Applications of algorithmic differentiation to phase retrieval algorithms

Alden S. Jurling and James R. Fienup*

Institute of Optics, University of Rochester, Rochester, New York 14627, USA
**Corresponding author: fienup@optics.rochester.edu*

Received April 2, 2014; revised April 4, 2014; accepted April 22, 2014;
posted April 25, 2014 (Doc. ID 200832); published June 9, 2014

In this paper, we generalize the techniques of reverse-mode algorithmic differentiation to include elementary operations on multidimensional arrays of complex numbers. We explore the application of the algorithmic differentiation to phase retrieval error metrics and show that reverse-mode algorithmic differentiation provides a framework for straightforward calculation of gradients of complicated error metrics without resorting to finite differences or laborious symbolic differentiation.

OCIS codes: (100.5070) Phase retrieval; (000.3860) Mathematical methods in physics; (000.4430) Numerical approximation and analysis; (010.7350) Wave-front sensing.
http://dx.doi.org/10.1364/JOSAA.31.001348

1. INTRODUCTION

Phase retrieval algorithms have seen a variety of applications in optics, including wavefront sensing and image reconstruction. Many of these algorithms employ a nonlinear-optimization approach, where an error metric (typically some form of data consistency metric, possibly modified with penalty terms to enforce additional constraints) is optimized (or minimized) using a gradient-based search algorithm to find a solution to the problem. This approach has three advantages: the forward model can be modified in a straightforward way to account for real-world effects such as broadband light or detector integration area, the error metric can be tuned to the particular noise statistics present in the data, and the search strategy can benefit from the extensive literature in nonlinear search algorithms. These advantages come with a price: the better-performing nonlinear search algorithms (such as conjugate gradient or BFGS [1]) require computation of the gradient of the error metric. Phase retrieval and image reconstruction algorithms typically optimize over many parameters, particularly in the case of a point-by-point phase or amplitude function for wavefront sensing or image reconstruction. In order to maintain computational efficiency, it is desirable to avoid using a finite-difference gradient calculation (whose cost scales linearly with the number of parameters). For a high-order polynomial wavefront or point-by-point aberration, a finite-difference gradient could cost hundreds or thousands of times the cost of a single error metric evaluation. Instead, analytic gradient formulations are employed [2,3], with costs on the same order as a single error metric evaluation. In previous work wherein analytic gradients are reported, they are typically derived by writing an explicit expression for the error metric and symbolically differentiating with respect to each of the input parameters to produce an expression for the gradient. This is mathematically straightforward, but somewhat laborious, and it limits the choice of forward models to those that the researcher understands how to differentiate symbolically. This leads to the following algorithm implementation and testing strategy:

1. Write forward model and error metric symbolically.
2. Implement forward model in computer and test.
3. Derive symbolic expression for gradients of the forward model.
4. Implement gradient model in computer and test.

The researcher is essentially writing two closely related pieces of software that together make up the complete routine. If this approach is followed rigorously, a small change in the forward model can require repeating a large amount of work in the gradient derivation to produce an appropriate new gradient. We and others have observed recurring patterns in the form of these gradient calculations; even with changes in the choice of error metric and propagation models, the overall flow of the calculation remains the same. We have also observed that, for phase retrieval algorithms, the computational cost to compute the gradient (having already computed the forward model) is approximately the same as the cost to compute the forward model itself. The cost of computing the forward model is also largely independent of the number of parameters; that is, the cost of computing the forward model and gradient is approximately twice that of computing the forward model alone.

The literature in the field of algorithmic differentiation [4] reveals that both of these observations reflect more general principles. The recurring computational structure of the phase retrieval analytic gradient is an *ad hoc* manual implementation of a reverse-mode algorithmic differentiation strategy. The observation that the cost of computing the gradient is a small multiple of the cost of computing the forward model is actually a manifestation of the "cheap derivative principle." This principle applies when one employs the reverse-mode strategy to compute the gradient of a scalar function of many variables. The algorithmic differentiation literature [4] also provides several key additional insights:

1. The cheap derivative principle applies to almost all well-behaved scalar functions, not just the largely linear operations typically employed in phase retrieval algorithms. The cheap derivative principle is also not limited to functions that can be expressed directly in a symbolic mathematical form but extends to functions that are described by algorithms and computer codes, provided that the elementary operations in the algorithm are differentiable in the vicinity of the evaluation point. Since any numerical function can be ultimately expressed as a sequence of floating-point additions and multiplications, which are differentiable, this is not a significant limitation in practice.

