108–1115 (2005).
57. D. Foley and J. Danskin, "Ultra-performance pascal GPU and NVLink interconnect," *IEEE Micro* **37**, 7–17 (2017).
58. E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and principles of distributed machine learning on big data," *Engineering* **2**, 179–195 (2016).
59. B. Behzad, J. Huchette, H. V. T. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat, "A framework for auto-tuning HDF5 applications," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, (Association for Computing Machinery, 2013), pp. 127–128.
60. M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf, "Tuning HDF5 for Lustre file systems," <https://digital.library.unt.edu/ark:/67531/metadc837191/>.
61. M. Eriksson, J. F. van der Veen, and C. Quitmann, "Diffraction-limited storage rings – a window to the science of tomorrow," *J. Synchrotron Radiat.* **21**, 837–842 (2014).
62. Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE T. Image Process.* **13**, 600–612 (2004).