nature methods

SUPPLEMENTARY INFORMATION

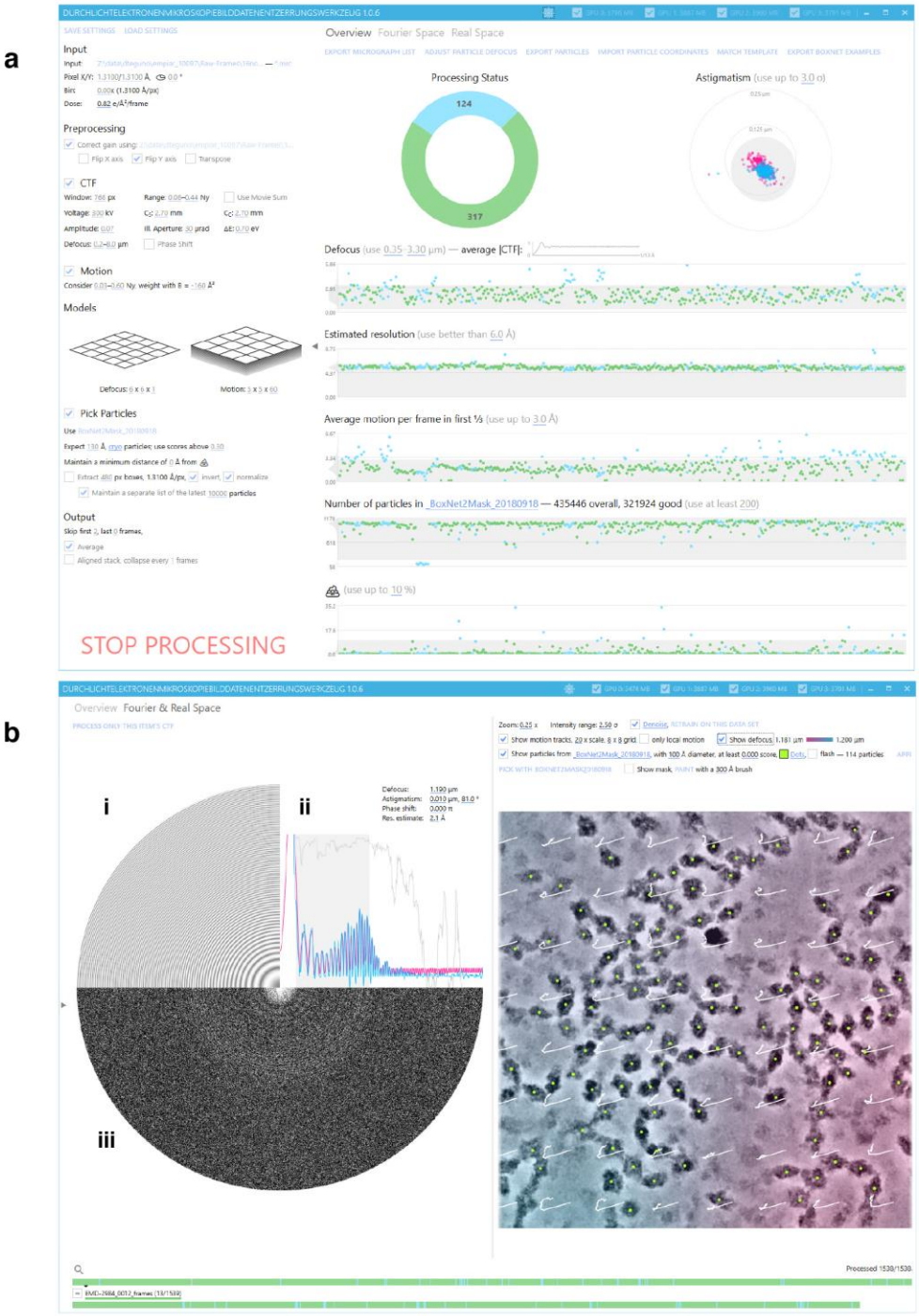
https://doi.org/10.1038/s41592-019-0580-y

In the format provided by the authors and unedited.