2. The software structure of the forward-model code provides the structure for the derivative code; it is possible to produce the derivatives on a step-by-step basis by reversing the steps in the forward model and applying appropriate differentiation rules.

3. As a consequence of the previous point, if a forward model can be divided into reusable subroutines, the gradient calculation can be subdivided in the same way. Therefore, we can modularize the components of a phase retrieval algorithm and its gradient. This means that different image-domain error metrics, numerical propagation models, and pupil models can potentially be used together in a "mix and match" fashion without requiring additional gradient derivations.

4. The generation of analytic gradients can be done entirely mechanistically in software, either by recording the execution history of the forward model and playing it back to compute the gradient or by making source code transformations on the forward model itself.

The fourth point suggests that computing analytic gradients for arbitrary functions is essentially a solved problem; indeed, various software packages implement algorithmic differentiation techniques in various languages. However, these packages have some limitations and design choices that hinder their application for phase retrieval algorithms. First, the formulation in [4] is primarily written for scalar operations, whereas phase retrieval algorithms are most naturally expressed in terms of multidimensional vector representations. Second, the formulation in [4] is given in terms of real-valued functions, while phase retrieval algorithms are typically expressed at least partly in terms of complex fields and complex-valued Fourier representations. Third, phase retrieval algorithms rely heavily on specialized numerical routines such as the fast Fourier transform (FFT), which are typically provided by third-party libraries; these libraries appear as a black box to typical algorithmic differentiation tools, and derivative computation cannot proceed through them unless the tool is provided with customized derivative rules for the particular library. Finally, algorithmic differentiation tools are most well developed for lower-level compiled languages such C, C++, and Fortran, while many phase retrieval algorithms are implemented in high-level interpreted languages such as MATLAB, where the tools are not as well developed and/or are expensive and proprietary. We will revisit the topic of current algorithmic differentiation tools in Appendix A.

Fortunately, it is not necessary to possess an automated tool in order to benefit from the concepts and techniques of algorithmic differentiation. Instead, so-called "manual automatic differentiation" requires only that the programmer understand the approach; the actual gradient code is produced by hand. The technique of algorithmic differentiation provides a recipe to convert the forward-model code into the reverse-model gradient code in a systematic and straightforward way.

In this paper, we extend the existing algorithmic differentiation techniques in order to apply them to the complex-valued multidimensional arrays employed in phase retrieval algorithms, as well as the commonly occurring special numerical functions. We will provide the "recipe" for operations that occur commonly in phase retrieval algorithms and show how these components can be connected to compute gradients for more complicated algorithms.

The cost of the traditional symbolic approach to computing analytic gradients can be seen by considering some examples. Equations (11) and (12) in [5] give lengthy analytic gradients for the optical field for a phase retrieval algorithm that models point-spread functions with undersampled broadband light. This is in contrast with the description of the forward model, which is broken down into individually digestible steps in Eqs. (2)–(6). Using the algorithmic differentiation techniques we will discuss here, the same step-by-step approach could be applied to the analytic gradient calculation. Furthermore, since many of the elements of the forward model in [5] are in common with those of the algorithm for the simpler monochromatic and adequately sampled case, only the novel elements of this particular algorithm, in this case mainly Eq. (6) of [5], would require new gradient derivations. As another example, consider Eqs. (C6) and (C20) in [6], which together give gradients for wavefront parameters in a problem with an unknown extended object. Again the gradient expression is lengthy and required substantial work to construct. Again an algorithmic differentiation approach to the gradient would greatly reduce the required effort and allow focusing on the novel aspect of the forward model (in this case convolution with unknown extended objects).

