The phase unwrapping of under-sampled interferograms using compact radial basis function neural networks

P.-A. Gourdain[1,*] **and A. Bachmann**[1]

[1]Physics and Astronomy Department,University of Rochester,Rochester NY14627, USA
[*]gourdain@pas.rochester.edu

ABSTRACT

Interferometry can measure the shape or the material density of a system that could not be measured otherwise by recording the difference between the phase change of a signal and a reference phase. This difference is always between $-\pi$ and π while it is the absolute phase that is required to get a true measurement. There is a long history of methods designed to recover accurately this phase from the phase "wrapped" inside $]-\pi,\pi]$. However, noise and under-sampling limit the effectiveness of most techniques and require highly sophisticated algorithms that can process imperfect measurements. Ultimately, analyzing successfully an interferogram amounts to pattern recognition, a task where radial basis function neural networks truly excel at. The proposed neural network is designed to unwrap the phase from two-dimensional interferograms, where aliasing, stemming from under-resolved regions, and noise levels are significant. The important aspects of this work are the introduction of a three-stage learning, allowing the network to eliminate phase discontinuities sequentially, the absence of a large training set, as the network is learning directly from the data it is trying to unwrap, and finally a parallel Levenberg-Marquardt algorithm, which uses local network clustering and global synchronization capable of speeding up the unwrapping process substantially . Finally, compact radial basis functions were used to keep Jacobian matrices sparse.

Introduction

Usually, interferometry is used to measure the properties of a system (e.g. elevation, electron density) that would be impractical to measure otherwise[1]. However, the absolute phase ϕ_{GT}, or "ground truth", cannot be measured directly. Rather, only the wrapped value[2] ϕ_W, bounded between $-\pi$ and π, can be computed from the interferogram. As the wrapped phase ϕ_W cannot be used directly, the data needs to be unwrapped, a task deceptively challenging, especially in the presence of noise. To further complicate matters, large chunks of phase data might be missing due to the presence of strong signal cut-off (often seen in magnetic resonance imaging[3]) or under-sampling, as in dense plasma interferometry[4]. Further, data sets have grown extremely large, straining serial algorithms used in phase unwrapping (e.g functional MRI[5], interferometric synthetic aperture radar[6], shape reconstruction[7] or fringe projection profilometry[8]). Regardless of the problem, the phase unwrapping procedure needs to find an approximate phase ϕ that is such that $\phi = \phi_{GT} + o(\phi_{GT})$. The ground truth must be extracted from the intensity I given by

$$I(x,y) = A(x,y) + B(x,y)\cos^2\phi_{GT}(x,y). \tag{1}$$

Note that in the ideal case, where $A \equiv 0$ and $B \equiv 1$, we effectively measure the wrapped phase $\phi_W = W(\phi_{GT})$, where W is the wrapping operator defined as

$$W(\phi) = \phi - 2k\pi \text{ with } k \in \mathbb{Z} \text{ such that } W(\phi) \in]-\pi,\pi]. \tag{2}$$

In this work we demodulate the phase, i.e. turning the intensity data given by Eq. 1 into wrapped phase, using filtered Fourier transforms[9–11].

When the phase ϕ_{GT} is well-behaved (i.e. continuous, noise free, over-sampled) then phase unwrapping is straightforward[12]. However, when the phase is corrupted by noise or under-sampled (i.e. aliasing), unwrapping becomes difficult[13] and a variety of methods have been developed to overcome this issue. Early methods used branch-cuts[14–16], least-square algorithm[17,18] or polynomial phase approximation[19,20]. However, they also did not react well to noise, leading to the development of algorithms capable of handling high noise levels[21–26] using Kalman filters[27,28]. Machine learning algorithms were equally successful, using artificial neural networks[29], then deep learning[30–34] and finally convolutional networks[35,36]. Unlike previous techniques, which tend to use the grid given by the natural data layout, machine learning is usually not relying on the physical data structure

to preform the unwrapping. Compared to other phase-based measurements, high energy density plasma interferometry[37–39] has its set of unique challenges. Typically, external noise levels are relatively low since most interferometers use a laser beam[40] that is extremely bright (> 100 MW/cm^2). However, diffraction can generate artifacts that degrade beam quality. Further, these plasmas have energy densities on the order of 1 kJ/cm^3, and the continuum light they produce as a result can cause large-scale intensity blotches embedded inside the interferogram. High energy density plasmas can also be surrounded by complex structures, which blocks part of the beam and create regions free of interference fringes. The shape of these structures is often complex[41,42] and required to be removed from the input data. Finally, with electron density gradients relatively large, interferometry data is often under-sampled, creating zones where the interference pattern is not directly usable[43].

To deal with such practical considerations, we developed a parallel neural network algorithm capable of unwrapping phase data following a staged supervised learning. Parallelization is obtained by clustering neurons across contiguous regions, where overlapping neurons, called *ghost* neurons[44], are used to synchronized the different networks. Unlike previous methods using Levenberg-Marquardt algorithms[45] or spatial derivatives[46], the proposed radial basis neural networks[47] (RBFNN) to analyze interferograms with under-sampled regions caused by inadequate digitization or lossy data compression. Since radial basis functions are good interpolants for smooth functions[48], they guarantee that the RBFNN can interpolate[49] a phase that is smooth enough using one hidden layer. While we have not investigated multi-layer perceptrons[50,51], they are known to be excellent approximators[52], both types of networks are not that different in practice.[53,54] The mian reason we chose radial basis functions in this work hinges on keeping the Jacobian matrix, used in gradient-based training, as sparse as possible. While multilayer perceptrons use sigmoid-like functions, which are not compact, it is possible use radial basis functions that are compact[55], leading to RBFNN with sparse Jacobian matrices.

While most methods expect the phase data to be noisy, dealing with corrupted or missing phase data is not a trivial task. Removing corrupted phase from the learning process is especially challenging when the region to exclude is geometrically complex. Further, wrapped phase data can have different sizes. So, the method proposed here does not use a training data set containing wrapped and unwrapped data as it is the case in machine learning (e.g. Ref. 34). Rather, we train the RBFNN only on the wrapped phase data that the RBFNN is trying to unwrap. So, every new unwrapping is also a new training operation. However, based on the consideration just listed above, this approach gives the most flexibility when successive acquisitions yield phase data that is fundamentally different from the previous ones both in size and shape, as it is often the case in plasma electron density measurements.

Training of the staged neural network

Preliminary remarks

A condition to detect aliasing

Super-resolution imaging uses numerical or physical techniques that allow to effectively increase the resolution of an image[56,57]. This technique is required when the phase was wrapped more than once across two pixels in interferometry data. This phenomenon is known as aliasing[13,58]. It can present itself as a series of swift, consecutive jumps, a case relatively easy to detect when noise levels are low. It can also be completely inconspicuous. For instance, a phase which varies as $\phi_{GT}(n) = \xi - \pi$ and $\phi_{GT}(n+1) = \xi + \pi$, where $0 < \xi \ll 1$, would yield a seemingly constant wrapped phase $\phi_W(n) = \xi - \pi$ and $\phi_W(n+1) = \xi - \pi$. As a lower bound, we can see that aliasing is present when $|\partial\phi_{GT}| > 2\pi$, then $W(\partial\phi_{GT})$ has jumps inside $]-\pi,\pi]$. While this is only a necessary condition, it becomes sufficient when ϕ_{GT} is continuous, as we will see later, allowing to detect the presence of aliasing inside the data.

A key property of periodic functions

When any phase Φ is discretized, we can define its left derivative as $\partial_L\Phi(n) = \Phi(n) - \Phi(n-1)$. Using Eq. 2, we get $\mathscr{F}\left(\partial_L\phi_{GT}(n)\right) = \mathscr{F}\left(\phi_W(n) - \phi_W(n-1) + 2(k_n - k_{n-1})\pi\right)$ for any 2π-periodic function $\mathscr{F}$. As a result, $\mathscr{F}\left(\partial_L\phi_{GT}(n)\right) = \mathscr{F}\left(\partial_L\phi_W(n)\right)$. This is true for the whole domain if we use the linear extrapolation to define the left derivative at the left boundary as $\partial_L\Phi(1) = -\Phi(1) + 2\Phi(2) - \Phi(3)$. For the right derivative, defined a $\partial_R\Phi(n) = \Phi(n+1) - \Phi(n)$, we use the same reasoning to get $\mathscr{F}\left(\partial_R\phi_{GT}(n)\right) = \mathscr{F}\left(\partial_R\phi_W(n)\right)$. This is valid across the whole domain if we now used the linear extrapolation of the right derivative at the right boundary as $\partial_R\Phi(N) = -\Phi(N-2) + 2\Phi(N-1) - \Phi(N)$. This property does not extend to the central derivative $\partial_C\Phi(n) = \frac{1}{2}[\Phi(n+1) - \Phi(n-1)]$ since $\mathscr{F}\left(\partial_C\phi_{GT}(n)\right) = \mathscr{F}\left(\partial_C\phi_W(n) + (k_{n+1} - k_{n-1})\pi\right)$ is equal to $\mathscr{F}\left(\partial_C\phi_W(n)\right)$ only when $k_n - k_{n-1}$ is even, but not when it is odd. In the end, we find

$$\mathscr{F}(\partial_{L,R}\phi_W) = \mathscr{F}(\partial_{L,R}\phi_{GT}). \tag{3}$$

It happens that the second derivative $\partial^2\Phi = \Phi(n+1) - 2\Phi(n) + \Phi(n-1)$ is also invariant since $\mathscr{F}\left(\partial^2\phi_{GT}(n)\right) = \mathscr{F}\left(\phi_W(n+1) - 2\phi_W(n) + \phi_W(n-1) + 2(k_{n+1} - 2k_n + k_{n-1})\pi\right) = \mathscr{F}\left(\partial^2\phi_W(n)\right)$. If we linearly extrapolate the second derivatives to the domain boundaries as $\partial^2\phi(1) = \phi(1) - 3\phi(2) + 3\phi(3) - \phi(4)$ and $\partial^2\phi(N) = \phi(N-3) - 3\phi(N-2) + 3\phi(N-1) - \phi(N)$, then

this result is still valid for the whole domain and we get

$$\mathscr{F}(\partial^2\phi_W) = \mathscr{F}(\partial^2\phi_{GT}). \tag{4}$$

Applying the same reasoning while using Eqs. 3 and 4 we can easily show that

$$W(\partial_{L,R}\phi_W) = W(\partial_{L,R}\phi_{GT}), \tag{5}$$

also known as the Itoh condition[12], and

$$W(\partial^2\phi_W) = W(\partial^2\phi_{GT}). \tag{6}$$

A restriction on the ground-truth

An important restriction arises when unwrapping the phase using RBFNNs. Since the output layer is a sum of radial basis functions, which are smooth, the output is also smooth. So, the RBFNN can only unwrap a phase which ground truth is smooth. The continuity criterion is defined by the interferometer resolution here. If the ground truth, while continuous, varies too quickly for the interferometer to measure the change, there will be a discontinuity in the signal and the ground truth will "appear" discontinuous. However, when few discontinuities are present, they can be hidden relatively easily from the RBFNN using a mask. This condition is usually not restrictive for interferograms generated by high energy density plasmas. If we work with a phase ϕ_{GT} that is twice-continuous and not aliased, i.e. $\partial_{L,R}\phi_{GT} \in]-\pi,\pi]$, Eq. 5 gives

$$W(\partial_{L,R}\phi_W) = \partial_{L,R}\phi_{GT}.$$

As a result, $W(\partial_{L,R}\phi_W)$ is continuous since $\partial_{L,R}\phi_{GT}$ is continuous, regardless of how many phase jumps are present in $\partial_{L,R}\phi_W$[12].

Since our goal is to deal with aliased phase, we can use the much less restrictive assumption $\partial^2\phi_{GT} \in]-\pi,\pi]$, and Eq. 6 gives

$$W(\partial^2\phi_W) = \partial^2\phi_{GT}. \tag{7}$$

Further, if $\partial^2\phi_{GT}$ is continuous across the whole domain then $W(\partial^2\phi_W)$ is also continuous everywhere. We will make both assumptions in the rest of the paper.

Construction of the input layer

While ϕ_W has jumps, we have shown that $W(\partial_{L,R}\phi_W)$ and $W(\partial^2\phi_W)$ are continuous if ϕ_{GT} is twice continuous. Yet, we cannot match the RBFNN output ϕ to ϕ_{GT} using gradient-based optimization since the wrapping operator W, which turns ϕ_{GT} into ϕ_W, is not differentiable. We will use here a gradient-based methods even if gradient-free methods[59–64] have been used successfully in machine learning. Eqs. 3 and 4 shows that we can use the differentiable sine and cosine functions instead of W where differentiability is required. As long as ϕ_{GT} is twice continuous, these functions remove the spurious discontinuities otherwise present in $\partial_{L,R}\phi_W$ and $\partial^2\phi_W$ at every phase jump of ϕ_W.

Input layer to achieve super-resolution

We can now construct an input layer $i_{1,\dots,12}$, where all the data is continuous. At every location inside the interferogram, we get:

$$\left.\begin{array}{llll} & i_3 = \cos(\partial_{xL}\phi_W), & i_7 = \cos(\partial_{xR}\phi_W) & \\ i_1 = \cos(\phi_W), & i_4 = \sin(\partial_{xL}\phi_W), & i_8 = \sin(\partial_{xR}\phi_W), & i_{11} = W(\partial_{xx}\phi_W) \\ i_2 = \sin(\phi_W), & i_5 = \cos(\partial_{yL}\phi_W), & i_9 = \cos(\partial_{yR}\phi_W), & i_{12} = W(\partial_{yy}\phi_W) \\ & i_6 = \sin(\partial_{yL}\phi_W), & i_{10} = \sin(\partial_{yR}\phi_W) & \end{array}\right\}. \tag{8}$$

We can now compare the input layer with the RBFNN output ϕ using the following set of equations

$$\left.\begin{array}{llll} & o_3 = \cos(\partial_x\phi), & o_7 = \cos(\partial_x\phi) & \\ o_1 = \cos(\phi), & o_4 = \sin(\partial_x\phi), & o_8 = \sin(\partial_x\phi), & o_{11} = \partial_{xx}\phi \\ o_2 = \sin(\phi), & o_5 = \cos(\partial_y\phi) & o_9 = \cos(\partial_y\phi), & o_{12} = \partial_{yy}\phi \\ & o_6 = \sin(\partial_y\phi), & o_{10} = \sin(\partial_y\phi) & \end{array}\right\}. \tag{9}$$

Note that, while i_{11} and i_{12} are still using the wrapping operator, this operator is not present in the equations used to compare the input layer and the RBFNN output because we restricted the second derivative to be between $-\pi$ and π. As the wrapping W has no effect of the RBFNN output, it has completely disappeared from o_{11} and o_{12} and we now can take their derivatives. However, this operator is still required on the left hand side of i_{11} and i_{12} to remove the phase jumps in $\partial^2\phi_W$. Also note that we have dropped the subscript L and R for the first derivatives of the output of the RBFNN, since it is a sum of analytical functions, which derivatives can be computed exactly.