Real-time cryo-electron microscopy data preprocessing with Warp

Dimitry Tegunov and Patrick Cramer

Max Planck Institute for Biophysical Chemistry, Department of Molecular Biology, Göttingen, Germany. e-mail: dteguno@mpibpc.mpg.de; patrick.cramer@mpibpc.mpg.de



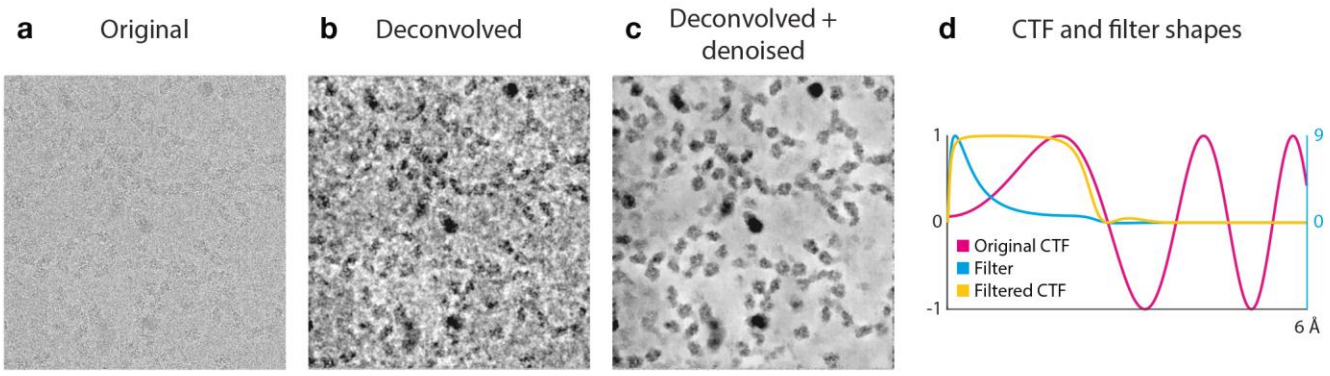
Supplementary Figure 1

User interface of Warp

(a) The processing settings (left) specify all steps and parameters for online data evaluation, correction and processing. The 'Overview' tab (right) presents all important processing results and lets the user specify selection filters to remove low-quality data.

(b) View of a single micrograph. In Fourier space (left), the simulated 2D CTF (i), the 1D power spectrum (PS) and its fit (ii), and the 2D PS (iii) are presented. The real space view (right) shows the aligned movie average with particle positions (green dots), motion tracks (white curves) and the defocus variation (transparent magenta-cyan overlay), and applies a deconvolution filter as well as denoising.

Individual display elements can be shown or hidden. The navigation bar (bottom) shows the processing status for all items and allows to quickly switch between them as well as to manuallyexclude single items from processing.



Supplementary Figure 2

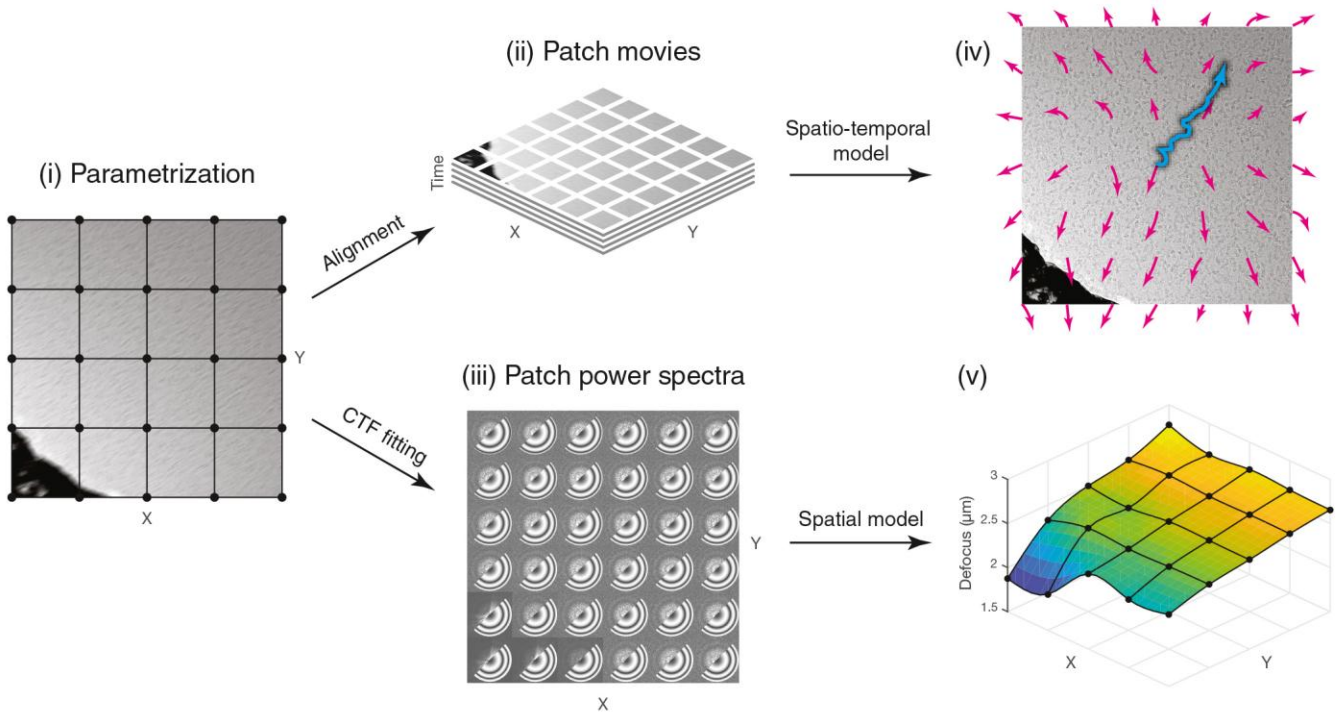
Deconvolution and denoising of a low-defocus micrograph