The advantages of the algorithmic differentiation approach to constructing phase retrieval algorithms can also be seen when combining elements from existing models to solve new problems. As a motivating example, consider modeling the following situation. We take through-focus images of a small object with a segmented-aperture telescope. The detected images are undersampled in intensity by more than a factor of two (so that the underlying optical fields are also undersampled) and furthermore were taken through a broadband filter. We wish to determine if the small object is a single star, two or more stars near one another, or a small extended object. Modeling this scenario involves combining the extended object convolution models of [6] with the undersampled imaging models of [5]. If the analytic gradients for this case were derived in the usual direct symbolic method, much of the work in both papers would have to be repeated to produce the gradient expression of the new model, even though the forward model would be a straightforward combination of two. If the two subproblems were treated with the algorithmic differentiation approach described later in this paper, the two gradient models could be combined simply to produce the correct gradient for the combined problem.

The remainder of this paper is organized as follows: Section 2 introduces the notation we will use to distinguish scalar and array quantities. Section 3 sketches the forward model for a simple phase retrieval problem to provide a

concrete context for the work in the remaining sections. Section 4 develops the basic mathematics of reverse-mode algorithmic differentiation for real-valued variables. Section 5 extends the treatment of algorithmic differentiation to include complex-valued variables. Section 6 gives reverse-mode gradient rules for many common arithmetic operations on complex-valued array quantities. Section 7 addresses some of the subtleties of calculations that mix real and complex variables. Section 8 develops the gradient rules for general linear operators in detail. Section 9 develops a reverse-mode gradient rule for an algorithm for downsampling images by pixel binning (boxcar averaging), which can be used for undersampled phase retrieval algorithms. Section 10 revisits the phase retrieval algorithm from Section 3 and applies the reverse-mode rules developed in the previous sections to it. Section 11 shows how the simple model from Sections 3 and 10 can be modified to incorporate nonlinear detector response. Section 12 summarizes the paper and concludes. Finally, Appendix A reviews currently available software tools for algorithmic differentiation in MATLAB and Python.

2. NOTATIONAL CONVENTIONS

In phase retrieval and image reconstruction algorithms, we are typically interested in numerical operations over two- or higher-dimensional arrays of real or complex numbers (typically represented as single or double precision floating-point numbers in a computer). In order to express algorithms using these arrays more compactly, we will adopt the following conventions: a variable written in italics, like x, is a scalar variable. A variable written in bold italics, like $\boldsymbol{y}$, is a vector or higher-dimensional array. An italics subscripted variable denotes a scalar element of the corresponding bold variable; e.g., y_n is the nth element of $\boldsymbol{y}$. A bold subscripted variable denotes one of a collection of vectors or tensors; e.g., in a focus-diverse phase retrieval problem, the two-dimensional array of measured data in the nth plane might be denoted $\boldsymbol{D}_n$.

Additionally, the application of scalar operations to every element of an array is a very common feature of our algorithms, so we will adopt the convention that a scalar function applied to a bold italics variable denotes the application of the scalar function to each element of the input vector to produce an output vector of the same size and shape. That is,

$$\boldsymbol{y} = f(\boldsymbol{x}) \Rightarrow y_n = f(x_n). \tag{1}$$

In order to accommodate this we do not extend the notation of bold indicating a vector quantity to indicate a vector-valued function; an unbolded function may be either scalar or vector depending on the context. Additionally, although the element-wise multiplication of two vectors or matrices (called a Hadamard product) is somewhat unusual in abstract linear algebra, it is quite common in phase retrieval algorithms, while the more typically usual matrix multiplication is relatively unusual in phase retrieval algorithms. We will adopt the convention that the element-wise product is denoted by $\boldsymbol{x} \circ \boldsymbol{y}$, with the implicit requirement that $\boldsymbol{x}$ and $\boldsymbol{y}$ must have the same size and shape. The matrix product, on the other hand, is denoted here by $\boldsymbol{x} * \boldsymbol{y}$, with the implicit requirement that the shapes of $\boldsymbol{x}$ and $\boldsymbol{y}$ must be compatible for matrix multiplication. Note that this use of the in-line asterisk is different from the convention adopted by some authors, where $\boldsymbol{x} * \boldsymbol{y}$ denotes a convolution. To avoid confusion between element-wise products and matrix products, in this paper we reserve the implied multiplication for scalar–scalar and scalar–vector products; that is, xy and $x\boldsymbol{y}$ are both valid expressions, but $\boldsymbol{xy}$ is not. We will define the superscripted power operators applied to arrays as their scalar counterparts applied to the whole array, e.g.,

$$[\boldsymbol{x}^2]_n = x_n^2. \tag{2}$$

Finally, since we are primarily interested in deriving gradient relations for numerical programs implemented in the computer, we do not formally distinguish between the set of real numbers and the set of floating-point numbers; both will be denoted by $\mathbb{R}$. Likewise, the set of complex numbers represented as pairs of real numbers or pairs of floating-point numbers will both be represented by $\mathbb{C}$.