Input layer when super-resolution is not required

The training can be substantially simplified when super-resolution is not required, i.e. $\partial\phi_{GT} \in]-\pi,\pi]$. In this case, we can replace Eq. 8 with

$$\left.\begin{array}{lll} \underline{i}_1 = \cos(\phi_W), & \underline{i}_3 = W(\partial_{xL}\phi_W), & \underline{i}_5 = W(\partial_{xR}\phi_W) \\ \underline{i}_2 = \sin(\phi_W), & \underline{i}_4 = W(\partial_{yL}\phi_W), & \underline{i}_6 = W(\partial_{yR}\phi_W) \end{array}\right\} \tag{10}$$

and Eq. 9 with

$$\left.\begin{array}{lll} \underline{o}_1 = \cos(\phi), & \underline{o}_3 = \partial_x\phi, & \underline{o}_5 = \partial_x\phi \\ \underline{o}_2 = \sin(\phi), & \underline{o}_4 = \partial_y\phi, & \underline{o}_6 = \partial_y\phi \end{array}\right\}. \tag{11}$$

The activation function

Throughout this paper, the RBFNN will use a compact Wendland function[55] as the activation function. Such functions can be constructed easily starting from

$$\psi_{p,0}(r) = (1-r)_+^p = \begin{cases} (1-r)^p & \text{for } 0 \le r \le 1 \\ 0 & \text{for } r > 1 \end{cases}$$

and using

$$\psi_{p,q}(r) = \Im^q \psi_{p,0} \text{ for } 0 \le r \le 1$$

to increase the function smoothness. Here $p,\ q \in \mathbb{N}$. The operator $\Im$ above is defined as $\Im f(r) = \int_r^\infty f(t)tdt$ for $0 \le r$. Wendland functions are C^k and can be computed analytically. They yield a strictly positive definite matrix in $\mathbb{R}^d$, where d<p and k=2q. The subscripts of $\psi_{p,q}$ will be dropped in the rest of the paper. Each neuron (x_n, y_n) in our two dimensional dataset is activated using such radial basis functions[47,65]. In this paper, we use exclusively the Wendland function ψ given by

$$\psi(r) = \begin{cases} \frac{1}{3}(1-r)^6\left[(35r+18)r+3\right] & \text{for } 0 \le r \le 1 \\ 0 & \text{for } r > 1 \end{cases} \tag{12}$$

obtained for $p = 3$ and $q = 2$.

The output layer

The output layer ϕ is expressed as a sum of radial basis functions $\psi_n(x,y) = \psi(r_n)$ centered on each neuron n located at (x_n, y_n) and scaled by the weight w_n. The output layer of a RBFNN with N neurons is continuous and defined as

$$\phi(x,y) = \sum_{n=1}^{N} w_n(x,y)\psi(r_n(x,y)) \text{ with } r_n(x,y) = \sqrt{(x-x_n)^2\rho_{x_n}^2 + (y-y_n)^2\rho_{y_n}^2}, \tag{13}$$

where ρ_{x_n} and ρ_{y_n} are the activation distance inverses for the n^{th} neuron along the x- and y-directions respectively. The analytical expression of the Jacobian matrix is greatly simplified when using the inverse of the activation distance. As discussed later in the paper, we need to match five constraints to give the neural network super-resolution, i.e. ϕ_{GT}, $\partial_x\phi_{GT}$, $\partial_y\phi_{GT}$, $\partial_{xx}\phi_{GT}$, $\partial_{yy}\phi_{GT}$. To match five constraints, we need to inject three degrees of freedom inside the weights as

$$w_n(x,y) = a_n + b_n(x-x_n) + c_n(y-y_n). \tag{14}$$

Together with ρ_{x_n}, and ρ_{y_n}, we now have five degrees of freedom per neuron. Note that the weights w_n are now local linear[66,67] in x and y.

Definition of the objective function

Objective function with super-resolution

We can now define the objective function $F(\mathbf{e})$, used by the training process to minimize the error vector $\mathbf{e} = [e_1, \ldots, e_N]^T$ for all neurons $n \in \{1, \ldots, N\}$

$$F(\mathbf{e}) = \mathbf{e}^{\mathrm{T}}\mathbf{e} = \sum_{n=1}^{N}\sum_{j=1}^{12} e_{jn}^2 \text{ with } e_{jn} = o_{jn} - i_{jn}. \tag{15}$$

The error e_{jn} is the difference between the j^{th} input layer value computed at the location (x_n, y_n), i.e. i_{jn}, and the j^{th} output layer value computed at the same location, i.e. o_{jn}. We expect the total error to remain high, even after full convergence, since the training will never match the left and right derivatives simultaneously. Therefore, we need to define another error $\hat{\mathbf{e}}$ to estimate when our network is fully trained,

$$\hat{\mathbf{e}}^{\mathrm{T}}\hat{\mathbf{e}} = \sum_{n=1}^{N}\sum_{j=1}^{8}\hat{e}_{jn}^2, \tag{16}$$

where $\hat{e}_{jn} = \hat{o}_{jn} - \hat{i}_{jn}$, and

$$\left.\begin{array}{lllllllll} \hat{i}_1 = i_1, & \hat{i}_2 = i_2, & \hat{i}_3 = \frac{1}{2}(i_3+i_7), & \hat{i}_4 = \frac{1}{2}(i_4+i_8), & \hat{i}_5 = \frac{1}{2}(i_5+i_9), & \hat{i}_6 = \frac{1}{2}(i_6+i_{10}), & \hat{i}_7 = i_{11}, & \hat{i}_8 = i_{12} \\ \hat{o}_1 = o_1, & \hat{o}_2 = o_2, & \hat{o}_3 = o_3, & \hat{o}_4 = o_4, & \hat{o}_5 = o_5, & \hat{o}_6 = o_6, & \hat{o}_7 = o_{11}, & \hat{o}_8 = o_{12} \end{array}\right\}.$$

Objective function without super-resolution

The objective function for interferograms where super-resolution is not needed is defined as $\underline{F}(\underline{\mathbf{e}})$ and should be used to minimize the error vector $\underline{\mathbf{e}} = [\underline{e}_1, \ldots, \underline{e}_N]^T$ for all neurons $n \in \{1, \ldots, N\}$

$$\underline{F}(\underline{\mathbf{e}}) = \underline{\mathbf{e}}^{\mathrm{T}}\underline{\mathbf{e}} = \sum_{n=1}^{N}\sum_{j=1}^{6}\underline{e}_{jn}^2 \text{ with } \underline{e}_{jn} = \underline{o}_{jn} - \underline{i}_{jn}. \tag{17}$$

As we did earlier we can define the error $\hat{\underline{\mathbf{e}}}$ to better assess the actual convergence error

$$\hat{\underline{\mathbf{e}}}^{\mathrm{T}}\hat{\underline{\mathbf{e}}} = \sum_{n=1}^{N}\sum_{j=1}^{8}\hat{\underline{e}}_{jn}^2, \tag{18}$$

where $\hat{\underline{e}}_{jn} = \hat{\underline{o}}_{jn} - \hat{\underline{i}}_{jn}$, and

$$\left.\begin{array}{llll} \hat{\underline{i}}_1 = \underline{i}_1, & \hat{\underline{i}}_2 = \underline{i}_2, & \hat{\underline{i}}_3 = \frac{1}{2}(\underline{i}_3+\underline{i}_5), & \hat{\underline{i}}_4 = \frac{1}{2}(\underline{i}_4+\underline{i}_6) \\ \hat{\underline{o}}_1 = \underline{o}_1, & \hat{\underline{o}}_2 = \underline{o}_2, & \hat{\underline{o}}_3 = \underline{o}_3, & \hat{\underline{o}}_4 = \underline{o}_4 \end{array}\right\}.$$

Regularisation

Simple Bayesian regularisation[68], or more complex variants such as using Markov chain Monte Carlo[69], have been proposed to avoid over-fitting noisy data and it is necessary in the presence of noise.

$$F_R(\mathbf{e}) = \sum_{n=1}^{N}\sum_{j=1}^{12} e_{jn}^2 + \Omega\sum_{n=1}^{N}\left(a_n^2 + b_n^2 + c_n^2 + \rho_{x_n}^2 + \rho_{y_n}^2\right). \tag{19}$$

We found that Ω should be 1 during the first and second train stages since the noise has the largest impact on the second derivative of the wrapped phase. Regularization is typically not necessary during the last stage of the training and we can use $\Omega = 0$. When super-resolution is not needed we use

$$\underline{F}_R(\underline{\mathbf{e}}) = \sum_{n=1}^{N}\sum_{j=1}^{6}\underline{e}_{jn}^2 + \Omega\sum_{n=1}^{N}\left(a_n^2 + b_n^2 + c_n^2 + \rho_{x_n}^2 + \rho_{y_n}^2\right). \tag{20}$$

Multistage training

The staged Levenberg-Marquardt algorithm

~~The first step in the neural network training tries to match the network output to the second derivative of measured phase, using only the inputs $i_{11..12}$. Once the network is fully trained and the second derivatives of the output layer ϕ matches the second derivatives of ϕ_{GT} up to a small error, we restart the training process, but this time using the inputs $i_{3..12}$ to match the second derivative and the sine/cosine values of the first derivatives. We use the trigonometric functions to hide the discontinuities of the first derivatives of ϕ_W because trigonometric functions are differentiable and allow to compute the Jacobian matrix analytically. Once the network is trained (i.e. $e_{3..12}$ minimized for all neurons), the output layer should match the first central derivatives of ϕ_{GT} (as the training procedure matched both left and right derivatives, ultimately yielding the central derivative $\partial_C\phi_{GT}$) as well as the second derivatives. We finalize training the network using the all inputs $i_{1..12}$, including now the sine and cosine of ϕ_W, to hide the phase jumps this time rather than their discontinuities inside their first derivatives. Once the errors $e_{1..12}$ are minimized, the output layer now matches the second and first derivatives of ϕ_{GT} as well as ϕ_{GT} itself.~~

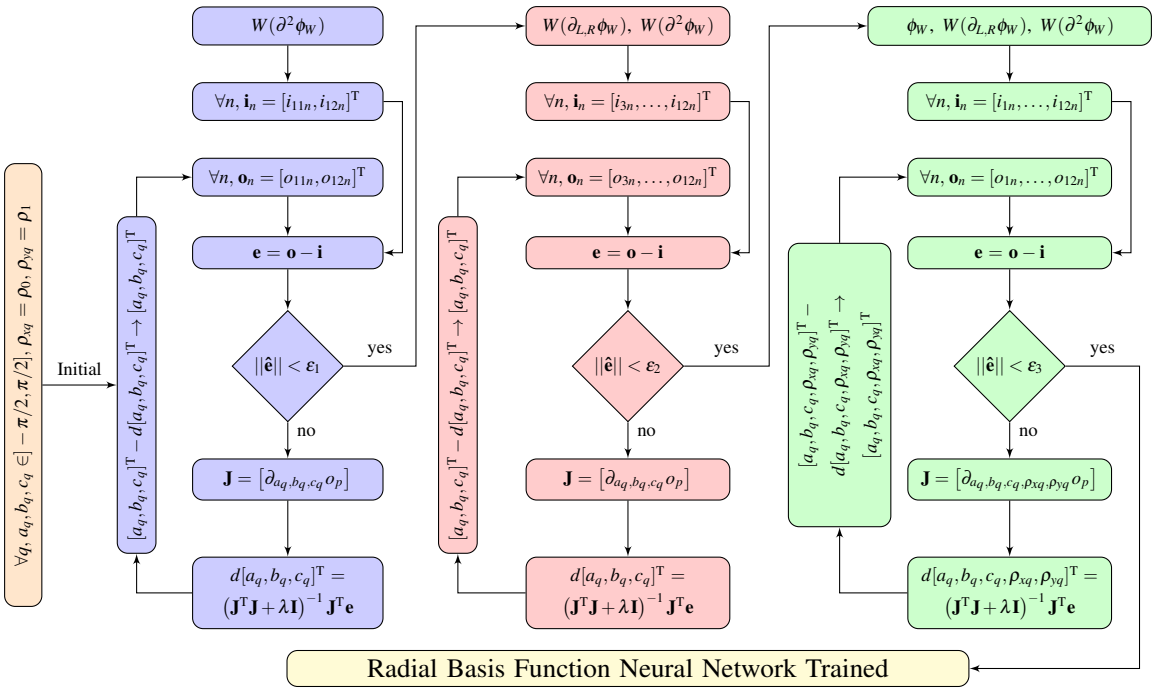

Figure 1. The staged training of the super-resolution RBFNN with the first stage in blue, the second stage in red and the last stage in green.

The minimization procedure needs to ~~highlighted above will~~ find the values of the basis function weights $u_n = (a_n, b_n, c_n, \rho_{x_n}, \rho_{y_n})$ for all neurons $n \in \{1, .., N\}$ using a gradient-based algorithm. We used here a standard Levenberg-Marquardt Algorithm[70,71] (LMA), which minimizes the error $\mathbf{e}$ (not $\hat{\mathbf{e}}$) in the sense of the least square using the Jacobian matrix $\mathbf{J}$. The solution is found by successive iterations, advancing the vector $\mathbf{u} = [u_1, \ldots, u_N]^{\mathrm{T}}$ such that $\mathbf{u}^{new} = \mathbf{u} - \mathrm{d}\mathbf{u}$ with

$$\mathrm{d}\mathbf{u} = \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda\mathbf{I}\right)^{-1}\mathbf{J}^{\mathrm{T}}\mathbf{e}.$$

The procedure to find λ follows a standard LM optimization method specific to RBFNN training[72–75]. This procedure can be altered, avoiding the storage of complete Jacobian matrices[76]. Regardless of the method used, the first two stages can be seen as an initialization of the weights of the RBFNN in the presence of wrapped input data. The last stage corresponds to the true optimization procedure. Since the different stages are following a conventional LM optimization using the appropriate inputs and Jacobian matrices, we only summarize here the goals of the three training stages:

1. *Matching the second derivatives of* ϕ_W: The error vector is defined as $\mathbf{e}_n = [e_{11n}, e_{12n}]^{\mathrm{T}} = [o_{11n} - i_{11n}, o_{12n} - i_{12n}]^{\mathrm{T}}$. We only train the neural network to optimize the radial basis function weights w_n at this stage using $\mathbf{u}_n = [u_{1n}, u_{2n}, u_{3n}]^{\mathrm{T}} = [a_n, b_n, c_n]^{\mathrm{T}}$. We have found that optimizing the activation distance early on does not really improve the quality of the output at this stage. The quality of convergence at this stage is crucial to super-resolution. This stage is shown in blue in Fig. 1.

2. *Matching the first and second derivatives of* ϕ_W: The error vector is now redefined as $\mathbf{e}_n = [e_{3n}, \ldots, e_{12n}]^{\mathrm{T}} = [o_{3n} - i_{3n}, \ldots, o_{12n} - i_{12n}]^{\mathrm{T}}$. Here again, we train the neural network to optimize the radial basis function weights w_n using $\mathbf{u}_n = [u_{1n}, u_{2n}, u_{n_3}]^{\mathrm{T}} = [a_n, b_n, c_n]^{\mathrm{T}}$. This stage propagates the super-resolution information to the first derivatives of the phase. This stage is shown in red in Fig. 1.

3. *Matching* ϕ_W *as well as the first and second derivatives of* ϕ_W: The error vector is defined as $\mathbf{e}_n = [e_{1n}, \ldots, e_{12n}]^{\mathrm{T}} = [o_{1n} - i_{1n}, \ldots, o_{12n} - i_{12n}]^{\mathrm{T}}$. We now optimize the neutral network to find the basis function weight w_n and the inverse activation distances at this stage so $\mathbf{u}_n = [u_{1n}, \ldots, u_{5n}]^{\mathrm{T}} = [a_n, b_n, c_n, \rho_{x_n}, \rho_{y_n}]^{\mathrm{T}}$. This stage unwraps the phase globally, in one single sweep. This stage is shown in green in Fig. 1.

When super resolution is not needed, the training will only try to match the first left and right derivatives, together with the wrapped phase using only two training stages:

1. *Matching the first derivatives of* ϕ_W: The error vector is first define as $\underline{\mathbf{e}}_n = [\underline{e}_{3n}, \ldots, \underline{e}_{6n}]^{\mathrm{T}} = [\underline{o}_{3n} - \underline{i}_{3n}, \ldots, \underline{o}_{6n} - \underline{i}_{6n}]^{\mathrm{T}}$. Here again, we train the neural network to optimize the radial basis function weights w_n using $\mathbf{u}_n = [u_{1n}, u_{2n}, u_{n_3}]^{\mathrm{T}} = [a_n, b_n, c_n]^{\mathrm{T}}$. This stage propagates the super-resolution information to the first derivatives of the phase. This stage is shown in blue in Fig. 2.