(a) A raw micrograph from EMPIAR-10061 acquired at 0.8 µm defocus.

(b) Same micrograph after applying deconvolution. Low-resolution contrast is boosted and the defocused signal is more localized, allowing to distinguish the particles better.

(c) Same micrograph after applying deconvolution and denoising with a noise2noise model retrained on this data set. The shapes of individual 400 kDa proteins nearly invisible in the raw image can be distinguished clearly against the background.

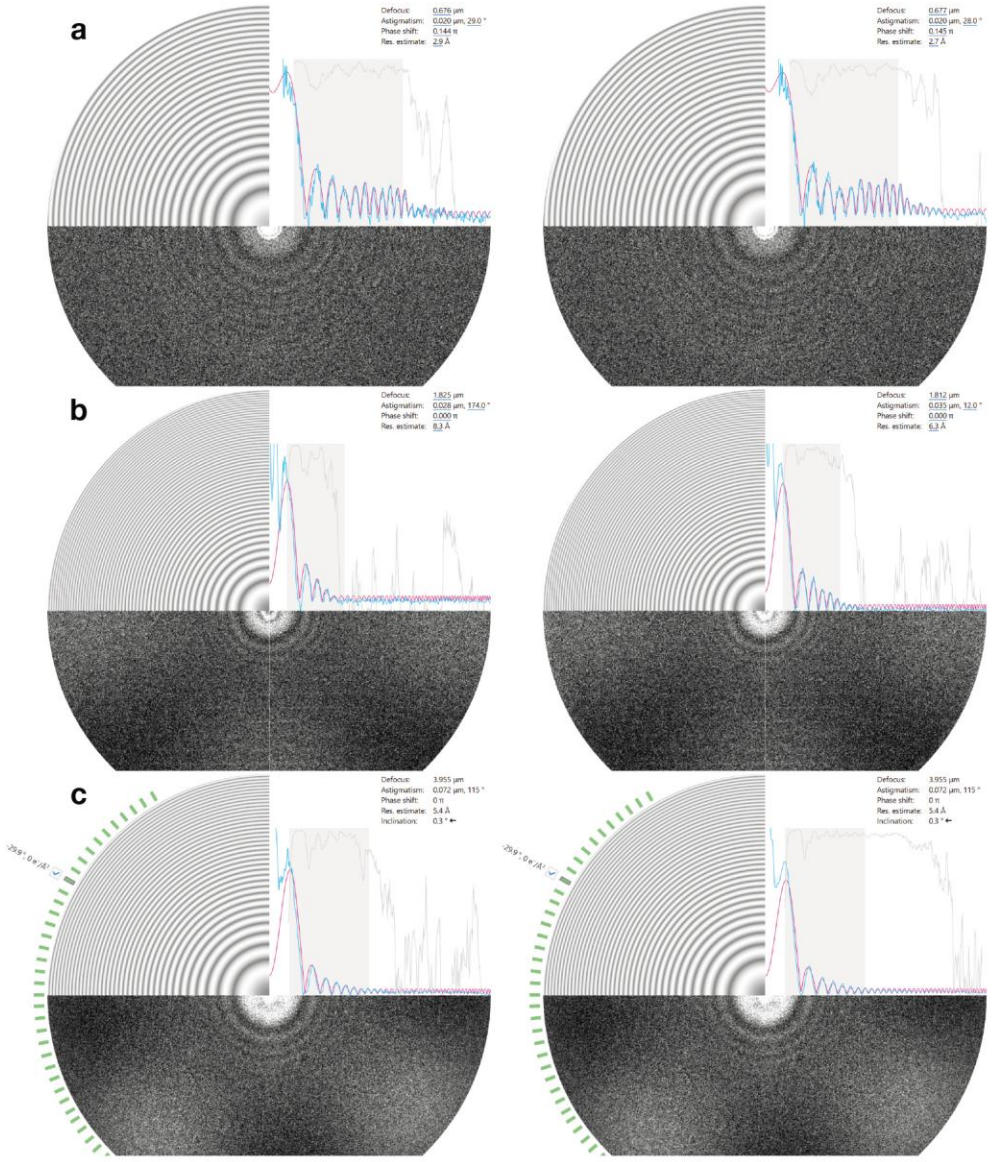
(d) Shape and effect of the deconvolution filter. The filter largely reverses the effect of the first CTF peak, while also suppressing the lowest and higher frequencies.