3. SIMPLE NONLINEAR-OPTIMIZATION PHASE RETRIEVAL ALGORITHM

In order to place the techniques developed in the remainder of the paper in a more concrete context, we will review the structure of a simple nonlinear-optimization-based phase retrieval algorithm. We will consider a simple conventional case: a system with a known pupil transmittance and unknown low-order Zernike aberrations illuminated by monochromatic light with point-spread functions detected with adequate sampling in a plane near focus. The forward model for this problem begins with the numerical specification of the pupil. Let the transmittance of the pupil be described by the array $\boldsymbol{A}$ and the wavefront aberrations of the system by the array $\boldsymbol{W}$, in units of optical path difference from a perfect converging spherical wave. The wavefront $\boldsymbol{W}$ will be given by the basis expansion

$$\boldsymbol{W} = \sum_n a_n \boldsymbol{Z}_n, \tag{3}$$

where $\boldsymbol{Z}_n$ are sampled Zernike basis functions and $\boldsymbol{a}$ is a vector of coefficients. The complex optical field in the pupil of the system is

$$\boldsymbol{g} = \boldsymbol{A} \circ \exp\left(\frac{i2\pi}{\lambda}\boldsymbol{W}\right), \tag{4}$$

where λ denotes the wavelength, and recall that $\circ$ denotes element-wise multiplication. Under the Fraunhofer approximation we can propagate the optical field in the pupil plane to the image plane with a FFT

$$\boldsymbol{G} = \mathrm{FFT}\{\boldsymbol{g}\}. \tag{5}$$

If the point-spread function is desired in a defocused plane, the focus term in $\boldsymbol{a}$ should be adjusted. For simplicity we here assume that the sampling is controlled by zero-padding $\boldsymbol{A}$ around the known transmittance function and that the overall power in $\boldsymbol{G}$ is controlled by adjusting the scale of the amplitude $\boldsymbol{A}$. The observed point-spread function intensity will be simply the squared magnitude of the optical field in the image plane:

$$\boldsymbol{I} = |\boldsymbol{G}|^2. \tag{6}$$

Given measured point-spread function data $\boldsymbol{D}$ we can form a sum-of-squared-differences error metric

$$E = \sum_m w_m (I_m - D_m)^2, \tag{7}$$

where m is a lexicographic index over image pixels and $\boldsymbol{w}$ is a weighting function used to reject bad pixels and control the normalization of the error metric if desired. $\boldsymbol{I}$ is indirectly a function of $\boldsymbol{a}$, the estimated Zernike coefficients. For a given choice of the error metric, E will describe how much our modeled point-spread data $\boldsymbol{I}$ disagrees with the measured data $\boldsymbol{D}$. In order to find the unknown Zernike coefficients, we wish to find the Zernike coefficients that minimize the error metric:

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} E. \tag{8}$$

Since Eqs. (4) and (6) are nonlinear, this minimization problem cannot be solved with simple linear techniques; instead it requires nonlinear-optimization techniques. The nonlinear-optimization approach we employ is to begin at a starting point $\boldsymbol{a}^{(0)}$ and search along directions that reduce E using gradient-based search algorithms such as nonlinear conjugate gradient or BFGS [1]. The number of terms in $\boldsymbol{a}$ varies greatly from problem to problem. If the system is dominated by a single aberration (e.g., spherical aberration) a single parameter may be sufficient. For a system with only low-order aberrations, 15 terms may be sufficient. For systems with significant higher-order aberrations, hundreds of terms may be required. Ultimately for systems with very high-order aberrations, the polynomial model in Eq. (3) can be replaced with a point-by-point phase map, which could have thousands or hundreds of thousands of pixels in it. Because of the high dimensionality of the search space in these problems, it is imperative to be able to compute the required search directions in a way whose cost does not increase proportionally with the number of parameters. Traditionally this was done as a single monolithic calculation using differential calculus; for this forward model, that would yield the following:

$$\begin{aligned} \overline{\boldsymbol{W}} &= \frac{2\pi}{\lambda} \Im\{\mathrm{IFFT}[4\boldsymbol{w} \circ (\boldsymbol{I} - \boldsymbol{D}) \circ \boldsymbol{G}^*] \circ g^*\}, \\ \frac{\partial E}{\partial a_n} &= \sum_p (\overline{\boldsymbol{W}} \circ Z_n)_p, \end{aligned} \tag{9}$$

where $\overline{\boldsymbol{W}}$ represents an array of the partial derivatives with respect to the individual pixels in the wavefront, i.e.,

$$\overline{W}_p = \frac{\partial E}{\partial W_p}, \tag{10}$$

and $\Im\{\bullet\}$ denotes taking the imaginary part. Equation (9) is equivalent to combining Eqs. (19) and (47) in [3] after accounting for the error in Eq. (47) of leaving off, in the second and third lines, the factor $W(u)$ in the first line. Most of the computational cost of Eq. (9) is in computing $\overline{\boldsymbol{W}}$, which is then used to compute all of the partial derivatives for the individual coefficients of $\boldsymbol{a}$. In the point-by-point phase map case, $\boldsymbol{W}$ itself is the wavefront model so $\overline{\boldsymbol{W}}$ is the required gradient with respect to individual pixel values. The techniques discussed in the following sections will allow us to construct a function that computes the same numerical results as Eq. (9) without having to actually derive and write down the full expression; instead we may work locally with the steps in the forward model to propagate gradients through the calculation. This will mean that, for example, the work needed to account for the error metric in Eq. (7) need not be repeated if we change the propagation method in Eq. (5) or the pupil model in Eqs. (3) and (4). This extends to the inclusion of more complicated effects such as extended objects, detector undersampling, segmented apertures, and chromatic effects, although we will not explore the details of all of those models in this paper.

4. BASICS OF ALGORITHMIC DIFFERENTIATION

In this section, we review algorithmic differentiation. For a detailed overview of these concepts, see [4]. Algorithmic differentiation is based on the chain rule for partial derivatives. We will use the form of the chain rule for partial derivatives given in [7]. If we have a function f defined by

$$f = f(x_1, x_2, \ldots, x_n), \tag{11}$$

where the $x's$ are given by

$$x_i = x_i(u_1, u_2, \ldots, u_m), \tag{12}$$

then the partial derivatives of f with respect to the u's are given by

$$\frac{\partial f}{\partial u_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial u_j}. \tag{13}$$

The transformation u to x is essentially a change of variables; the final value f is the same whether it is expressed as a function of the x or u variables. In the context of a numerical calculation, the u variables would represent an earlier set of intermediate variables, and the x variables would represent a later set of intermediate variables. All that is required to compute the gradient with respect to the u variables is to already know the gradient with respect to the x variables and to know the slope of the x variables with respect to the u variables at the point of interest.

Let us consider a more complicated scenario:

$$\boldsymbol{v}_1 = f_1(\boldsymbol{x}), \qquad \boldsymbol{v}_2 = f_2(\boldsymbol{v}_1), \qquad E = f_3(\boldsymbol{v}_2), \tag{14}$$

where bold denotes a vector quantity (so f_1 and f_2 are implicitly vector-valued functions). We wish to ultimately obtain the derivatives of E with respect to the elements of $\boldsymbol{x}$. Working from the end, we can use the chain rule for partial derivatives to break this apart. The third line above gives us

$$\frac{\partial E}{\partial \boldsymbol{v}_2} = f_3'(\boldsymbol{v}_2), \tag{15}$$

where prime denotes the first derivative of the (scalar) function f_3 with respect to each input variable, and the vector in the denominator indicates a vector of derivatives. This could also be written in the form

$$\left(\frac{\partial E}{\partial \boldsymbol{v}_2}\right)_k \equiv \frac{\partial E}{\partial v_{2,k}}, \tag{16}$$

where k indexes the elements of $\boldsymbol{v}_2$. Alternately, one can write it as an explicit gradient,

$$\frac{\partial E}{\partial \boldsymbol{v}_2} \equiv \nabla_{\boldsymbol{v}_2} E. \tag{17}$$