2. *Matching* ϕ_W *as well as the first derivatives of* ϕ_W: The error vector is defined as $\underline{\mathbf{e}}_n = [\underline{e}_{1n}, \ldots, \underline{e}_{6n}]^{\mathrm{T}} = [\underline{o}_{1n} - \underline{i}_{1n}, \ldots, \underline{o}_{6n} - \underline{i}_{6n}]^{\mathrm{T}}$. We now optimize the neutral network to find the actual basis function weight w_n and activation distances at this stage so $\mathbf{u}_n = [u_{1n}, \ldots, u_{5n}]^{\mathrm{T}} = [a_n, b_n, c_n, \rho_{x_n}, \rho_{y_n}]^{\mathrm{T}}$. This stage unwraps the phase globally, in one single sweep. This stage is shown in red in Fig. 2.

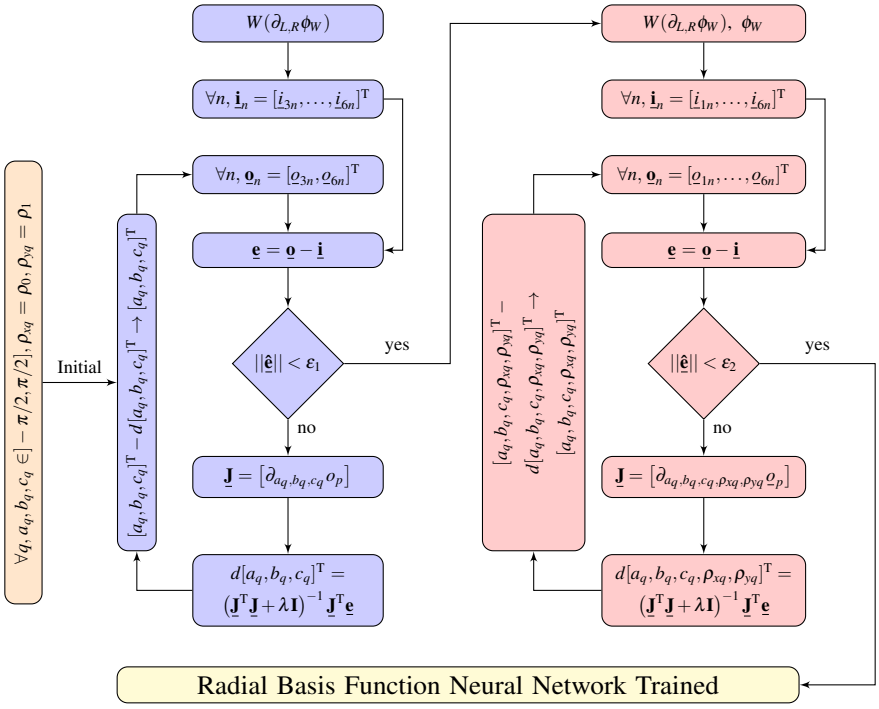

Figure 2. Staged training of the RBFNN without super-resolution. The first stage is in blue and the second stage in red.

Computation of the Jacobian matrix with super-resolution

The Jacobian matrix used in the last stage of the training is given by

$$\mathbf{J} = \begin{bmatrix} \partial_{u_1}\mathbf{e}_1 & \ldots & \partial_{u_N}\mathbf{e}_1 \\ \vdots & \ddots & \vdots \\ \partial_{u_1}\mathbf{e}_N & \ldots & \partial_{u_N}\mathbf{e}_N \end{bmatrix} \text{ where } \partial_{u_q}\mathbf{e}_p = \begin{bmatrix} \partial_{a_q}o_{1p} & \partial_{b_q}o_{1p} & \partial_{c_q}o_{1p} & \partial_{\rho_{x_q}}o_{1p} & \partial_{\rho_{y_q}}o_{1p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial_{a_q}o_{12p} & \partial_{b_q}o_{12p} & \partial_{c_q}o_{12p} & \partial_{\rho_{x_q}}o_{12p} & \partial_{\rho_{y_q}}o_{12p} \end{bmatrix}. \tag{21}$$

Here the matrix $\partial_{u_q}e_p$ corresponds to the partial derivative of the error e_p between the output layer and the input layer computed at the p^{th} neuron with respect to the weights u_q of the q^{th} neuron. Since the input layer does not depend on any neuron weights, the input values i_{1p} to i_{12p} have been dropped inside the partial derivatives and only the output values o_{1p} to o_{12p} were retained. To form the smaller Jacobian matrix matrices necessary to the first two stages, we just need to drop the corresponding terms in the full matrix $\partial_{u_q}e_p$, leading to

$$\text{First stage: } \partial_{u_q}\mathbf{e}_p = \begin{bmatrix} \partial_{a_q}o_{11p} & \partial_{b_q}o_{11p} & \partial_{c_q}o_{11p} \\ \partial_{a_q}o_{12p} & \partial_{b_q}o_{12p} & \partial_{c_q}o_{12p} \end{bmatrix} \text{ and Second stage: } \partial_{u_q}\mathbf{e}_p = \begin{bmatrix} \partial_{a_q}o_{3p} & \partial_{b_q}o_{3p} & \partial_{c_q}o_{3p} \\ \vdots & \vdots & \vdots \\ \partial_{a_q}o_{12p} & \partial_{b_q}o_{12p} & \partial_{c_q}o_{12p} \end{bmatrix}.$$

All the functions used in o_{1_n} to o_{12_n} are analytical and can be differentiated, since the wrapping operator W was dropped from o_{11_n} and o_{12_n} using the condition $\partial^2\phi \in]-\pi, \pi]$. We can now compute the Jacobian matrix elements taking the partial

derivative on every term in Eq. 9 with respect to $\omega \in \{a_q, b_q, c_q, \rho_{x_q}, \rho_{y_q}\}$

$$\left.\begin{array}{llllllll} & & \partial_\omega o_3 = & -\partial_{x\omega}\phi \sin(\partial_x\phi), & \partial_\omega o_7 = & -\partial_{x\omega}\phi \sin(\partial_x\phi) & & \\ \partial_\omega o_1 = & -\partial_\omega\phi \sin(\phi), & \partial_\omega o_4 = & \partial_{x\omega}\phi \cos(\partial_x\phi), & \partial_\omega o_8 = & \partial_{x\omega}\phi \cos(\partial_x\phi), & \partial_\omega o_{11} = & \partial_{xx\omega}\phi \\ \partial_\omega o_2 = & \partial_\omega\phi \cos(\phi), & \partial_\omega o_5 = & -\partial_{y\omega}\phi \sin(\partial_y\phi), & \partial_\omega o_9 = & -\partial_{y\omega}\phi \sin(\partial_y\phi), & \partial_\omega o_{12} = & \partial_{yy\omega}\phi \\ & & \partial_\omega o_6 = & \partial_{y\omega}\phi \cos(\partial_y\phi), & \partial_\omega o_{10} = & \partial_{y\omega}\phi \cos(\partial_y\phi) & & \end{array}\right\}. \tag{22}$$

The values of the partial derivatives used in Eq. 22 are listed in the Methods section.

Computation of the Jacobian matrix without super-resolution

When dropping super-resolution, the Jacobian matrix $\underline{\mathbf{J}}$ of Fig. 2

$$\underline{\mathbf{J}} = \begin{bmatrix} \partial_{u_1}\underline{\mathbf{e}}_1 & \dots & \partial_{u_N}\underline{\mathbf{e}}_1 \\ \vdots & \ddots & \vdots \\ \partial_{u_1}\underline{\mathbf{e}}_N & \dots & \partial_{u_N}\underline{\mathbf{e}}_N \end{bmatrix} \tag{23}$$

can be computed in a similar manner. Here the error $\partial_{u_q}\underline{\mathbf{e}}_p$ is given by

$$\partial_{u_q}\underline{\mathbf{e}}_p = \begin{bmatrix} \partial_{a_q}\underline{o}_{3p} & \partial_{b_q}\underline{o}_{3p} & \partial_{c_q}\underline{o}_{3p} \\ \vdots & \vdots & \vdots \\ \partial_{a_q}\underline{o}_{6p} & \partial_{b_q}\underline{o}_{6p} & \partial_{c_q}\underline{o}_{6p} \end{bmatrix},$$

for the first stage, and

$$\partial_{u_q}\underline{\mathbf{e}}_p = \begin{bmatrix} \partial_{a_q}\underline{o}_{1p} & \partial_{b_q}\underline{o}_{1p} & \partial_{c_q}\underline{o}_{1p} & \partial_{\rho_{x_q}}\underline{o}_{1p} & \partial_{\rho_{y_q}}\underline{o}_{1p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \partial_{a_q}\underline{o}_{6p} & \partial_{b_q}\underline{o}_{6p} & \partial_{c_q}\underline{o}_{6p} & \partial_{\rho_{x_q}}\underline{o}_{6p} & \partial_{\rho_{y_q}}\underline{o}_{6p} \end{bmatrix}$$

for the second stage.

Masking and clustering strategies

We now focus on the initialization of our network, looking at masking, neuron clustering and receptor connections. The mask should be chosen before the training starts and should remain the same throughout the training. Most interferometry data carries noise, discontinuities, and regions that should be dropped from the interferogram. The mask should keep ~~inside the input layer~~ only the data that can be unwrapped with minimal error propagation. The mask over discarded data should slightly overlap with useful data. This strategy allows to compute properly phase derivatives at the mask boundary rather than using extrapolations. Further, the mask should neither split the data into separate regions nor have constricted regions.

The optimal number of receptors is integrated in the optimization procedure and does not have to be computed beforehand. Since we are using compact radial basis functions, any input p such that $r_{pq} = [(x_p - x_q)^2\rho_{x_q}^2 + (y_p - y_q)^2\rho_{y_q}^2]^{1/2} > 1$ will not be connected to the neuron q. The training process is initialized by choosing arbitrary values for ρ_{x_q} and ρ_{y_q} and these values should be chosen carefully. In regions with rapid phase changes the activation distance inverses should be large.

Some neural networks can use a clustering method, such as k-means[77,78], to improve the quality and speed of the training. However, in our case, the data pattern is rather inextricable *a priori* without super-resolution, which is only gained *a posteriori*. As a result, the shape of the mask and the distance between neurons, rather than the data inside the input layer, truly shapes neurons clustering in this work. This greatly simplifies the clustering procedure, which now boils down to a straightforward graph partitioning[79] based on nearest-neighbor connections.

Parallel training

Parallelization becomes necessary for moderately large dataset[80,81], as the size of the Jacobian matrix $\mathbf{J}$, even sparse, could be difficult to handle on today's supercomputers. This is especially true for high resolution two-dimensional interferograms obtained when measuring the electron density of high energy density plasmas. The basic clustering strategy described above can be used to split the main network into K non-overlapping networks. As it is often the case with parallel codes, we introduce *ghost* neurons[44], which are duplicated neurons shared by exactly two networks. Since the training of each network is now done independently, a synchronization step is require to make sure that all the output layer match seamlessly.

We used a single-nearest-neighbor search to define a single-layer of ghost neurons at the boundary between each clusters, allowing for some overlap between networks so that output layers can match seamlessly after synchronization. However, the synchronization procedure needs to keep very few of these ghost neurons to "stitch" the domains together.

The output layer

The synchronization uses a constant phase Φ_k, which is added to the output layer of the network $k \in \{1,\ldots,K\}$ as

$$\phi_k(x,y) = \Phi_k + \sum_{q=M_k}^{N_k} w_q \psi(r_q) + \sum_{q=M_{gk}}^{N_{gk}} w_q \psi(r_q), \tag{24}$$

where the neurons $\{M_k,\ldots,N_k\}$ are the neurons only owned by the k^{th} network, while the neurons $\{M_{gk},\ldots,N_{gk}\}$ are the ghost neurons of the k^{th} network, owned by the neighbors of the k^{th} network. The last two terms of Eq. 24 are the non-synchronized outputs of the RBFNNs obtained using Eq. 13, and their weights are computed using the Levenberg-Marquardt procedure highlighted above. However, under such conditions, the Jacobian matrix is a block matrix given by

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & & \\ & \ddots & \\ & & \mathbf{J}_K \end{bmatrix}.$$

All the missing elements are 0. Here $\mathbf{J}_k$ is the Jacobian matrix of Eq. 21, but restricted to the k^{th} network where the error $\mathbf{e}_k$ is defined by Eq. 15, also restricted to the k^{th} network. As a result, $\mathbf{J}^{\mathrm{T}}\mathbf{J}$ is block diagonal and the Levenberg-Marquardt algorithm can be solved in parallel using

$$\mathrm{d}\mathbf{u}_k = \left(\mathbf{J}_k^{\mathrm{T}}\mathbf{J}_k + \lambda_k \mathbf{I}_k\right)^{-1} \mathbf{J}^{\mathrm{T}}{}_k \mathbf{e}_k \text{ for } k \in \{1,\ldots,K\}.$$

If super-resolution is not needed, we will use the Jacobian matrix for Eq. 23 instead of Eq. 21. Once the training is over, we need to synchronize the output layers across all networks. As the synchronization focuses solely on Φ_k, the network parameters a_k, b_k, c_k, ρ_{x_k}, and ρ_{y_k} are constant, so the last two terms of Eq. 24 are now two constants and do not need to be computed again throughout the synchronization procedure.

The input layer

For any ghost neuron q shared with the network k but owned by the network labelled l_{qk}, the value $\phi_k(x_q,y_q)$ might be initially different from the value $\phi_{l_{qk}}(x_q,y_q)$ when the staged training is over. Yet, we can synchronize the output layers across the different networks by simply defining the synchronization input layer of the k^{th} network as

$$I_{qk} = \phi_{l_{qk}}(x_q,y_q), \tag{25}$$

with corresponding output value

$$O_{qk} = \phi_k(x_q,y_q). \tag{26}$$

There is no need to use wrapping functions like sine or cosine here. We are dealing with a phase that has been unwrapped successfully for each separate networks at this point. But it remains out of synchronization across the domain. Now, the error to minimize is given by

$$\mathbf{E}^{\mathrm{T}}\mathbf{E} = \sum_{k=1}^{K} \sum_{q=M_{gk}}^{N_{gk}} E_{qk}^2 \text{ with } E_{qk} = O_{qk} - I_{qk}. \tag{27}$$

The error E_{qk} is the squared difference of the input layer value from Eq. 25 computed at the location (x_q,y_q) inside the k^{th} cluster, i.e. I_{qk}, and the output layer value computed at the same location, i.e. O_{qk}.

The synchronization Levenberg-Marquardt algorithm

We use again the Levenberg-Marquardt algorithm to minimize the error $\mathbf{E}$ in the sense of the least square using the synchronization Jacobian matrix

$$\mathbf{J_s} = \begin{bmatrix} \partial_{\Phi_1} E_{M_{g1}} & \ldots & \partial_{\Phi_K} E_{M_{g1}} \\ \vdots & \ddots & \vdots \\ \partial_{\Phi_1} E_{N_{gK}} & \ldots & \partial_{\Phi_K} E_{N_{gK}} \end{bmatrix}. \tag{28}$$

Here we cannot drop the input values I_{qk} from the Jacobian matrix since the input layer for the k^{th} network may depend on a phase bias $\Phi_{k'}$ when ghost neurons in the network k are owned by the network k'. We are now using a standard Levenberg-Marquardt algorithm to solve this problem. One final parallel three-stage training can be done after the synchronization procedure to eliminate any residual errors, while keeping all Φ_k constant.

Accuracy of the staged neural network training using experimental interferometry data

~~After a series of test used to determine the accuracy of the RBFNN and presented in the *Methods* section,~~ We now use the proposed staged training on real interferograms generated by the interference of a green laser beam with a high energy density plasma[82]. The phase shift corresponds to the line-average electron density[83] of the plasma. The plasmas were generated by using a multipin radial foil configuration[84] connected to the electrodes of a pulsed-power driver[85]. In this case, we do not know the ground-truth. So, we assessed the quality of the unwrapping procedure by looking at the difference between the measured wrapped phase and the neural network output layer. The final error, $\varepsilon(x,y) = |W(\phi_W(x,y)) - W(\phi(x,y)|)/(2\pi)$, is given in units of 2π. It is the normalized error with respect to the wrapped phase ϕ_W, which spans an interval of 2π.