Supplementary Figure 3

Motion and CTF model fitting by Warp

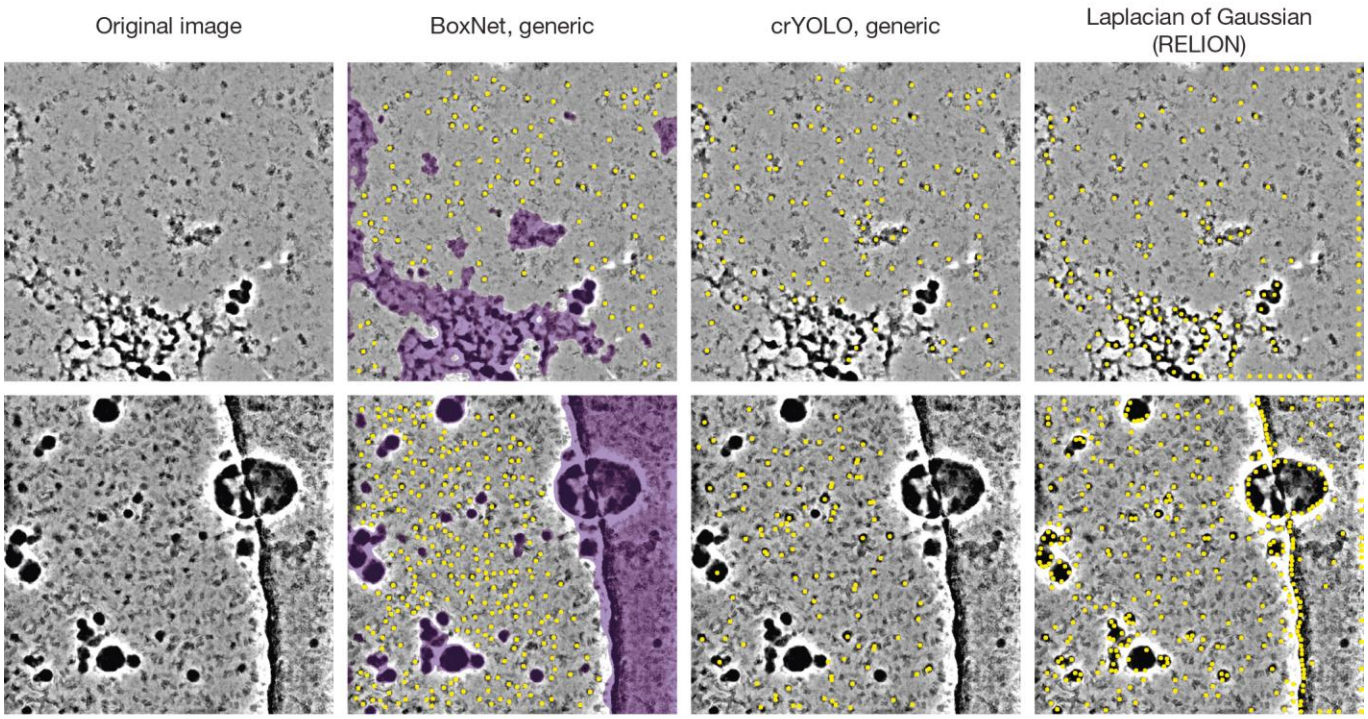
The unaligned, defocused movie (i) is parametrized with a coarse grid (black dots), divided into patches for the alignment (ii), and power spectra of these patches are computed (iii) for CTF fitting. The motion model (iv) includes 2 components: global motion (cyan trajectory) with fine temporal and no spatial resolution, and local motion (magenta trajectories) with coarse temporal, and fine spatial resolution. Both components are optimized to minimize the squared difference between the individual patch frames and their aligned average. The spatially resolved CTF model (v) is optimized to minimize the squared difference between the power spectra (iii, upper left part of each patch) and the simulated local 2D CTF (iii, bottom right part of each patch). Here, the defocus gradient follows the 40° tilt of the specimen, with the notable exception of the hole edge in the bottom left corner.