This initial step does not require the chain rule since it involves only a single function. Working backward, we can apply the chain rule to the next-to-last step to compute the partial derivatives of E with respect to the wth element of $\boldsymbol{v}_1$:

$$\left(\frac{\partial E}{\partial \boldsymbol{v}_1}\right)_w = \sum_k \frac{\partial E}{\partial v_{2,k}} \frac{\partial v_{2,k}}{\partial v_{1,w}}. \tag{18}$$

The first term in the product is the quantity we computed in the previous step, and the second is made of entries from the Jacobian of the function f_2. We can apply the chain rule again to compute the final gradients:

$$\left(\frac{\partial E}{\partial \boldsymbol{x}}\right)_r = \sum_w \frac{\partial E}{\partial v_{1,w}} \frac{\partial v_{1,w}}{\partial x_r}. \tag{19}$$

Again, the first term in the product is the quantity we computed in the previous step, and the second is made of entries from the Jacobian of the function f_1. This general strategy can be applied to any longer sequence of intermediate calculations. Consider a general case in which

$$\boldsymbol{v}_k = f_k(\boldsymbol{v}_{k-1}). \tag{20}$$

Assuming we know the gradients of E with respect to $\boldsymbol{v}_k$, we can compute the gradients of E with respect to $\boldsymbol{v}_{k-1}$ as

$$\frac{\partial E}{\partial v_{k-1,w}} = \sum_q \frac{\partial E}{\partial v_{k,q}} \frac{\partial v_{k,q}}{\partial v_{k-1,w}}. \tag{21}$$

This process can be repeated reverse iteratively from the output variable to the input variable to propagate the gradient dependencies through all the steps in the calculation. Note that although the f_k functions are not necessarily themselves linear, the gradients are transformed linearly at each step. Also, each step need only be aware of its own input and output variables; its role in the larger calculation has no impact on how the derivatives are propagated through the calculation. In [4], the authors adopt the notation that differentiation of the final dependent variable is denoted by an overbar, so that

$$\bar{x} \equiv \frac{\partial E}{\partial x} \tag{22}$$

if the final dependent variable is E. They do not explicitly distinguish between scalar and vector x, but we will: if $\boldsymbol{x}$ is a vector, $\bar{\boldsymbol{x}}$ will be a vector of partial derivatives, so that

$$\bar{x}_k \equiv \frac{\partial E}{\partial x_k} \tag{23}$$

or

$$\bar{\boldsymbol{x}} = \nabla_x E. \tag{24}$$

Because the metrics we are considering are scalar functions that ultimately yield a single output variable, the use of the overbar is unambiguous: it always refers to differentiation of that final scalar output variable. Let us consider an example using this notation. Let

$$\boldsymbol{y} = f(\boldsymbol{x}), \tag{25}$$

and let its Jacobian be given by

$$f'_{mn}(\boldsymbol{x}) = \frac{\partial y_m}{\partial x_n}, \tag{26}$$

where it should be taken that f is one step in a larger calculation that will ultimately produce the scalar quantity E. Our goal is, given $\bar{\boldsymbol{y}}$, to compute $\bar{\boldsymbol{x}}$. If we let the final scalar quantity be E, then the two overbar vectors are given by

$$\bar{x}_n \equiv \frac{\partial E}{\partial x_n}, \qquad \bar{y}_m \equiv \frac{\partial E}{\partial y_m}. \tag{27}$$

We can find $\bar{\boldsymbol{x}}$ using the chain rule for partial derivatives:

$$\frac{\partial E}{\partial x_n} = \sum_m \frac{\partial E}{\partial y_m} \frac{\partial y_m}{\partial x_n}, \qquad \bar{x}_n = \sum_m \bar{y}_m f'_{mn}(\boldsymbol{x}). \tag{28}$$

Notice that the second part of Eq. (28) computes $\bar{\boldsymbol{x}}$ as a function of $\boldsymbol{x}$ and $\bar{\boldsymbol{y}}$. We will take this relationship as defining a function that we will call $\bar{f}(\boldsymbol{x}, \bar{\boldsymbol{y}})$, so that we can write

$$\bar{x}_n = [\bar{f}(\boldsymbol{x}, \bar{\boldsymbol{y}})]_n = \sum_m \bar{y}_m f'_{mn}(\boldsymbol{x}) \tag{29}$$

or in vector form

$$\bar{\boldsymbol{x}} \equiv \bar{f}(\boldsymbol{x}, \bar{\boldsymbol{y}}). \tag{30}$$