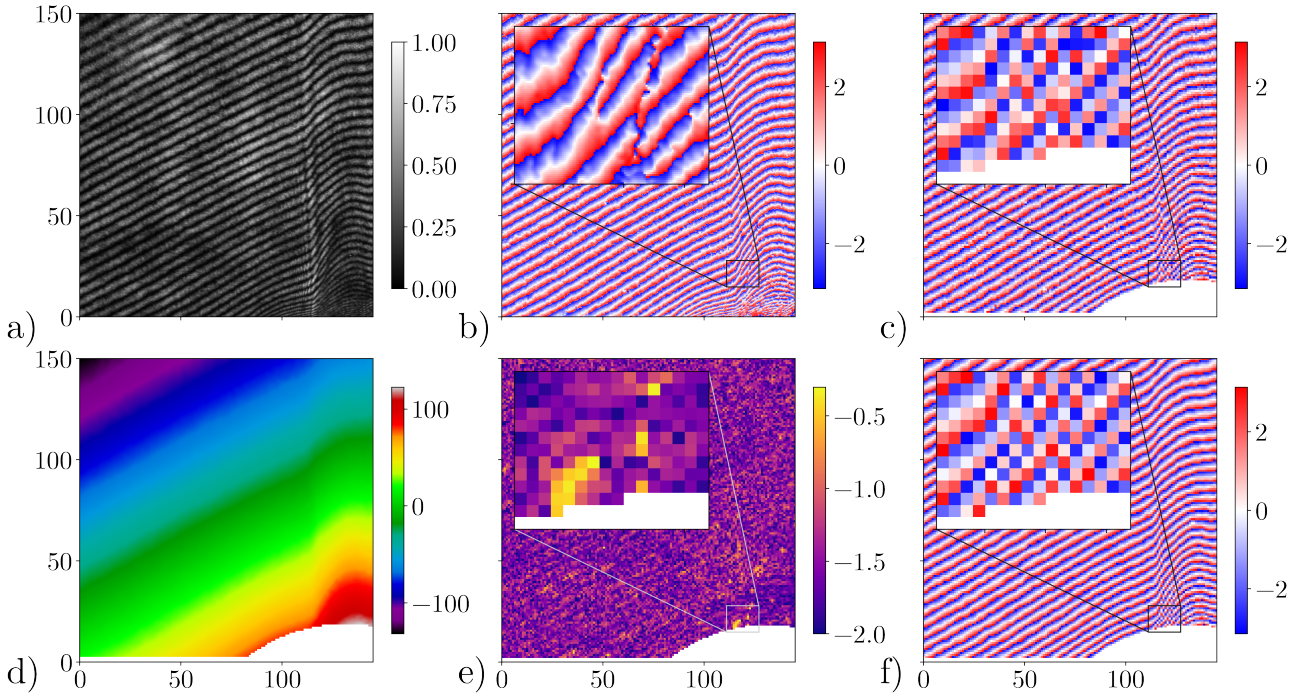

Figure 3. a) Normalized interferometer of the left side of a hollow plasma jet. The jet is symmetric with respect to the right axis. b) The wrapped phase after applying Fourier filtering to keep the dominant modes. c) Digitized down-sampled phase with mask to hide the regions where phase data should not be used. d) The actual output ϕ of the RBFNN and e) the $\log_{10}$ error between the wrapped phase ϕ_W and $W(\phi)$. f) The wrapped RBFNN output is given for reference. All phase data are in radians. The zoomed panels highlight the region with strongest aliasing.

The interferogram is presented in Fig. 3-a. The measurement is based on shearing[4] rather than Mach-Zehnder interferometry. The former uses a single reference path which is insensitive to mechanical vibrations, which greatly affects the fringe pattern of the latter. As a result, it is possible to use a reference phase, by using phase data without plasma, and subtract it from the measurement done when a plasma is present. The difference in phase is proportional to the line-average electron density. Starting with the region of interest shown in Fig. 3-a, the Fourier transform gives a spectrum that is symmetric with respect to the origin since the phase data is real valued. We use a single square filter to isolate the dominant modes, but excluding the origin, where the DC component is located. The inverse Fourier transform is now complex valued since the filter broke the symmetry with respect to the origin. The phase of each complex values corresponds to the wrapped phase measured by the interferogram[9–11] and seen in Fig. 3-b.

The data is then down-sampled by a factor of 6×6 to compress the interferogram (seen in Fig. 3-c). Using this data, we get a total of 150×140 neurons in the output layer. The input layer has a total of $12 \times 150 \times 140$ inputs when super-resolution is needed (see Eq. 8) and $6 \times 150 \times 140$ inputs where it is not (see Eq. 10). The only sample in the training set is the data itself, given in Fig. 3-c. While the compression is not necessary to demonstrate the efficacy of the RBFNN, this compression created a region with strong aliasing (the zoomed portion of Fig. 3-c). In reality, a strongly under-resolved interferogram may have a fringe count so high that some regions will have very low contrast. The Fourier filtering used to turn the experimental data, which form follows Eq. 1, into phase data cannot handle this high-density low-contrast situation. Note that, if the second derivative of the phase falls outside of $]-\pi,\pi]$, the RBFNN will also fail regardless of the quality of the filtering. The mask

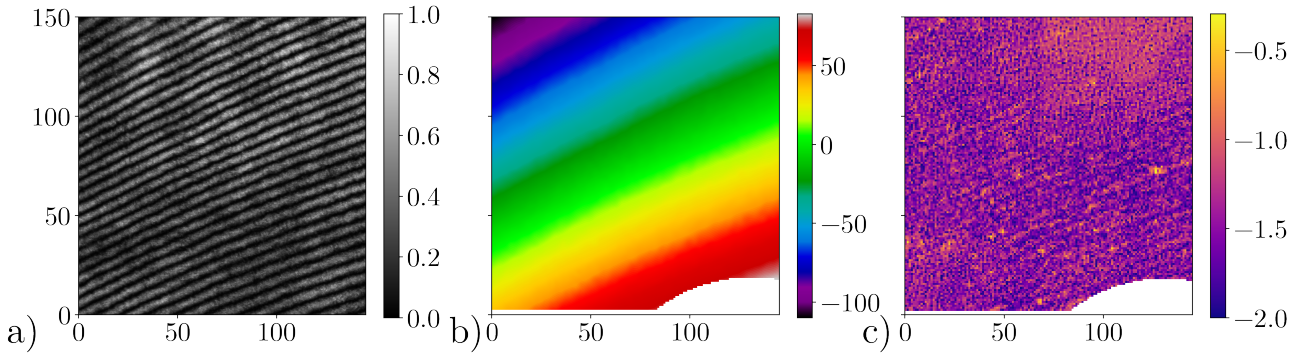

Figure 4. a) Normalized interferometer with no plasma present, b) the RBFNN output in radians ϕ and c) the $\log_{10}$ error between the wrapped phase ϕ_W and $W(\phi)$.

drops the data where fringes could not be resolved clearly, some edge data caused by the Fourier filtering (see Fig. 3-b near the x-axis), as well as under-resolved fringes, partly caused by the Fourier filtering. We hid the data using a disk-shape mask. We then trained the neural network with super-resolution. The output of the network is presented in Fig. 3-d. The bump in electron density caused the plasma jet appears clearly in the figure. The error between the measured wrapped phase of Fig. 3-c and the wrapped value of the output of the RBFNN of Fig. 3-d is on Fig.3-e is on the order of 10%, leading to an average error that is comparable to the noise recorded by the interferometer and clearly visible in the insert of Fig. 3-b. We see two types of error larger than 10 % in this figure. The error that is randomly distributed throughout is caused by a local phase jump caused by the noise when the wrapped phase is close to to $-\pi$ or π. This noise is not present in the output of the network due to regularization. The second type of error is closer to a true error, as the RBFNN has some difficulty to unwrap the phase accurately (region shown in the zoomed insert of Fig. 3-e). This error is coming from the 6×6 compression ratio, which has aliased the phase slightly beyond the capabilities of the RBFNN. However, this error can disappear if we use a 5×5 compression ratio. It is important to note that this error did not propagate to the neighboring neurons. If we consider the low, average error level of Fig. 3-e and the smoothness of the output, the RBFNN unwrapped the phase successfully. The wrapped output is shown in Fig. 3-f and can be compared to the measured phase in Fig. 3-c. Without super-resolution, the RBFNN was not able to unwrap the phase.

Since the shearing interferometer is mechanically stable, we can measure accurately the density of the jet by subtracting the background phase from Fig. 3-d. Following the exact same procedure, we can process the same region of the interferogram without any plasma. In this case, the fringe pattern is relatively periodic, as shown in Fig. 4-a. We get the wrapped background phase using the same Fourier filter as the one used for the interferogram with plasma. Since the pattern of the interferogram is clearly resolved, we trained the RBFNN without super-resolution, leading to the output presented in Fig. 4-b. It is interesting to note that the error between the wrapped output layer and the data, shown in Fig. 4-c, is similar to the error when the plasma is present. This indicates that the error is mostly caused by noise. Once the background phase is removed from the phase with plasma, we get the line average density of the jet presented in Fig. 5-a. While noise is present in the density measurement, its source has been filtered by our earlier Fourier transform. We believe that the density fluctuations seen in the RBFNN output derivatives shown in Fig. 5-b and c do not carry any physical information of the density itself. As a result, an Abel inversion technique that is robust to significant noise levels (e.g. Ref. 86) should be used to compute the volume electron density. We can note the difference in smoothness between domain due to the optimization of the activation distance during the last stage.

Discussion

The proposed RBFNN incorporates the functions necessary to deal with aliased interferograms by combining: 1-scattered neuron placement, allowing to discard relatively easily corrupted data while keeping data carrying high fidelity information; 2- the use of a mask to hide external geometries, which are often present in phase measurement; 3-a regularization scheme which can filter noise very effectively. The RBFNN can unwrapped the phase extracted from an interferogram by comparing measurements to the output of the RBFNN through sine and cosine functions. These functions hide the existence of any discontinuity in the wrapped phase from the training set. Taking into account that the RBFNN output is continuous by construction, the neural network yields a phase that is fully unwrapped once the error between the input and output layers has been minimized. As the network is trained to match the first and second derivatives of the phase, high-fidelity gradients can be computed directly from the RBFNN output since the impact of noise was limited by regularization. The network structure allow a clustering strategy where parallelization is easy to implement. It transforms a dense matrix into a block diagonal matrix,

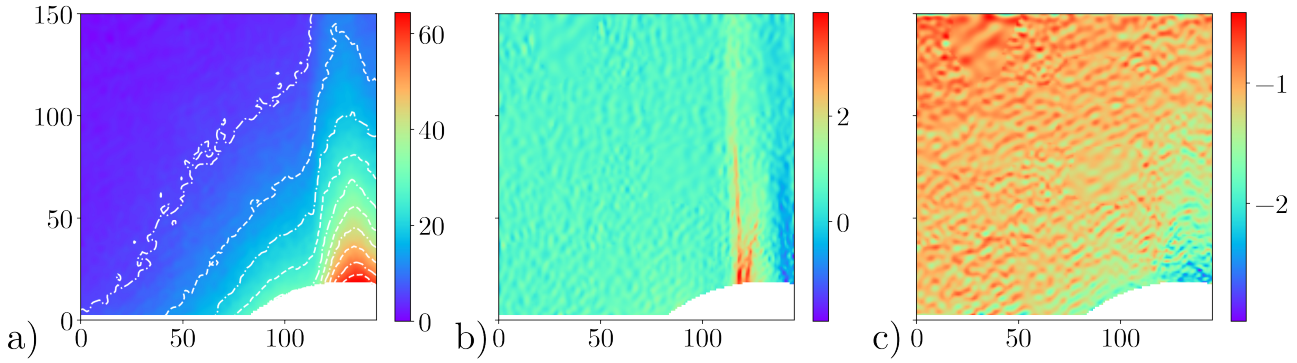

Figure 5. a) The line average density of the hollow plasma jet (given in radian) and the derivatives along b) the horizontal and c) vertical directions in arbitrary units. The axis of symmetry in to the right of each panel

speeding up the training substantially.

This work did not attempt to do any filtering in the pre-learning stage, except from a Fourier filter,which was mostly used to get the complex amplitude field, allowing to compute the wrapped phase readily. However, filtering techniques can be used in conjunction with the proposed algorithm. While regularization does filter data by limiting over-fitting, it should not be considered a very effective filter. First, the regularization parameter is global. Second, the regularization is static and there is no mechanism in place in our training that can optimize it.

While the RBFNN presented here requires more memory and computational power than more basic phase unwrapping algorithm (e.g. Ref. 17), errors are relatively easy to detect and remain local, as shown in Fig. 3-e. Combined with the proposed parallelization strategy, the unwrapping time can be reduced substantially. While the training procedures with and without super-resolution are clearly separated in this work, it is possible to use phase derivative averages to find region where super-resolution is required (i.e. $|W(\phi_w)| > \pi$) and regions where its not (i.e. $|W(\phi_w)| < \pi$). The Jacobian matrix can be adapted locally to each method seamlessly. However, this criterion are not absolute and super-resolution should be used as much as possible. It is also possible to extend the method to three-dimensional interferograms easily, as discussed in the *Method* section. At this point, a heavy use of block diagonalization is required to generate Jacobian matrices sparse enough to allow for reasonable training times.

Data availability

The datasets are available from the corresponding author upon request. Code samples can be found at https://github.com/Pierre-Alexandre-Gourdain/Phase-unwrapping.git

Methods

Computation of the partial derivatives used inside the Jacobian matrices

This section lists the analytic functions used to compute the Jacobian matrices used in this paper. The output layer is $\phi(x,y) = \sum_{q=1}^{N} w_q \psi(r_q)$ with $r_q = \sqrt{(x-x_q)^2 \rho_{x_q}^2 + (y-y_q)^2 \rho_{y_q}^2 + \varepsilon}$. Further, $\psi' = \frac{d}{dr}\psi$, $\psi'' = \frac{d^2}{dr^2}\psi$ and $\psi''' = \frac{d^3}{dr^3}\psi$. ε is used in computations to avoid a possible division by zero, which only happens numerically. The problematic terms w_q/r_q found below are multiplied by ψ' or ψ''', while $\lim_{r_q \to 0} \psi' \propto r_q$ and $\lim_{r_q \to 0} \psi''' \propto r_q$ for radial basis functions.