Supplementary Figure 4

CTF fitting of flat, tilted and tilt series data

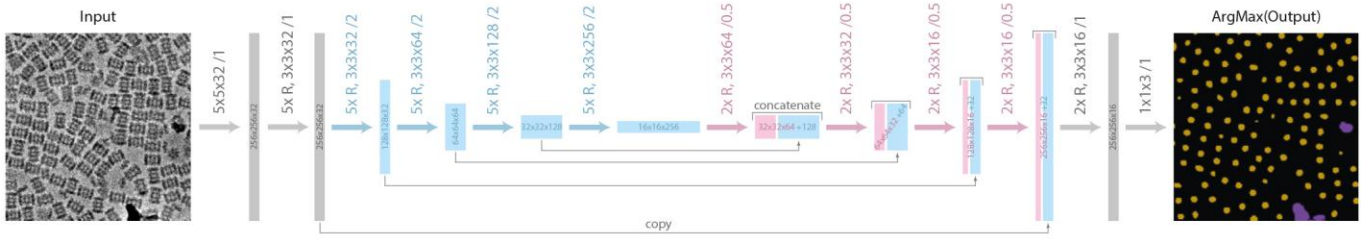
Fitted spectra without (left column) and with (right column) a spatially resolved model. The samples are (a) flat (EMPIAR-10078), (b) tilted at 40 ° (EMPIAR-10097), (c) a tilt series ranging from -60 ° to +60 ° (EMPIAR-10045). In all three cases, using a spatially resolved model allowed to fit the sample geometry more accurately, as evidenced by the clearer Thon rings in the rescaled, averaged 1D spectra. The fitting range (grey rectangle in the 1D spectra) was chosen well below the estimated resolution to avoid overfitting the higher number of parameters in the spatially resolved model.



Supplementary Figure 5

Unbiased particle picking with Warp's BoxNet

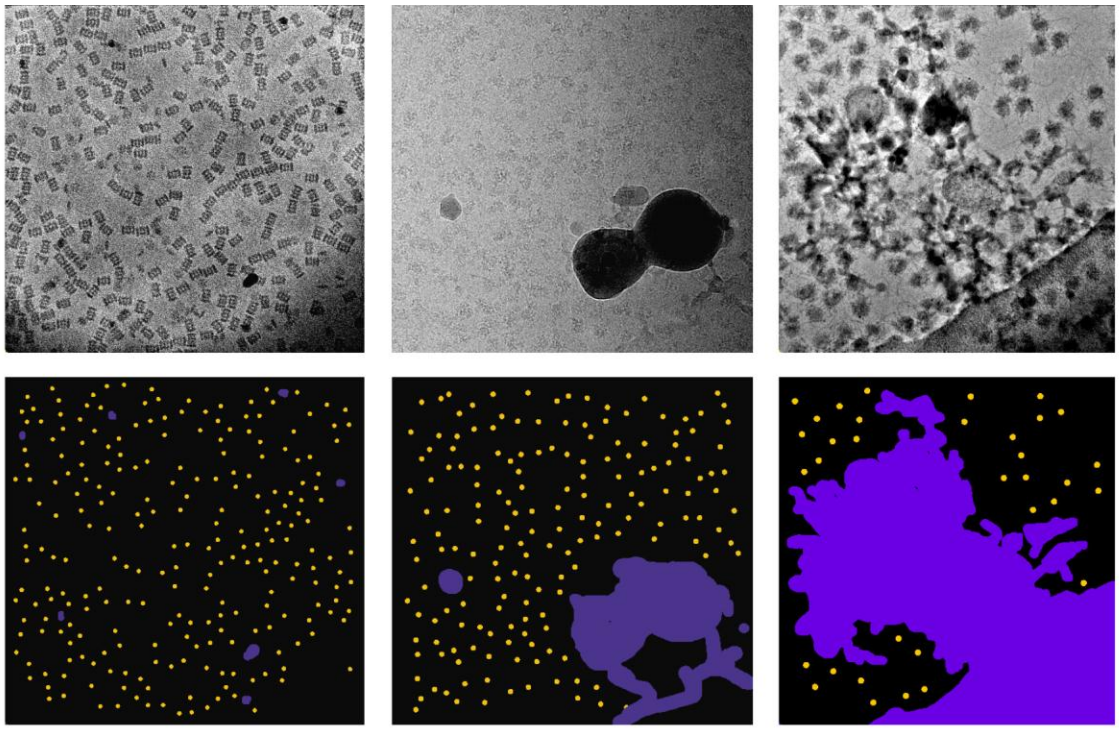
Examples of automated particle picking on samples not seen by BoxNet in training. For comparison, the same micrographs were picked with crYOLO's generic model, and RELION's Laplacian of Gaussian (LoG) method. Micrographs were selected from in-house data to make sure they were absent in crYOLO's knowledge base. BoxNet reliably recognizes almost all particles (yellow), and masks out all artifacts (purple). LoG is often confused by high-contrast edges and ethane impurities. crYOLO performs better than LoG, but is also routinely confused by ethane impurities and protein aggregates, and misses many of the small particles (bottom row).