That is, if f is some function that maps its domain ($\boldsymbol{x}$) to its range ($\boldsymbol{y}$), then $\bar{f}$ is the function that maps a gradient with respect to $\boldsymbol{y}$ ($\bar{\boldsymbol{y}}$) to a gradient with respect to $\boldsymbol{x}$ ($\bar{\boldsymbol{x}}$). Because we can write Eq. (28) for a general function, we know that $\bar{f}$ exists and depends only on $\boldsymbol{x}$ and $\bar{\boldsymbol{y}}$. Equation (28) defines a procedure for evaluating $\bar{f}$ using the Jacobian of f; this provides a formal definition of $\bar{f}$, but in general for a particular f, it is not required to explicitly compute the Jacobian.

Notice that Eq. (28) is a matrix product between $\bar{\boldsymbol{y}}$ and the Jacobian matrix of f, which we could also write as

$$\bar{f}(\boldsymbol{x}, \bar{\boldsymbol{y}}) = \bar{\boldsymbol{y}} * f'(\boldsymbol{x}) \tag{31}$$

if we interpret $\bar{\boldsymbol{y}}$ as a column vector (which is the convention in [4]). This emphasizes the simple linearity of the derivative propagation. Note, however, that in the rest of this paper we are interested in arbitrary multidimension variables, so we will not adopt the column vector format employed in [4].

Although the matrix product description of the chain rule given above is mathematically correct, in general it is not necessary to explicitly calculate the Jacobian and perform the multiplication in order to propagate derivatives. Instead, for each elementary numerical operation that makes up a program, an appropriate manipulation on the gradient variables can be derived. Before deriving these elementary operations,

let us illustrate a generalized numerical function G that computes an error metric E in terms of a vector of parameters $\boldsymbol{x}_0$, so that the total metric is

$$E = G(\boldsymbol{x}_0). \tag{32}$$

We wish to extend it so that we can also compute $\bar{\boldsymbol{x}}_0$, the gradient with respect to $\boldsymbol{x}_0$. Let us assume the forward model for G has the form

$$\begin{aligned} \boldsymbol{x}_1 &= f_1(\boldsymbol{x}_0), \\ \boldsymbol{x}_2 &= f_2(\boldsymbol{x}_1), \\ &\vdots \\ x_n &= f_n(\boldsymbol{x}_{n-1}), \\ E &= x_n. \end{aligned} \tag{33}$$

Note that x_n is a scalar while the other $\boldsymbol{x}$ variables are vectors. Let us assume for the moment that we know how to construct the $\bar{f}_m(\boldsymbol{x}, \bar{\boldsymbol{x}})$ functions, defined analogously with $\bar{f}$ in Eq. (31), corresponding to the f_m functions in the forward model. Then to propagate gradients, the reverse-mode form of the gradient will be

$$\begin{aligned} \overline{E} &= 1, \\ \bar{x}_n &= \overline{E}, \\ \bar{\boldsymbol{x}}_{n-1} &= \bar{f}_n(\boldsymbol{x}_{n-1}, \bar{x}_n) \\ &\vdots \\ \bar{\boldsymbol{x}}_0 &= \bar{f}_1(\boldsymbol{x}_0, \bar{\boldsymbol{x}}_1). \end{aligned} \tag{34}$$

This shows the general pattern of a reverse-mode gradient calculation: for every step in the forward model, there is a corresponding step in the gradient model, carried out in the reverse order.