Partial derivatives with respect to the RBFNN parameters

$$\partial_{a_q} \phi = \psi(r_q)$$

$$\partial_{b_q} \phi = (x - x_q)\, \psi(r_q)$$

$$\partial_{c_q} \phi = (y - y_q)\, \psi(r_q)$$

$$\partial_{\rho_{x_q}} \phi = \frac{w_q (x - x_q)^2 \rho_{x_q}}{r_q} \psi'(r_q)$$

$$\partial_{\rho_{y_q}} \phi = \frac{w_q (y - y_q)^2 \rho_{y_q}}{r_q} \psi'(r_q)$$

Partial derivatives along x

$$\partial_x \phi = \sum_{q=1}^{N} b_q \psi(r_q) + \frac{w_q (x-x_q) \rho_{x_q}^2 \psi'(r_q)}{r_q}$$

$$\partial_{xx} \phi = \sum_{q=1}^{N} \left[\frac{(w_q + b_q (x-x_q)) \rho_{x_q}^2}{r_q} - \frac{w_q (x-x_q)^2 \rho_{x_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{w_q (x-x_q)^2 \rho_{x_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{xa_q} \phi = \frac{(x-x_q) \rho_{x_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{xxa_q} \phi = \left[\frac{\rho_{x_q}^2}{r_q} - \frac{(x-x_q)^2 \rho_{x_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(x-x_q)^2 \rho_{x_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{xb_q} \phi = \psi(r_q) + \frac{(x-x_q)^2 \rho_{x_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{xxb_q} \phi = \left[\frac{3(x-x_q) \rho_{x_q}^2}{r_q} - \frac{(x-x_q)^3 \rho_{x_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(x-x_q)^3 \rho_{x_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{xc_q} \phi = \frac{(x-x_q)(y-y_q) \rho_{x_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{xxc_q} \phi = \left[\frac{(y-y_q) \rho_{x_q}^2}{r_q} - \frac{(x-x_q)^2 (y-y_q) \rho_{x_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(x-x_q)^2 (y-y_q) \rho_{x_q}^4}{r_q^2} \psi''(r_q)$$

Partial derivatives along y

$$\partial_y \phi = \sum_{q=1}^{N} c_q \psi(r_q) + \frac{w_q (y-y_q) \rho_{y_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{yy} \phi = \sum_{q=1}^{N} \left[\frac{(w_q + c_q (y-y_q)) \rho_{y_q}^2}{r_q} - \frac{w_q (y-y_q)^2 \rho_{y_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{w_q (y-y_q)^2 \rho_{y_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{ya_q} \phi = \frac{(y-y_q) \rho_{y_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{yya_q} \phi = \left[\frac{\rho_{y_q}^2}{r_q} - \frac{(y-y_q)^2 \rho_{y_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(y-y_q)^2 \rho_{y_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{yb_q} \phi = \frac{(x-x_q)(y-y_q) \rho_{y_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{yyb_q} \phi = \left[\frac{(x-x_q) \rho_{y_q}^2}{r_q} - \frac{(x-x_q)(y-y_q)^2 \rho_{y_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(x-x_q)(y-y_q)^2 \rho_{y_q}^4}{r_q^2} \psi''(r_q)$$

$$\partial_{yc_q} \phi = \psi(r_q) + \frac{(y-y_q)^2 \rho_{y_q}^2}{r_q} \psi'(r_q)$$

$$\partial_{yyc_q} \phi = \left[\frac{3(y-y_q) \rho_{y_q}^2}{r_q} - \frac{(y-y_q)^3 \rho_{y_q}^4}{r_q^3} \right] \psi'(r_q) + \frac{(y-y_q)^3 \rho_{y_q}^4}{r_q^2} \psi''(r_q)$$

Accuracy of the staged neural network using synthetic phase

This section presents the performance of the neural network for different types of synthetic phase variation with strong local aliasing. The first test looks at smoothly varying phase. Then, we focus on phase that varies randomly. The non-monotonic nature of the phase variation creates a new set of challenges on top of phase aliasing, especially in the presence of a fragmented mask and high noise levels. We looked at the accuracy of the neural network by computing the error between the ground-truth phase and the output layer, $\varepsilon(x,y) = |\phi_{GT}(x,y) - \phi(x,y)|/(2\pi)$, which is given in units of 2π rather than radians and represents the normalized error with respect to the wrapped phase ϕ_W, which spans an interval of 2π. Each network is trained until the maximum error goes below 10^{-3} or when the overall error cannot be improved.

Quasi-monotonic phase

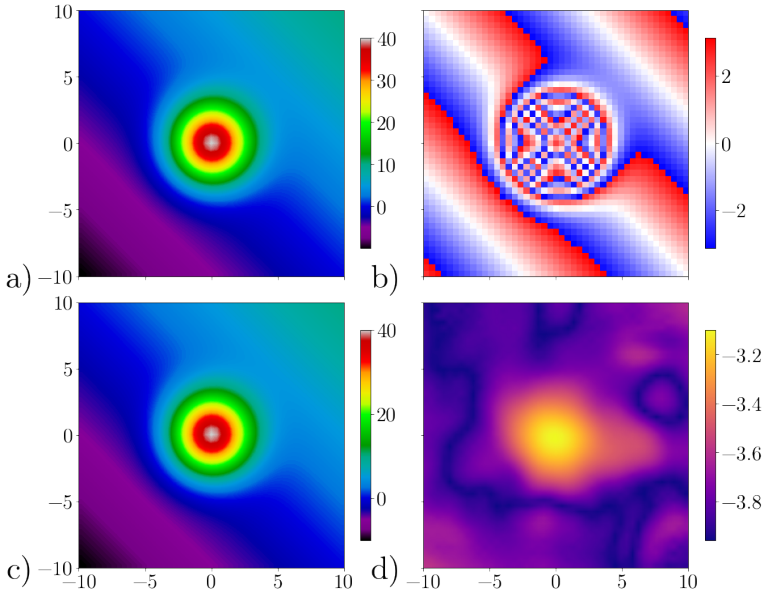

Figure 6. a) The ground truth phase, b) the digitized wrapped phase, c) the RBFNN output, all in radians, and d) the output phase error on the $\log_{10}$ scale.

The quasi-monotonic phase is given by

$$\phi_{GT}(x,y) = \alpha(x+y) + \beta \exp\left(-\frac{x^2+y^2}{\sigma^2}\right). \tag{29}$$

Fig. 6-a shows the initial ground truth phase and Fig. 6-b the digitized wrapped phase with strong aliasing, all in radians. The neural network output layer is virtually identical to ϕ_{GT}. However, the very high accuracy is obtained only after removing a constant bias that exist between the two phases. This bias is not an error. Rather it comes from a lack of absolute reference between the two phases. Since this bias cannot be determined from the wrapped phase shown in Fig. 6-b, we computed this bias to make the average of network output equal to the average of the ground truth and the recovered phase is shown in Fig. 6-c. In reality, we would not have access to this information when performing real phase measurements. But this limitation is physical rather than imposed by the method presented here. For $\beta < 40$, the RBFNN recovers the ground-truth phase from the digitized phase with an error well below 10^{-3}. The error becomes quickly worse with larger values of β. After this correction, Fig. 6-d shows that the maximum error between the RBFNN and the initial phase is less than 0.1%.

Random phase with masked data

When the phase varies randomly across the domain, the neural network cannot exploit any trend to recover the ground truth ϕ_{GT}. If aliasing is introduced, then it becomes very difficult to even attempt the task manually. While Fig. 7-a shows that ϕ_{GT} does not vary wildly, the digitized, wrapped phase in Fig. 7-b shows that a randomly varying phase is in fact relatively difficult to unwrap. Yet the output of the neural network shown in Fig. 7-c matches well ϕ_{GT}, with and error below 0.1% shown in Fig. 7-d. The error is more homogeneously distributed compared to the quasi monotonic phase presented in the previous section, mostly caused by global (rather than local) aliasing. There is very little change of the overall error compared to the unmasked case (not shown). Fig. 8-a shows that aliasing is large enough to cause the wrapped phase to increase smoothly, while the

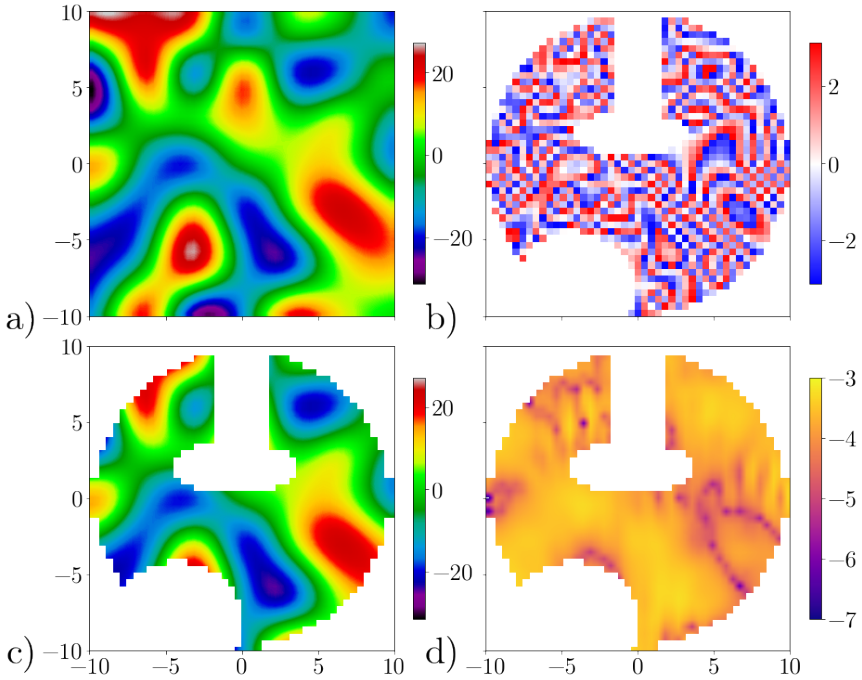

Figure 7. a) The ground truth phase, neural network output phase and wrapped phase of Fig. 6 along the x-direction, together with their b) first and c) second derivatives along the x-direction with mask. d) The output phase error on the $\log_{10}$ scale.

ground-truth phase actually decreases. This happens in regions where the first derivative of ϕ_{GT}, shown in Fig. 8-b, is smaller than $-\pi$, causing $W(\partial\phi_W)$ to wrap around. Note that this wrapping is not problematic since we are using the sine and cosine functions when training our neural network on first derivatives, which continuously vary throughout phase jumps.

Since the neural network is trained on a dataset that contains the first and second derivatives of the phase, we can take the derivatives of the neural network output to estimate the derivatives of the phase. Fig. 8-b shows an excellent agreement with the ground truth phase derivative. We clearly see here that the RBFNN cannot match the left and right first derivatives simultaneously, since they have different values. Rather the RBFNN matches the average, which is the central first derivative. As shown in Eq. 8, the neural network uses the left and right derivatives of ϕ_W to compute the weights use in the output layer. So, the derivative of ϕ, which also matches the derivative of ϕ_{GT}, is located in between the left and right derivatives of ϕ_W, as expected (see Fig. 8-b). As a result, using the error $\hat{\mathbf{e}}$ given in Eq. 18 makes more sense than using $\mathbf{e}$. Based on the assumptions that $\phi_{GT} \in]-\pi,\pi]$ and ϕ_{GT} is continuous, we see that $W(\partial_x^2\phi_W)$ has no jump since $W(\partial_x^2\phi_W) = \partial_x^2\phi_{GT}$.

Fig. 8 shows clearly how the neural network can recover the ground truth ϕ_{GT}, without explicitly unwrapping it. The output of the neural network and its derivatives are continuous by construction, since they are the sum of continuous radial basis functions. At the end of the first stage of the training, the second derivative of the neural network matches directly the second derivative of the wrapped phase, which is continuous since $W(\partial_x^2\phi_W) = \partial_x^2\phi_{GT}$ and $\partial_x^2\phi_{GT}$ is continuous. At the end of the first stage, the output of the neural network is continuous since the output is continuous by construction. At the end of second stage, the network output matches the first derivatives of the wrapped phase via the sine and cosine functions. This approach hides the phase jumps created by the wrapping operator W when aliasing exists. Again, at the end of this stage, the output of the neural network is also continuous since it is the sum of continuous functions. During the third stage, where the network is trained to match the wrapped phase values via the sine and cosine functions, its output again remains continuous. So, the training process forced the output of the neural network to match the sine and cosine of the wrapped phase and the radial basis functions used to build this network forced the output to be continuous, allowing to remove the jumps of the wrapped phase.

Random phase with noise

When there is no aliasing, the noise can be removed from the wrapped phase using standard filtering techniques specifically developed for interferograms, such as the fringe smoothing approach[87], local fringe frequency estimation[88], windowed Fourier filtering[89,90], or Gabor filter local frequency[91]. Any of these techniques can be applied to the wrapped phase before feeding it to the neural network. When filtering the wrapped phase, we can detect locations with noisy data by computing the phase residues and mask out locations where the residues leads to a non conservative result[92], providing that the ground truth phase is conservative (e.g. interferogram of topographic data). Filtering can also be done during the unwrapping procedure[93–97] but

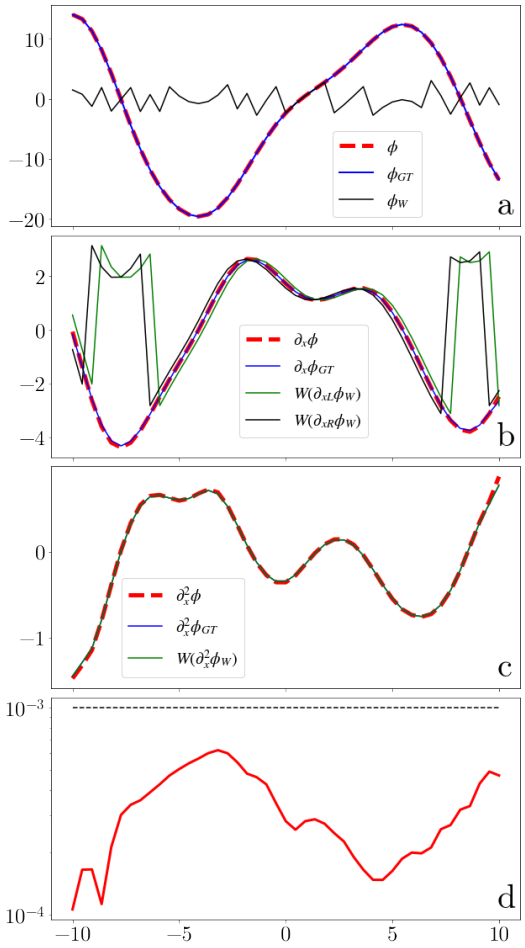

Figure 8. a) The ground truth phase, neural network output phase and wrapped phase of Fig. 6 along the x-direction, together with their b) first and c) second derivatives along the x-direction. d) The output phase error on the $\log_{10}$ scale.

cannot be applied here as the filtering procedure is deeply dependant of the unwrapping method. However, when aliasing is present, direct filtering becomes more problematic. For one, the method of residue cannot be used reliably. Furthermore, aliasing can behave like noise and it becomes difficult to differentiate between good data that was wrapped multiple times and noisy data.

To look at the impact of noise on the neural network performance for strongly aliased phase, we added noise $\mathcal{N}(x,y) \in [-1,1]$ to the wrapped phase as

$$\phi_W(x,y) = W\left(\phi_{GT}(x,y)\right) + \gamma\pi\mathcal{N}(x,y) \tag{30}$$

where γ is a constant controlling the maximum noise level. Fig. 9 shows that the neural network recovers the ground truth ϕ_{GT} with an error that is on the order of the noise level added to the wrapped phase. The neural network tends to perform well for $\gamma < 0.1$, but tend to develop O(1) error when $\gamma > 0.1$. Similar results are found with masked data. So, without specific noise filtering strategy working on the aliased wrapped phase, we find that the neural network remains reliable for noise levels below 10% of the wrapped phase. Larger noise levels will require some filtering beforehand. Fig. 10 shows how the regularization avoids over-fitting of the network output, limiting the impact of the noise on unwrapped phase.

Comparison with other methods

Since the proposed RBFNN gives excellent results compared to cases where the phase is quasi-monotonic or random, with or without noise, it is worth comparing it to standard methods used in the literature. The done here comparison is clearly not exhaustive. Furthermore, we are not trying to show that the RBFNN is superior to other methods. Each method are usually designed with a particular application in mind and a method working well in a given set of conditions can have subpar results

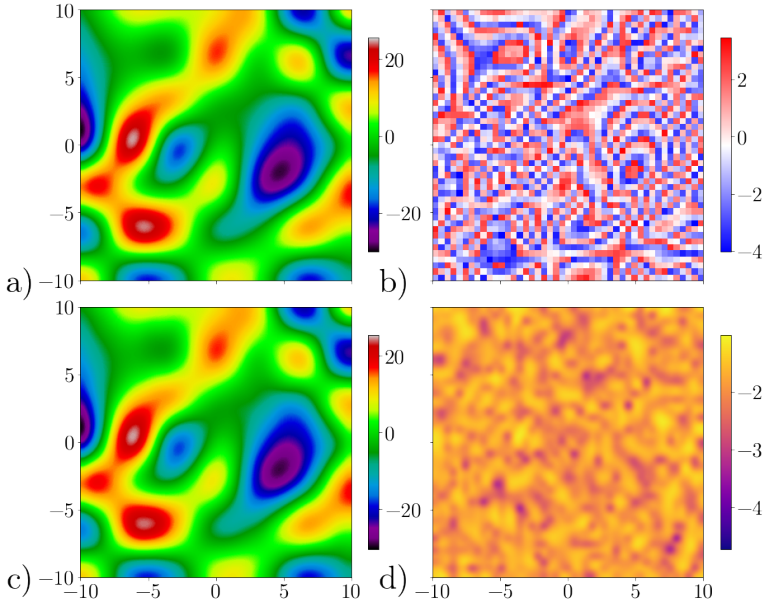

Figure 9. a) The ground truth phase, b) the digitized wrapped phase, c) the RBFNN output, all in radians, and d) the phase error on the $\log_{10}$ scale with a noise level $\gamma = 0.1$ or 10% of the maximum value of the wrapped phase.

when faced with another. Overall, our method requires large Jacobian matrices and should be used when the phase has been heavily digitized or when regions have missing or corrupted phase.