Supplementary Figure 6

Neural network architecture of BoxNet

Rectangles depict the intermediate tensor dimensions. Their width and height are proportional to the number of channels and the spatial extent, respectively. Thick arrows represent convolution operations. Their format is encoded as "(Kx R), LxMxN /O", where K is the number of consecutive ResNet blocks, or absent in case of a single convolution operation; L and M are the dimensions of the convolution kernel; N is the number of kernels, resulting in N channels in the output; O is the stride length (1 = no change, 2 = downsampling by factor of 2, 0.5 = upsampling by factor of 2 through transposed convolution). The stride parameter is only applied to the first convolution in a chain of ResNet blocks, whereas all subsequent convolutions use stride = 1. The contractive part of the network is colored in cyan, the expanding part in magenta. The final image shows the result of applying a per-pixel ArgMax operator to the result of the last convolution to obtain the spatial distribution of the 3 labels the model is trained to predict: background (black), particle (yellow), artifact (purple).



Supplementary Figure 7

Examples of data used to train BoxNet

Examples of micrographs presented to BoxNet as input (top row), and the per-pixel labels used as the desired output during training (bottom row). The pixel classes predicted by BoxNet are background (black), particles (yellow), and artifacts (purple).

Supplementary Table 1 | Experimental and synthetic data used to train the general BoxNet model.

Access code	Sample	Synthetic	Carbon support	Phase plate
EMPIAR-10017	beta-galactosidase			
EMPIAR-10077	80S ribosome		✓	
EMPIAR-10078	20S proteasome			✓
EMPIAR-10081	HCN1 channel			
EMPIAR-10084	Haemoglobin			✓
EMPIAR-10089	TcdA1 in prepore state			
EMPIAR-10097	Influenza Hemagglutinin			
EMPIAR-10122	Apoferritin			✓
EMPIAR-10153	80S ribosome		✓	✓
EMPIAR-10156	80S ribosome		✓	
EMPIAR-10200	Apoferritin			
	RNA Polymerase II complex			
	RNA Polymerase II complex		✓	
	RNA Polymerase II complex			
	Viral polymerase			
	Nucleosome complex			
	Chromatin-related		✓	
	Chromatin-related			
	Transcription-related complex			
	Transcription-related complex			
	Transcription-related complex		✓	
	Chromatin-related			
	Chromatin-related		✓	
	RNA Polymerase II complex			
	Nucleosome			
PDB-1sa0	Tubulin-Colchicine	✓		
PDB-2gtl	Lumbricus Erythrocruorin	✓		
PDB-2wri	70S ribosome	✓		
PDB-3j9i	20S proteasome	✓		
PDB-4hhb	Haemoglobin	✓		
PDB-4zor	S37P MS2 viral capsid	✓		
PDB-5foj	Grapevine Fanleaf virus	✓		
PDB-5mmi	Chloroplast ribosome, large subunit	✓		
PDB-5ngm	70S ribosome	✓		
PDB-5vy5	Aldolase	✓		
PDB-5w3l	Rhinovirus B14	✓		
PDB-5w3s	TRPML3 channel	✓		
PDB-5xnl	Stacked PSII-LHCII supercomplex	✓		
PDB-5xwy	LbuCas13a-crRNA complex	✓		
PDB-5y6p	Phycobilisome	✓		
PDB-6az1	80S ribosome, small subunit	✓		
PDB-6b7n	Coronavirus spike protein	✓		
PDB-6b44	CRISPR Csy surveillance complex	✓		
PDB-6bco	TRPM4 channel	✓		
PDB-6bcx	mTORC1	✓		
PDB-6bhu	MRP1	✓		

Supplementary Note 1 | Pseudo-code description of the CTF and motion fitting algorithms.

CTF fitting algorithm, time-invariant case:

```
Vec2[] TilePositions = UniformSpacing(MicrographDimensions, WindowSize, Overlap)

// Extract tiles from frames, apply FFT, accumulate the 2D amplitudes
// for each tile.

Image[] TileSpectra
for each Position in TilePositions:
    for each Frame in Movie:
        Tile = SoftEdge(Normalize(Extract(Frame, Position, WindowSize)))
        TileFT = FFT(Tile)
        TileSpectra[Position] += Amplitudes(TileFT)

// Prepare 1D spectrum for initial grid search as sum of rotational
// averages of all tile spectra.

float[] Spectrum1D
for each Spectrum in TileSpectra:
    Spectrum1D += RotationalAverage(Spectrum)

// Perform exhaustive grid search within specified parameter ranges.

Vec2 BestParameters
float BestScore = -1
for each Defocus, PhaseShift in ParameterRanges:
    Simulated1D = SimulateCTF1D(Defocus, PhaseShift)
    Score = NormalizedCorrelation(Spectrum1D, Simulated1D)
    if Score > BestScore:
        BestScore = Score
        BestParameters = (Defocus, PhaseShift)

// Fit background and envelope, and subtract background from spectra

Background = FitSpline(Spectrum1D, CTFZeros(BestParameters))
Envelope = FitSpline(Spectrum1D - Background, CTFPeaks(BestParameters))
for each Spectrum in TileSpectra:
    Spectrum -= Background

// Precalculate weights that relate contribution of individual
// tiles' scores to spatial model parameters, i.e. how much
// a tile's defocus will change if a change is made to one
// of the model's parameters.

float[][] Weights
for each ModelPoint in SpatialDefocusModel:
    for each TilePosition in TilePositions:
        Weight = CubicSplineWeight(ModelPoint, TilePosition)
        Weights[ModelPoint][TilePosition] = Weight

// Define cost and gradient functions for optimization.
// GlobalModel describes all CTF parameters that aren't
// spatially resolved, e.g. astigmatism, phase shift.

float TileCost(TileDefocus, Position, GlobalModel):
    Simulated2D = SimulateCTF2D(TileDefocus, GlobalModel)
    Simulated2D *= Envelope

    return NormalizedCorrelation(TileSpectra[Position], Simulated2D)

float Cost(SpatialDefocusModel, GlobalModel):
    OverallCost = 0
    for each Position in TilePositions:
        TileDefocus = CubicSplineInterpolate(SpatialDefocusModel, Position)
        OverallCost += TileCost(TileDefocus, Position, GlobalModel)

    return OverallCost

float[] Gradient(SpatialDefocusModel, GlobalModel):
    float[] TileGradients
    for each Position in TilePositions:
        TileDefocus = CubicSplineInterpolate(SpatialDefocusModel, Position)
        Gradient = (TileCost(TileDefocus + Delta) -
                    TileCost(TileDefocus - Delta)) / (Delta * 2)
```

```
        TileGradients[Position] = Gradient

    float[] Gradients
    for each ModelPoint in SpatialDefocusModel:
        for each TilePosition in TilePositions:
            Gradients[ModelPoint] += TileGradients[TilePosition] *
                                     Weights[ModelPoint][TilePosition]

    for each Parameter in GlobalModel:
        GlobalModelAltered = GlobalModel
        GlobalModelAltered[Parameter] += Delta
        CostP = Cost(SpatialDefocusModel, GlobalModelAltered)

        GlobalModelAltered[Parameter] -= Delta * 2
        CostM = Cost(SpatialDefocusModel, GlobalModelAltered)

        Gradients[Parameter] = (CostP - CostM) / (Delta * 2)

    return Gradients

// Initialize SpatialDefocusModel with previously fitted global defocus,
// maximize the cost function using BFGS.

SpatialDefocusModel[:] = BestParameters[Defocus]
(SpatialDefocusModel, GlobalModel) = BFGSMaximize(Cost, Gradient, SpatialDefocusModel, GlobalModel)

// Finally, create a 1D rotational average from all tiles that considers
// the local defocus and the global astigmatism and magnification anisotropy.

Spectrum1D[:] = 0
for each Position in TilePositions:
    TileDefocus = CubicSplineInterpolate(SpatialDefocusModel, Position)
    TileSpectrum1D = AnisotropicRotationalAverage(TileSpectra[Position], TileDefocus, GlobalModel)
    Spectrum1D += ScaleDefocus(TileSpectrum1D, TileDefocus, BestParameters[Defocus])
```