Fig. 11 shows that the RBFNN performs as well as the discrete cosine transform (DCT) of Ref. 13 just before super-resolution is needed, at which points the DCT method fails. Note that the precision of the RBFNN can be improved up to an error of 10^{-4} by reducing the limit of the convergence error, while there is no mechanism to decrease the error of the DCT method, except by increasing the resolution. However, the DCT is extremely fast and returned a solution in 38 ms. The RBFNN required 48 s on a hardware with 48 cores at 2 GHz and for a 50×50 grid to arrive to an error on the order of 10^{-3}. The RBFNN was written in python which can substantially slow down computations. However, all the functions were accelerated using numba, and the Levenberg-Marquardt algorithm was done in parallel. Further acceleration could be gained using GPUs.

We also compared our results against a convolutional Long Short Term Memory (LSTM) network described in Ref. 36. This deep-learning method is one of the best performers with phase unwrapping algorithm according to the authors. The LSTM took 229 s to create the learning set, 1894 s to train on this set and 1 s to compute the solution on a 256×256 grid. Note that the network can actually be trained an order of magnitude faster by using GPUs, but we used CPUs here to keep the same hardware in both tests. Fig. 12-a shows the initial wrapped phase, part of the test deck of the LSTM network python notebook, together with the unwrapped phase done by both methods. Fig. 12-b shows that the LSTM network gave a much higher error for the standard parameters given with the code. We believe that the higher error does not come from an inherent issue of the LSTM network but from the fact than the training set insluded noisy phase. The proposed RBFNN took 220 s for a grid that was 64×64 using the hardware described above.

Once we move to noisy data, the two methods follow ϕ_{GT} in a similar fashion, as shown in Fig. 12-c, the error is noise limited (see Fig. 12-d). It is important to note that the LSTM algorithm requires that the size of the grid of the phase to be unwrapped needs to match the size of the training dataset phase. And there is no provision to allow for masking regions where the phase is not usable. As a result, if each new phase unwrapping requires a different mask, the LSTM network has to go through a new learning step.

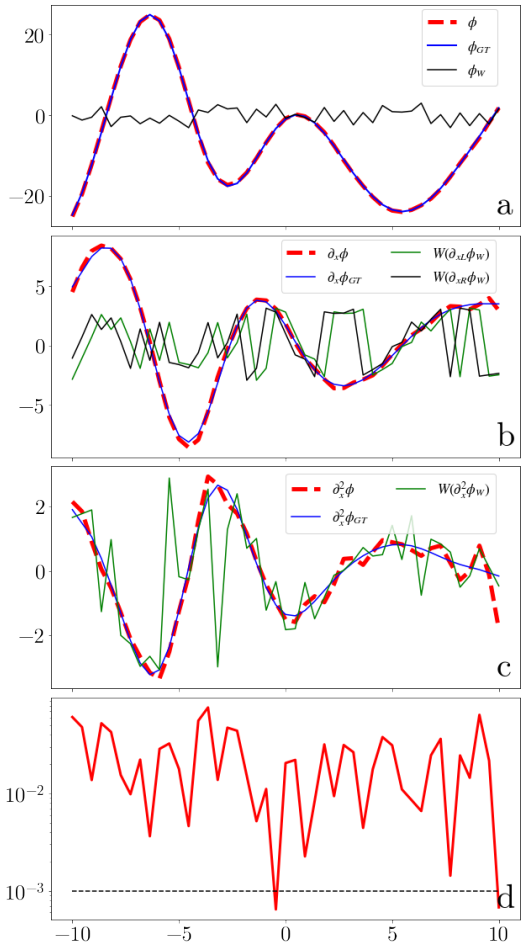

Figure 10. a) The ground truth phase, neural network output phase and wrapped phase of Fig. 9 along the x-direction with a noise level $\gamma = 0.1$ or 10% of the maximum value of the wrapped phase. Their b) first and c) second derivatives along the x-direction. d) The output phase error on the $\log_{10}$ scale.

A note about spatial dimensions

The neural network can be extended to three dimensional phase unwrapping simply by adding the derivatives along the third dimension:

$$\begin{array}{llll} & i_3 = \cos(\partial_{xL}\phi_W) & i_9 = \cos(\partial_{xR}\phi_W) & \\ & i_4 = \sin(\partial_{xL}\phi_W) & i_{10} = \sin(\partial_{xR}\phi_W) & \\ i_1 = \cos(\phi_W) & i_5 = \cos(\partial_{yL}\phi_W) & i_{11} = \cos(\partial_{yR}\phi_W) & i_{15} = W(\partial_{xx}\phi_W) \\ i_2 = \sin(\phi_W) & i_6 = \sin(\partial_{yL}\phi_W) & i_{12} = \sin(\partial_{yR}\phi_W) & i_{16} = W(\partial_{yy}\phi_W) \\ & i_7 = \cos(\partial_{zL}\phi_W) & i_{13} = \cos(\partial_{zR}\phi_W) & i_{17} = W(\partial_{zz}\phi_W) \\ & i_8 = \sin(\partial_{zL}\phi_W) & i_{14} = \sin(\partial_{zR}\phi_W) & \end{array} \tag{31}$$

with the output layer given by $\phi(x,y,z) = \sum_{q=1}^{N} w_q \psi(r_q)$ where $r_q = \sqrt{(x-x_q)^2\rho_{x_q}^2 + (y-y_q)^2\rho_{y_q}^2 + (z-z_q)^2\rho_{z_q}^2}$ and $w_q = a_q + b_q(x-x_q) + c_q(y-y_q) + d_q(z-z_q)$. Using the formulas from Eq. 31, we get the corresponding output values $o_1, \ldots, o_{17}$. Neural networks with even higher number of dimensions can be built trivially by extending this procedure as necessary.

References

1. Oppenheim, A. V. & Lin, J. S. The importance of phase in signals. *Proc. IEEE* **69**, 529–541 (1981).
2. Blackledge, J. *Quantitative Coherent Imaging* (Academic Press, London, 1989).

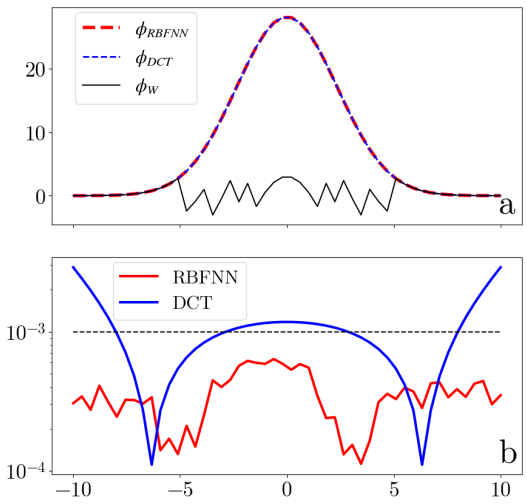

Figure 11. a) The phase unwrapped in radians using the discrete cosine transform (DCT) of Ref. 13 and the proposed RBFNN, together with the wrapped phase. b) The output phase error on the $\log_{10}$ scale.

3. Witoszynskyj, S., Rauscher, A., Reichenbach, J. R. & Barth, M. Phase unwrapping of MR images using ΦUN – a fast and robust region growing algorithm. *Med. Image Analysis* **13**, 257–268 (2009). Includes Special Section on Functional Imaging and Modelling of the Heart.
4. Sarkisov, G. S. Shearing interferometer with air wedge for electron plasma diagnostics in a dense plasma. *Instruments Exp. Tech.* **39**, 110–114 (1996).
5. Stanley, O. W., Kuurstra, A. B., Klassen, L. M., Menon, R. S. & Gati, J. S. Effects of phase regression on high-resolution functional MRI of the primary visual cortex. *NeuroImage* **227**, 117631 (2021).
6. Yu, H., Lan, Y., Yuan, Z., Xu, J. & Lee, H. Phase unwrapping in insar: A review. *IEEE Geosci. Remote. Sens. Mag.* **7**, 40–58 (2019).
7. Zhang, S., Li, X. & Yau, S.-T. Multilevel quality-guided phase unwrapping algorithm for real-time three-dimensional shape reconstruction. *Appl. Opt.* **46**, 50–57 (2007).
8. Gorthi, S. S. & Rastogi, P. Fringe projection techniques: whither we are? *Opt. lasers engineering* **48**, 133–140 (2010).
9. Takeda, M., Ina, H. & Kobayashi, S. Fourier-transform method of fringe-pattern analysis for computer-based topography and interferometry. *J. Opt. Soc. Am.* **72**, 156–160 (1982).
10. Macy, W. W. Two-dimensional fringe-pattern analysis. *Appl. Opt.* **22**, 3898–3901 (1983).
11. Roddier, C. & Roddier, F. Interferogram analysis using fourier transform techniques. *Appl. Opt.* **26**, 1668–1673 (1987).
12. Itoh, K. Analysis of the phase unwrapping algorithm. *Appl. optics* **21**, 2470–2470 (1982).
13. Ghiglia, D. C. & Pritt, M. D. *Two-dimensional phase unwrapping: theory, algorithms, and software* (1998).
14. Cusack, R., Huntley, J. & Goldrein, H. Improved noise-immune phase-unwrapping algorithm. *Appl. Opt.* **34**, 781–789 (1995).
15. Goldstein, R. M. & Werner, C. L. Radar interferogram filtering for geophysical applications. *Geophys. research letters* **25**, 4035–4038 (1998).
16. Zheng, D. & Da, F. A novel algorithm for branch cut phase unwrapping. *Opt. Lasers Eng.* **49**, 609–617 (2011).
17. Ghiglia, D. C. & Romero, L. A. Robust two-dimensional weighted and unweighted phase unwrapping that uses fast transforms and iterative methods. *J. Opt. Soc. Am. A* **11**, 107–117 (1994).
18. Wang, X., Fang, S. & Zhu, X. Weighted least-squares phase unwrapping algorithm based on a non-interfering image of an object. *Appl. Opt.* **56**, 4543–4550 (2017).
19. Katkovnik, V., Astola, J. & Egiazarian, K. Phase local approximation (phasela) technique for phase unwrap from noisy data. *IEEE Transactions on Image Process.* **17**, 833–846 (2008).

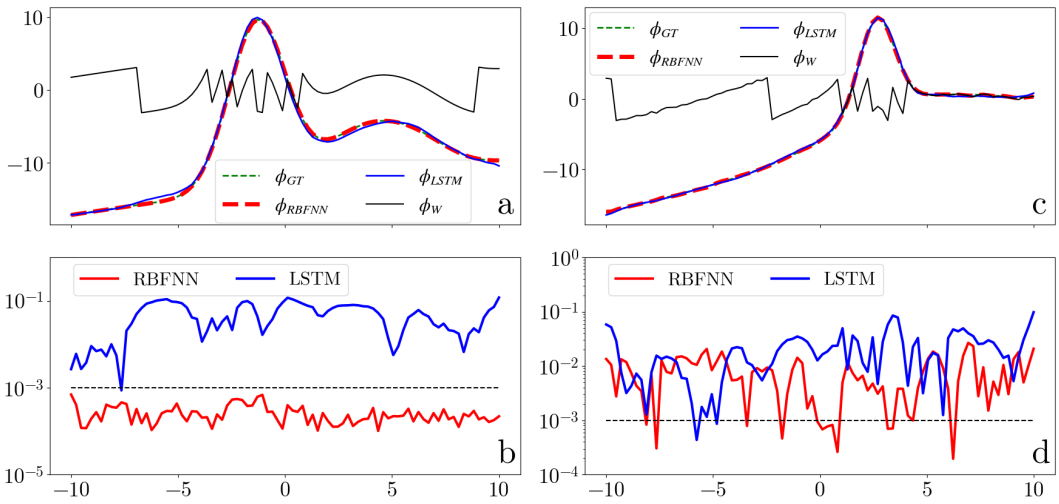

Figure 12. a) The phase unwrapped in radians using the long short term memory network (LSTM) of Ref. 36 and the proposed RBFNN, together with the wrapped phase. b) The output phase error on the $\log_{10}$ scale. c) Another phase unwrapping when noise is present and d) the corresponding error.

20. Gorthi, S. S. & Rastogi, P. Piecewise polynomial phase approximation approach for the analysis of reconstructed interference fields in digital holographic interferometry. *J. Opt. A: Pure Appl. Opt.* **11**, 065405 (2009).

21. Servin, M., Marroquin, J. L., Malacara, D. & Cuevas, F. J. Phase unwrapping with a regularized phase-tracking system. *Appl. Opt.* **37**, 1917–1923 (1998).

22. Loffeld, O., Nies, H., Knedlik, S. & Yu, W. Phase unwrapping for sar interferometry—a data fusion approach by kalman filtering. *IEEE Transactions on Geosci. Remote. Sens.* **46**, 47–58 (2007).

23. Xie, X. & Li, Y. Enhanced phase unwrapping algorithm based on unscented kalman filter, enhanced phase gradient estimator, and path-following strategy. *Appl. Opt.* **53**, 4049–4060 (2014).

24. Xie, X. M. & Zeng, Q. N. Efficient and robust phase unwrapping algorithm based on unscented kalman filter, the strategy of quantizing paths-guided map, and pixel classification strategy. *Appl. Opt.* **54**, 9294–9307 (2015).

25. Cheng, Z. *et al.* Practical phase unwrapping of interferometric fringes based on unscented kalman filter technique. *Opt. Express* **23**, 32337–32349 (2015).

26. Kulkarni, R. & Rastogi, P. Phase unwrapping algorithm using polynomial phase approximation and linear kalman filter. *Appl. optics* **57**, 702–708 (2018).

27. Kalman, R. E. A New Approach to Linear Filtering and Prediction Problems. *J. Basic Eng.* **82**, 35–45 (1960).

28. Julier, S. J. & Uhlmann, J. K. Unscented filtering and nonlinear estimation. *Proc. IEEE* **92**, 401–422 (2004).

29. Schwartzkopf, W., Milner, T., Ghosh, J., Evans, B. & Bovik, A. Two-dimensional phase unwrapping using neural networks. In *4th IEEE Southwest Symposium on Image Analysis and Interpretation*, 274–277 (2000).

30. Wang, K., Li, Y., Kemao, Q., Di, J. & Zhao, J. One-step robust deep learning phase unwrapping. *Opt. express* **27**, 15100–15115 (2019).

31. Yin, W. *et al.* Temporal phase unwrapping using deep learning. *Sci. Reports* **9**, 20175 (2019).

32. Zhang, T. *et al.* Rapid and robust two-dimensional phase unwrapping via deep learning. *Opt. Express* **27**, 23173–23185 (2019).

33. Qin, Y. *et al.* Direct and accurate phase unwrapping with deep neural network. *Appl. Opt.* **59**, 7258–7267 (2020).

34. Wang, K., Kemao, Q., Di, J. & Zhao, J. Deep learning spatial phase unwrapping: a comparative review. *Adv. Photonics Nexus* **1**, 014001 (2022).