Motion fitting algorithm:

```
Vec2[] TilePositions = UniformSpacing(MicrographDimensions, WindowSize, Overlap)

// Extract tiles from frames, apply FFT, apply band-pass and B-factor.

Image[][] TileFTs
for each Position in TilePositions:
    for each Frame in Movie:
        Tile = SoftEdge(Normalize(Extract(Frame, Position, WindowSize)))
        TileFT = FFT(Tile)
        TileFTs[Frame][Position] = Bandpass(TileFT) * SimulateBFactor()

// Precalculate weights that relate contribution of individual
// tiles' scores to spatio-temporal model parameters, i.e. how much
// a tile's shift will change if a change is made to one
// of the model's parameters.

float[][][] WeightsStage
float[][][] WeightsBIM
for each Frame in Movie
    for each TilePosition in TilePositions:
        for each ModelPoint in ModelStage:
            Weight = CubicSplineWeight(ModelPoint, Frame, TilePosition)
            WeightsStage[ModelPoint][Frame][TilePosition] = Weight
        for each ModelPoint in ModelBIM:
            Weight = CubicSplineWeight(ModelPoint, Frame, TilePosition)
            WeightsBIM[ModelPoint][Frame][TilePosition] = Weight

// Define cost and gradient functions for optimization.

float TileFrameCost(Shift, Frame, Position, TileAverage):
    Shifted = ShiftFT(TileFTs[Frame][Position], Shift)

    return EuklideanDistance(Shifted, TileAverage)
```

```
Image MakeTileAverage(Position, ModelStage, ModelBIM)
    Image Average
    for each Frame in Movie:
        ShiftStage = CubicSplineInterpolate(ModelStage, Frame, Position)
        ShiftBIM = CubicSplineInterpolate(ModelBIM, Frame, Position)
        Average += ShiftFT(TileFTs[Frame][Position], ShiftStage + ShiftBIM)

    return Average / NFrames

float Cost(ModelStage, ModelBIM):
    OverallCost = 0
    for each Position in TilePositions:
        TileAverage = MakeTileAverage(Position, ModelStage, ModelBIM)
        for each Frame in Movie:
            ShiftStage = CubicSplineInterpolate(ModelStage, Frame, Position)
            ShiftBIM = CubicSplineInterpolate(ModelBIM, Frame, Position)
            OverallCost += TileFrameCost(ShiftStage + ShiftBIM, Frame, Position, TileAverage)

    return OverallCost

float[] Gradient(ModelStage, ModelBIM):
    Vec2[][] TileGradients
    for each Position in TilePositions:
        TileAverage = MakeTileAverage(Position, ModelStage, ModelBIM)
        for each Frame in Movie:
            ShiftStage = CubicSplineInterpolate(ModelStage, Frame, Position)
            ShiftBIM = CubicSplineInterpolate(ModelBIM, Frame, Position)
            Shift = ShiftStage + ShiftBIM
            GradientX = (TileFrameCost(Shift + DeltaX, Frame, Position, TileAverage) -
                         TileFrameCost(Shift + DeltaX, Frame, Position, TileAverage)) / (DeltaX * 2)
            GradientY = (TileFrameCost(Shift + DeltaY, Frame, Position, TileAverage) -
                         TileFrameCost(Shift + DeltaY, Frame, Position, TileAverage)) / (DeltaY * 2)
            TileGradients[Frame][Position] = (GradientX, GradientY)

    float[] Gradients
    for each ModelPoint in ModelStage:
        for each TilePosition in TilePositions:
            for each Frame in Movie:
                Gradients[ModelPoint] += TileGradients[Frame][TilePosition] *
                                         Weights[ModelPoint][Frame][TilePosition]

    for each ModelPoint in ModelBIM:
        for each TilePosition in TilePositions:
            for each Frame in Movie:
                Gradients[ModelPoint] += TileGradients[Frame][TilePosition] *
                                         Weights[ModelPoint][Frame][TilePosition]

    return Gradients

// Finally, minimize the cost function using BFGS.

(ModelStage, ModelBIM) = BFGSMinimize(Cost, Gradient, ModelStage, ModelBIM)
```