35. Yang, F. *et al.* Robust phase unwrapping via deep image prior for quantitative phase imaging. *IEEE Transactions on Image Process.* **30**, 7025–7037 (2021).

36. Perera, M. V. & De Silva, A. A joint convolutional and spatial quad-directional lstm network for phase unwrapping. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4055–4059 (IEEE, 2021).

37. Domier, C. W., Peebles, W. A. & Luhmann, N. C. Millimeter-wave interferometer for measuring plasma electron density. *Rev. Sci. Instruments* **59**, 1588–1590 (1988).

38. Thaury, C. *et al.* Probing electron acceleration and x-ray emission in laser-plasma accelerators. *Phys. Plasmas* **20**, 063101 (2013).

39. Swadling, G. F. *et al.* Diagnosing collisions of magnetized, high energy density plasma flows using a combination of collective thomson scattering, faraday rotation, and interferometry. *Rev. Sci. Instruments* **85**, 11E502 (2014).

40. Lebedev, S. *et al.* Laboratory astrophysics and collimated stellar outflows: The production of radiatively cooled hypersonic plasma jets. *The Astrophys. J.* **564**, 113 (2002).

41. Ampleford, D. *et al.* Supersonic radiatively cooled rotating flows and jets in the laboratory. *Phys. review letters* **100**, 035001 (2008).

42. Hasson, H. R. *et al.* Design of a 3-d printed experimental platform for studying the formation and magnetization of turbulent plasma jets. *IEEE Transactions on Plasma Sci.* **48**, 4056–4067 (2020).

43. Gourdain, P.-A. & Seyler, C. Impact of the hall effect on high-energy-density plasma jets. *Phys. review letters* **110**, 015002 (2013).

44. Long, L. N. & Gupta, A. Scalable massively parallel artificial neural networks. *J. Aerosp. Comput. Information, Commun.* **5**, 3–15 (2008).

45. Kulkarni, R. & Rastogi, P. Direct unwrapped phase estimation in phase shifting interferometry using Levenberg-Marquardt algorithm. *J. Opt.* **19**, 015608 (2017).

46. Gao, P. *et al.* Phase and amplitude reconstruction from a single carrier-frequency interferogram without phase unwrapping. *Appl. Opt.* **47**, 2760–2766 (2008).

47. Broomhead, D. S. & Lowe, D. Multivariable functional interpolation and adaptive networks. *Complex Syst.* **2**, 321–355 (1988).

48. Hardy, R. Multiquadric equations of topography and other irregular surfaces. *J. Geophys. Res.* **176**, 1905–1915 (1971).

49. Park, J. & Sandberg, I. W. Universal approximation using radial-basis-function networks. *Neural computation* **3**, 246–257 (1991).

50. Chen, M.-S. & Manry, M. T. Power series analyses of back-propagation neural networks. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. 1, 295–300 (IEEE, 1991).

51. Manry, M. T., Chandrasekaran, H. & Hsieh, C.-H. Signal processing using the multilayer perceptron. In *Handbook of Neural Network Signal Processing*, 2–1 (CRC Press, 2018).

52. Hornik, K., Stinchcombe, M. & White, H. Multilayer feedforward networks are universal approximators. *Neural networks* **2**, 359–366 (1989).

53. Casasent, D. & Barnard, E. Adaptive clustering neural net for piecewise nonlinear discriminant surfaces. In *1990 IJCNN International Joint Conference on Neural Networks*, 423–428 (IEEE, 1990).

54. Sarajedini, A. & Hecht-Nielson, R. The best of both worlds: Casasent networks integrate multilayer perceptrons and radial basis functions. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, vol. 3, 905–910 (IEEE, 1992).

55. Wendland, H. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv. Comput. Math.* **4**, 389–396 (1995).

56. Wang, Z., Chen, J. & Hoi, S. C. H. Deep learning for image super-resolution: A survey. *IEEE Transactions on Pattern Analysis Mach. Intell.* **43**, 3365–3387 (2021).

57. Bizhani, M., Ardakani, O. H. & Little, E. Reconstructing high fidelity digital rock images using deep convolutional neural networks. *Sci. Reports* **12** (2022).

58. Kulkarni, R. & Rastogi, P. Simultaneous unwrapping and low pass filtering of continuous phase maps based on autoregressive phase model and wrapped kalman filtering. *Opt. Lasers Eng.* **124**, 105826 (2020).

59. Datta, R. & Regis, R. G. A surrogate-assisted evolution strategy for constrained multi-objective optimization. *Expert. Syst. with Appl.* **57**, 270–284 (2016).

60. Müller, J. *et al.* Surrogate optimization of deep neural networks for groundwater predictions. *J. Glob. Optim.* **81**, 203–231 (2020).

61. Kirkpatrick, S., Gelatt Jr, C. D. & Vecchi, M. P. Optimization by simulated annealing. *Science* **220**, 671–680 (1983).

62. Rere, L. R., Fanany, M. I. & Arymurthy, A. M. Simulated annealing algorithm for deep learning. *Procedia Comput. Sci.* **72**, 137–144 (2015). The Third Information Systems International Conference 2015.

63. Gudise, V. G. & Venayagamoorthy, G. K. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)*, 110–117 (IEEE, 2003).

64. Carvalho, M. & Ludermir, T. B. Particle swarm optimization of feed-forward neural networks with weight decay. In *2006 Sixth International Conference on Hybrid Intelligent Systems (HIS'06)*, 5–5 (IEEE, 2006).

65. Buhmann, M. D. *Radial Basis Function Networks*, 823–827 (Springer US, Boston, MA, 2010).

66. Chen, Y., Yang, B. & Dong, J. Time-series prediction using a local linear wavelet neural network. *Neurocomputing* **69**, 449–465 (2006).

67. Nekoukar, V. & Beheshti, M. T. H. A local linear radial basis function neural network for financial time-series forecasting. *Appl. Intell.* **33**, 352–356 (2009).

68. MacKay, D. J. C. Bayesian Interpolation. *Neural Comput.* **4**, 415–447 (1992).

69. Sariev, E. & Germano, G. Bayesian regularized artificial neural networks for the estimation of the probability of default. *Quant. Finance* **20**, 311–328 (2020).

70. Levenberg, K. A method for the solution of certain non-linear problems in least squares. *Q. Appl. Math.* **2**, 164–168 (1944).

71. Marquardt, D. Algorithms for the solution of the nonlinear least-squares problem. *SIAM J. on Numer. Analysis* **11**, 431–441 (1963).

72. Gorinevsky, D. An approach to parametric nonlinear least square optimization and application to task-level learning control. *IEEE Transactions on Autom. Control.* **42**, 912–927 (1997).

73. McLoone, S., Brown, M. D., Irwin, G. & Lightbody, A. A hybrid linear/nonlinear training algorithm for feedforward neural networks. *IEEE transactions on Neural Networks* **9**, 669–684 (1998).

74. Peng, H., Ozaki, T., Haggan-Ozaki, V. & Toyoda, Y. A parameter optimization method for radial basis function type models. *IEEE Transactions on neural networks* **14**, 432–438 (2003).

75. Xie, T., Yu, H., Hewlett, J., Rózycki, P. & Wilamowski, B. Fast and efficient second-order method for training radial basis function networks. *IEEE transactions on neural networks learning systems* **23**, 609–619 (2012).

76. Wilamowski, B. M. & Yu, H. Improved computation for levenberg–marquardt training. *IEEE transactions on neural networks* **21**, 930–937 (2010).

77. Forgy, E. W. Analysis of multivariate data: Efficiency vs interpretabilty of classifications. *Biometrics* **21**, 768–769 (1965).

78. Lloyd, S. P. Least squares quantization in pcm. *IEEE Transactions on Inf. Theory* **28**, 129–137 (1982).

79. Karypis, G. & Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Sci. Comput.* **20**, 359–392 (1998).

80. Ribeiro, A. H. & Aguirre, L. A. "Parallel training considered harmful?": Comparing series-parallel and parallel feedforward network training. *Neurocomputing* **316**, 222–231 (2018).

81. Gunther, S., Ruthotto, L., Schroder, J. B., Cyr, E. C. & Gauger, N. R. Layer-parallel training of deep residual neural networks. *SIAM J. on Math. Data Sci.* **2**, 1–23 (2020).

82. Drake, R. P. *Introduction to high-energy-density physics* (Springer, 2018).

83. Hutchinson, I. H. *Principles of Plasma Diagnostics* (Cambridge University Press, 2002), 2 edn.

84. Gourdain, P.-A. *et al.* The impact of hall physics on magnetized high energy density plasma jets. *Phys. Plasmas* **21**, 056307 (2014).

85. Greenly, J. *et al.* A 1 MA, variable risetime pulse generator for high energy density plasma research. *Rev. Sci. Instruments* **79**, 073501 (2008).

86. Li, X.-F., Huang, L. & Huang, Y. A new abel inversion by means of the integrals of an input function with noise. *J. Phys. A: Math. Theor.* **40**, 347 (2006).

87. Berardino, P., Fornaro, G., Lanari, R. & Sansosti, E. A new algorithm for surface deformation monitoring based on small baseline differential sar interferograms. *IEEE Transactions on geoscience remote sensing* **40**, 2375–2383 (2002).

88. Feng, Q. *et al.* Improved goldstein interferogram filter based on local fringe frequency estimation. *Sensors* **16**, 1976 (2016).

89. Kemao, Q. Two-dimensional windowed fourier transform for fringe pattern analysis: principles, applications and implementations. *Opt. Lasers Eng.* **45**, 304–317 (2007).

90. Kemao, Q., Gao, W. & Wang, H. Windowed fourier-filtered and quality-guided phase-unwrapping algorithm. *Appl. optics* **47**, 5420–5428 (2008).

91. Estrada, J. C., Marroquin, J. L. & Medina, O. M. Reconstruction of local frequencies for recovering the unwrapped phase in optical interferometry. *Sci. Reports* **7**, 1–10 (2017).

92. Goldstein, R. M., Zebker, H. A. & Werner, C. L. Satellite radar interferometry: Two-dimensional phase unwrapping. *Radio science* **23**, 713–720 (1988).

93. Flynn, T. J. Two-dimensional phase unwrapping with minimum weighted discontinuity. *JOSA A* **14**, 2692–2701 (1997).

94. Costantini, M. A novel phase unwrapping method based on network programming. *IEEE Transactions on geoscience remote sensing* **36**, 813–821 (1998).

95. Fornaro, G., Pauciullo, A. & Sansosti, E. Phase difference-based multichannel phase unwrapping. *IEEE Transactions on image processing* **14**, 960–972 (2005).

96. Yu, H., Li, Z. & Bao, Z. Residues cluster-based segmentation and outlier-detection method for large-scale phase unwrapping. *IEEE Transactions on Image Process.* **20**, 2865–2875 (2011).

97. Tayebi, B., Sharif, F. & Han, J.-H. Smart filtering of phase residues in noisy wrapped holograms. *Sci. Reports* **10** (2020).

Acknowledgements

This research was supported by the NSF CAREER Award PHY-1943939.

Author contributions statement

P.-A.G. developed the neural network architecture. He also tested the accuracy of the neural network on random phases and plasma interferograms. A.B. tested the accuracy of the neural network for quasi-monotonic phases

Sample code

We provide here a sample code in python, which uses the serial version of the RBFNN.

Libraries and basic function definitions

```
#constants#
N_EQ=12 #number of equations per neuron
N_DOF=5 #number of degrees of freedom per neuron
tiny=1e-15 #tiny number

#libraries
import numpy as np
import math
from math import sin, cos, sqrt
from numpy.linalg import norm, solve
from numba import jit

#Wendland (3,2) radial basis function without check on r (i.e. 0<r<1 needs to be checked before use) and its derivatives
@jit(nopython=True)
def rbf(r):
    return 1 + r**2*(-9.333333333333334 + r**2*(70 + r*(-149.33333333333334 + r*(140 + r*(-64 + (35*r)/3.)))))
@jit(nopython=True)
def drbf(r):
    return r*(-18.666666666666668 + r**2*(280 + r*(-746.6666666666666 + r*(840 + r*(-448 + (280*r)/3.)))))
@jit(nopython=True)
def d2rbf(r):
    return -18.666666666666668 + r**2*(840 + r*(-2986.6666666666665 + r*(4200 + r*(-2688 + (1960*r)/3.))))

# wrapping functions
@jit(nopython=True)
```

```
def W(x):
    while (x>np.pi):
        x-=2.*np.pi
    while (x<=-np.pi):
        x+=2.*np.pi
    return x

@jit(nopython=True)
def WF(x):
    for i in range (len(x)):
        x[i]=W(x[i])
    return x

@jit(nopython=True)
def WF2D(x):
    y=np.copy(x)
    for i in range (len(x[:,0])):
        for j in range (len(x[0,:])):
            y[i,j]=W(x[i,j])
    return y

#derivatives
@jit(nopython=True)
def DTR(x): #first right derivative
    n=len(x)
    y=np.zeros(n)
    for i in range(0,n-1):
        y[i]=x[i+1]-x[i]
    y[n-1]=y[n-3]+(y[n-2]-y[n-3])/((n-2)-(n-3))*((n-1)-(n-3))
    return y

@jit(nopython=True)
def DTL(x): # first left derivative
    n=len(x)
    y=np.zeros(n)
    for i in range(1,n):
        y[i]=x[i]-x[i-1]
    y[0]=y[2]+(y[1]-y[2])/((1)-(2))*((0)-(2))
    return y

@jit(nopython=True)
def DTC(x): #first centered derivative
    n=len(x)
    y=np.zeros(n)
    for i in range(1,n-1):
        y[i]=(x[i+1]-x[i-1])*.5
    y[0]=y[2]+(y[1]-y[2])/((1)-(2))*((0)-(2))
    y[n-1]=y[n-3]+(y[n-2]-y[n-3])/((n-2)-(n-3))*((n-1)-(n-3))
    return y

@jit(nopython=True)
def D2T(x): #second (centered) derivative
    n=len(x)
    y=np.zeros(n)
    for i in range(1,n-1):
        y[i]=(x[i+1]-2*x[i]+x[i-1])
    y[0]=y[2]+(y[1]-y[2])/((1)-(2))*((0)-(2))
    y[n-1]=y[n-3]+(y[n-2]-y[n-3])/((n-2)-(n-3))*((n-1)-(n-3))
    return y
```

Grid functions

```
#turns grid data into gridless data
def unpack_data(phi,mask):
    ni=len(phi[:,0])
    nj=len(phi[0,:])
    Nt=0
    for j in range (nj):
        for i in range (ni):
            if (mask[i,j]==1):
                Nt+=1
    data=np.zeros(Nt*N_EQ)
    grid=np.zeros(Nt*2)
    k=0
    d_phi_dx_R=np.zeros((ni,nj))
    d_phi_dx_L=np.zeros((ni,nj))
    d2_phi_dx2=np.zeros((ni,nj))
    for j in range (nj):
        d_phi_dx_R[:,j]=DTR(phi[:,j])
        d_phi_dx_L[:,j]=DTL(phi[:,j])
        d2_phi_dx2[:,j]=WF(D2T(phi[:,j]))
    d_phi_dy_R=np.zeros((ni,nj))
    d_phi_dy_L=np.zeros((ni,nj))
```

```
    d2_phi_dy2=np.zeros((ni,nj))
    for i in range (ni):
        d_phi_dy_R[i,:]=DTR(phi[i,:])
        d_phi_dy_L[i,:]=DTL(phi[i,:])
        d2_phi_dy2[i,:]=WF(D2T(phi[i,:]))
    for j in range (nj):
        for i in range (ni):
            if (mask[i,j]==1):
                grid[k*2+0]=i
                grid[k*2+1]=j
                data[k*N_EQ+0]=cos(phi[i,j])
                data[k*N_EQ+1]=sin(phi[i,j])
                data[k*N_EQ+2]=cos(d_phi_dx_L[i,j])
                data[k*N_EQ+3]=sin(d_phi_dx_L[i,j])
                data[k*N_EQ+4]=cos(d_phi_dy_L[i,j])
                data[k*N_EQ+5]=sin(d_phi_dy_L[i,j])
                data[k*N_EQ+6]=cos(d_phi_dx_R[i,j])
                data[k*N_EQ+7]=sin(d_phi_dx_R[i,j])
                data[k*N_EQ+8]=cos(d_phi_dy_R[i,j])
                data[k*N_EQ+9]=sin(d_phi_dy_R[i,j])
                data[k*N_EQ+10]=d2_phi_dx2[i,j]
                data[k*N_EQ+11]=d2_phi_dy2[i,j]
                k+=1
    return data,grid

#turns gridless data into grid data
def repack_data(w,grid,phi_in,Nt,sigma=5):
    ni=len(phi_in[:,0])
    nj=len(phi_in[0,:])
    n=Nt
    phi=np.zeros((ni,nj))
    d_phi_dx=np.zeros((ni,nj))
    d2_phi_dx2=np.zeros((ni,nj))
    d_phi_dy=np.zeros((ni,nj))
    d2_phi_dy2=np.zeros((ni,nj))
    sigma2=sigma**2
    for i in range (n):
        x=int(grid[i*2+0])
        y=int(grid[i*2+1])
        phi[x,y]=0
        for j in range (n):
            xj=grid[j*2+0]
            yj=grid[j*2+1]
            sxj=abs(w[j*N_DOF+3])
            syj=abs(w[j*N_DOF+4])
            rj2=(x-xj)**2*sxj**2+(y-yj)**2*syj**2
            if (rj2<1):
                rj2+=tiny
                rj=sqrt(rj2)
                aj=w[j*N_DOF+0]
                bj=w[j*N_DOF+1]
                cj=w[j*N_DOF+2]
                wj = aj + bj*(x - xj) + cj*(y - yj)
                phi[x,y] += wj*rbf(rj)
                d_phi_dx[x,y]   += (sxj**2*wj*(x - xj)*drbf(rj))/rj + bj*rbf(rj)
                d2_phi_dx2[x,y] += (sxj**4*wj*(x - xj)**2*d2rbf(rj))/rj2 \
                                 + ((sxj**2*wj)/rj + (2.*bj*sxj**2*(x - xj))/rj - (sxj**4*wj*(x - xj)**2)/rj2**1.5)*drbf(rj)
                d_phi_dy[x,y]   += (syj**2*wj*(y - yj)*drbf(rj))/rj + cj*rbf(rj)
                d2_phi_dy2[x,y] += (syj**4*wj*(y - yj)**2*d2rbf(rj))/rj2 \
                                 + ((syj**2*wj)/rj + (2.*cj*syj**2*(y - yj))/rj - (syj**4*wj*(y - yj)**2)/rj2**1.5)*drbf(rj)
    return phi,d_phi_dx,d_phi_dy,d2_phi_dx2,d2_phi_dy2
```

Optimization functions

```
#computation of the output layer and Jacobian
@jit(nopython=True,cache =False)
def comp_data(grid,phi_s,jac,Nt,w,sigma,comp_jac=True,full=True,fuller=True):
    if(comp_jac):
        jac=np.zeros(jac.shape)
    n=Nt
    sigma2=sigma**2
    for i in range(n):
        x=grid[i*2+0]
        y=grid[i*2+1]
        phi=0
        d_phi_dx=0
        d_phi_dy=0
        d2_phi_dx2=0
        d2_phi_dy2=0
        for j in range(n):
            xj=grid[j*2+0]
            yj=grid[j*2+1]
            sxj=abs(w[j*N_DOF+3])
```

```
        syj=abs(w[j*N_DOF+4])
        rj2=(x-xj)**2*sxj**2+(y-yj)**2*syj**2
        if (rj2<1):
            rj2+=tiny
            rj=sqrt(rj2)
            aj=w[j*N_DOF+0]
            bj=w[j*N_DOF+1]
            cj=w[j*N_DOF+2]
            wj = aj + bj*(x - xj) + cj*(y - yj)
            phi += wj*rbf(rj)
            d_phi_dx += (sxj**2*wj*(x - xj)*drbf(rj))/rj + bj*rbf(rj)
            d2_phi_dx2 += (sxj**4*wj*(x - xj)**2*d2rbf(rj))/rj2 \
                        + ((sxj**2*wj)/rj + (2.*bj*sxj**2*(x - xj))/rj \
                        - (sxj**4*wj*(x - xj)**2)/rj2**1.5)*drbf(rj)
            d_phi_dy += (syj**2*wj*(y - yj)*drbf(rj))/rj + cj*rbf(rj)
            d2_phi_dy2 += (syj**4*wj*(y - yj)**2*d2rbf(rj))/rj2 \
                        + ((syj**2*wj)/rj + (2.*cj*syj**2*(y - yj))/rj
                        - (syj**4*wj*(y - yj)**2)/rj2**1.5)*drbf(rj)

    phi_s[i*N_EQ+0]=cos(phi) #eqn 0
    phi_s[i*N_EQ+1]=sin(phi) #eqn 1
    phi_s[i*N_EQ+2]=cos(d_phi_dx)  #eqn 2
    phi_s[i*N_EQ+3]=sin(d_phi_dx)  #eqn 3
    phi_s[i*N_EQ+4]=cos(d_phi_dy)  #eqn 4
    phi_s[i*N_EQ+5]=sin(d_phi_dy)  #eqn 5
    phi_s[i*N_EQ+6:i*N_EQ+10]=phi_s[i*N_EQ+2:i*N_EQ+6] #eqn 6 to 9
    phi_s[i*N_EQ+10]=d2_phi_dx2  #eqn 10
    phi_s[i*N_EQ+11]=d2_phi_dy2  #eqn 11

    if (comp_jac):
        for j in range (n):
            xj=grid[j*2+0]
            yj=grid[j*2+1]
            sxj=abs(w[j*N_DOF+3])
            syj=abs(w[j*N_DOF+4])
            rj2=(x-xj)**2*sxj**2+(y-yj)**2*syj**2
            if (rj2<1):
                rj2+=tiny
                rj=sqrt(rj2)
                aj=w[j*N_DOF+0]
                bj=w[j*N_DOF+1]
                cj=w[j*N_DOF+2]
                wj = aj + bj*(x - xj) + cj*(y - yj)

                d_phi_daj = rbf(rj)
                d_phi_dbj = (x - xj)*rbf(rj)
                d_phi_dcj = (y - yj)*rbf(rj)
                d_phi_dsxj = (sxj*wj*(x - xj)**2*drbf(rj))/rj
                d_phi_dsyj = (syj*wj*(y - yj)**2*drbf(rj))/rj

                d2_phi_dx_daj = (sxj**2*(x - xj)*drbf(rj))/rj
                d2_phi_dx_dbj = (sxj**2*(x - xj)**2*drbf(rj))/rj + rbf(rj)
                d2_phi_dx_dcj = (sxj**2*(x - xj)*(y - yj)*drbf(rj))/rj

                d3_phi_dx2_daj = (sxj**4*(x - xj)**2*d2rbf(rj))/rj2 \
                               + ((sxj**2)/rj - (sxj**4*(x - xj)**2)/rj2**1.5)*drbf(rj)
                d3_phi_dx2_dbj = (sxj**4*(x - xj)**3*d2rbf(rj))/rj2 \
                               + ((3.*sxj**2*(x - xj))/rj - (sxj**4*(x - xj)**3)/rj2**1.5)*drbf(rj)
                d3_phi_dx2_dcj = (sxj**4*(x - xj)**2*(y - yj)*d2rbf(rj))/rj2 \
                               + ((sxj**2*(y - yj))/rj - (sxj**4*(x - xj)**2*(y - yj))/rj2**1.5)*drbf(rj)

                d2_phi_dy_daj = (syj**2*(y - yj)*drbf(rj))/rj
                d2_phi_dy_dbj = (syj**2*(x - xj)*(y - yj)*drbf(rj))/rj
                d2_phi_dy_dcj = (syj**2*(y - yj)**2*drbf(rj))/rj + rbf(rj)

                d3_phi_dy2_daj = (syj**4*(y - yj)**2*d2rbf(rj))/rj2 \
                               + ((syj**2)/rj - (syj**4*(y - yj)**2)/rj2**1.5)*drbf(rj)
                d3_phi_dy2_dbj = (syj**4*(x - xj)*(y - yj)**2*d2rbf(rj))/rj2 \
                               + ((syj**2*(x - xj))/rj - (syj**4*(x - xj)*(y - yj)**2)/rj2**1.5)*drbf(rj)
                d3_phi_dy2_dcj = (syj**4*(y - yj)**3*d2rbf(rj))/rj2 \
                               + ((3.*syj**2*(y - yj))/rj - (syj**4*(y - yj)**3)/rj2**1.5)*drbf(rj)

                if (fuller):
                    #d cos(phi) / d aj->cj
                    eqn=0
                    jac[i*N_EQ+eqn,j*N_DOF+0]=d_phi_daj
                    jac[i*N_EQ+eqn,j*N_DOF+1]=d_phi_dbj
                    jac[i*N_EQ+eqn,j*N_DOF+2]=d_phi_dcj
                    jac[i*N_EQ+eqn,j*N_DOF+3]=d_phi_dsxj
                    jac[i*N_EQ+eqn,j*N_DOF+4]=d_phi_dsyj
                    jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=-sin(phi)

                    #d sin(phi) / d aj->cj
```

```
                        eqn=1
                        jac[i*N_EQ+eqn,j*N_DOF+0]=d_phi_daj
                        jac[i*N_EQ+eqn,j*N_DOF+1]=d_phi_dbj
                        jac[i*N_EQ+eqn,j*N_DOF+2]=d_phi_dcj
                        jac[i*N_EQ+eqn,j*N_DOF+3]=d_phi_dsxj
                        jac[i*N_EQ+eqn,j*N_DOF+4]=d_phi_dsyj
                        jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=cos(phi)

                    if (full):
                        #d cos(d_phi_dx) / d aj->cj
                        eqn=2
                        jac[i*N_EQ+eqn,j*N_DOF+0]=d2_phi_dx_daj
                        jac[i*N_EQ+eqn,j*N_DOF+1]=d2_phi_dx_dbj
                        jac[i*N_EQ+eqn,j*N_DOF+2]=d2_phi_dx_dcj
                        jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=-sin(d_phi_dx)

                        #d sin(d_phi_dx) / d aj->cj
                        eqn=3
                        jac[i*N_EQ+eqn,j*N_DOF+0]=d2_phi_dx_daj
                        jac[i*N_EQ+eqn,j*N_DOF+1]=d2_phi_dx_dbj
                        jac[i*N_EQ+eqn,j*N_DOF+2]=d2_phi_dx_dcj
                        jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=cos(d_phi_dx)

                        #d cos(d_phi_dy) / d aj->cj
                        eqn=4
                        jac[i*N_EQ+eqn,j*N_DOF+0]=d2_phi_dy_daj
                        jac[i*N_EQ+eqn,j*N_DOF+1]=d2_phi_dy_dbj
                        jac[i*N_EQ+eqn,j*N_DOF+2]=d2_phi_dy_dcj
                        jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=-sin(d_phi_dy)

                        #d sin(d_phi_dy) / d aj->cj
                        eqn=5
                        jac[i*N_EQ+eqn,j*N_DOF+0]=d2_phi_dy_daj
                        jac[i*N_EQ+eqn,j*N_DOF+1]=d2_phi_dy_dbj
                        jac[i*N_EQ+eqn,j*N_DOF+2]=d2_phi_dy_dcj
                        jac[i*N_EQ+eqn,j*N_DOF+0:j*N_DOF+N_DOF]*=cos(d_phi_dy)

                        #repeat above for both spatial derivatives
                        jac[i*N_EQ+6:i*N_EQ+10,j*N_DOF+0:j*N_DOF+N_DOF]=jac[i*N_EQ+2:i*N_EQ+6,j*N_DOF+0:j*N_DOF+N_DOF]

                    #d3_phi_dx2 / d aj->cj
                    eqn=10
                    jac[i*N_EQ+eqn,j*N_DOF+0]=d3_phi_dx2_daj
                    jac[i*N_EQ+eqn,j*N_DOF+1]=d3_phi_dx2_dbj
                    jac[i*N_EQ+eqn,j*N_DOF+2]=d3_phi_dx2_dcj

                    #d3_phi_dy2 / d aj->cj
                    eqn=11
                    jac[i*N_EQ+eqn,j*N_DOF+0]=d3_phi_dy2_daj
                    jac[i*N_EQ+eqn,j*N_DOF+1]=d3_phi_dy2_dbj
                    jac[i*N_EQ+eqn,j*N_DOF+2]=d3_phi_dy2_dcj

        return phi_s,jac

    #the staged Levenberg-Marquardt algorithm
    def LMA_2D(data,grid,mask,sigma=5,filtering=1):
        Ns=80
        ms1=10 #initially we only try to match the second derivatives, not the phase.
        ms2=20 #then we only try to match the second and first derivatives, not the phase. When m>ms2 then we match all
        init_lmb=.001 #smaller is better with noise lambda~1e-3
        lmb_mult=5
        max_lmb=1e20
        old_err=1e15
        stage=1
        #compute target vector
        #empty vectors necessary to avoid multiple creations to avoid memory leaks
        Nt=int(len(grid)/2)
        phi_d=np.copy(data)
        #initialiaze weights
        w=np.zeros(N_DOF*Nt)
        sp=1
        for i in range(Nt):
            w[i*N_DOF+0]=np.random.normal(-sp,sp)
            w[i*N_DOF+1]=np.random.normal(-sp,sp)
            w[i*N_DOF+2]=np.random.normal(-sp,sp)
            w[i*N_DOF+3]=1./(sigma)
            w[i*N_DOF+4]=1./(sigma+2)
        jac=np.zeros((Nt*N_EQ,N_DOF*Nt))
        phi_s=np.zeros(Nt*N_EQ)
        Id=np.eye(len(w))
        #compute solution vector
        #start LMA
        lmb=init_lmb
```

```
for m in range(Ns):
    if (m==ms1):
        lmb=init_lmb #we restart with
        old_error=1e15
        stage+=1
    if (m==ms2):
        lmb=init_lmb #we restart with
        old_error=1e15
        stage+=1
    phi_s,jac=comp_data(grid,phi_s,jac,Nt,w,sigma,True,m>=ms1,m>=ms2)
    jact=jac.transpose()
    jjt=jact.dot(jac)

    #compute optimal lamba
    delta_phi=phi_d-phi_s
    err=norm(delta_phi)
    init_err=err
    err+=1
    while (err>init_err):
        dw=np.linalg.solve(jjt+lmb*Id,jact.dot(delta_phi))
        w_new=w+dw
        phi_s,jac=comp_data(grid,phi_s,jac,Nt,w_new,sigma,False,m>=ms1,m>=ms2)
        err=norm(phi_d-phi_s)
        if (err!=err):
            return w_new,grid,Nt,jac
        if (m==0):
            start_err=err
        if (err<init_err):
            w=w_new
            lmb/=lmb_mult
            if (lmb<1./max_lmb):
                lmb=1./max_lmb
            break
        lmb*=lmb_mult
        if (lmb>max_lmb):
            lmb=max_lmb
            break
    errc=corrected_error(phi_s,phi_d)/Nt
    #keep matching derivatives ONLY until error is acceptable
    if ((abs(errc-old_err)<1e-3 or errc<1e-3) and m<ms1):
        ms1=m+1
    if ((abs(errc-old_err)<1e-3 or errc<1e-3) and m>5+ms1 and m<ms2):
        ms2=m+1
    #match both derivatives AND wrapped phase value
    if ((abs(errc-old_err)<1e-6 or errc<1e-5) and m>5+ms2):
        break
    clear_output(wait=True)
    print(f"{'Stage:':<15}{stage:>10}",
          f"\n{'Steps:':<15}{m:>10}",
          f"\n{'Lambda:':<15}{lmb:>10}",
          f"\n{'Error:':<15}{errc:>10}")
    old_err=errc
return w,grid,Nt,jac
```