5. EXTENSION TO COMPLEX VARIABLES

We wish to derive an extension of the algorithmic differentiation procedure and the overbar gradient notation to include complex variables. To clarify, we do not wish to generalize the algorithmic differentiation procedure to encompass the differential operator of complex analysis, where the domain of applicability would then be functions that are differentiable in the complex sense (i.e., satisfying the Cauchy–Riemann equations). Instead, we wish to generalize the approach to the practice typically seen in numerical computing, where complex numbers are stored and processed in the computer in terms of their real and imaginary parts. Additionally, we will require that the final scalar variable denoted by the overbar is a real scalar variable, so that for gradient calculations we are always differentiating that real scalar variable with respect to some other intermediate variable. It is tempting to think that, even under these stipulations, the complex-differential operator may be the appropriate choice in dealing with intermediate calculations involving complex numbers. However, many simple operations, such as taking the real or imaginary part of a complex number, do not satisfy the Cauchy–Riemann equations and cannot be addressed via a complex differentiation. Since these operations are indispensable in numerical computing with complex numbers, we must have a framework that is not limited to complex-differentiable functions. As long as the models we are interested in can be thought of ultimately as some vector of real-valued parameters leading to a real-valued scalar result, we do not lose generality or introduce an approximation by ignoring requirements for complex differentiability.

Any complex number stored in the computer can be thought of as two real numbers, so that for every function from N complex numbers onto M complex numbers, there is an equivalent function from $2N$ real numbers to $2M$ real numbers. We will apply the concepts of conventional real-variable calculus and algorithmic differentiation to these real numbers. Given the complex-vector-valued function

$$\boldsymbol{y} = f(\boldsymbol{x}), \tag{35}$$

we wish to preserve and extend the overbar notation for differentiation with respect to an implicit final output variable and the definition of a gradient propagation for a function so that we can write

$$\bar{\boldsymbol{x}} = \bar{f}(\boldsymbol{x}, \bar{\boldsymbol{y}}) \tag{36}$$

with an appropriate interpretation for complex-valued functions. We can make the following definitions:

$$\begin{aligned} \boldsymbol{y} &\equiv \boldsymbol{y}_{\mathrm{R}} + i\boldsymbol{y}_{\mathrm{I}}, \\ \boldsymbol{x} &\equiv \boldsymbol{x}_{\mathrm{R}} + i\boldsymbol{x}_{\mathrm{I}}, \\ \boldsymbol{y}_{\mathrm{R}} &\equiv f_{\mathrm{R}}(\boldsymbol{x}_{\mathrm{R}}, \boldsymbol{x}_{\mathrm{I}}), \\ \boldsymbol{y}_{\mathrm{I}} &\equiv f_{\mathrm{I}}(\boldsymbol{x}_{\mathrm{R}}, \boldsymbol{x}_{\mathrm{I}}). \end{aligned} \tag{37}$$

In this section, we will assume the final dependent variable of interest is E, some scalar error metric. We can define the overbar gradient variables as

$$\begin{aligned} \bar{x}_{\mathrm{R},n} &= \frac{\partial E}{\partial x_{\mathrm{R},n}}, \\ \bar{x}_{\mathrm{I},n} &= \frac{\partial E}{\partial x_{\mathrm{I},n}}, \\ \bar{y}_{\mathrm{R},n} &= \frac{\partial E}{\partial y_{\mathrm{R},n}}, \\ \bar{y}_{\mathrm{I},n} &= \frac{\partial E}{\partial y_{\mathrm{I},n}}. \end{aligned} \tag{38}$$

Essentially, the problem here is the following: given $\bar{\boldsymbol{y}}_{\mathrm{R}}$ and $\bar{\boldsymbol{y}}_{\mathrm{I}}$, compute $\bar{\boldsymbol{x}}_{\mathrm{R}}$ and $\bar{\boldsymbol{x}}_{\mathrm{I}}$. We can do this with the chain rule,

$$\begin{aligned} \bar{x}_{\mathrm{R},n} &= \sum_m \frac{\partial E}{\partial y_{\mathrm{R},m}} \frac{\partial y_{\mathrm{R},m}}{\partial x_{\mathrm{R},n}} + \sum_m \frac{\partial E}{\partial y_{\mathrm{I},m}} \frac{\partial y_{\mathrm{I},m}}{\partial x_{\mathrm{R},n}} \\ &= \sum_m \bar{y}_{\mathrm{R},m} \frac{\partial f_{\mathrm{R},m}(x_{\mathrm{R}}, x_{\mathrm{I}})}{\partial x_{\mathrm{R},n}} + \sum_m \bar{y}_{\mathrm{I},m} \frac{\partial f_{\mathrm{I},m}(x_{\mathrm{R}}, x_{\mathrm{I}})}{\partial x_{\mathrm{R},n}} \\ &\equiv \bar{f}_{\mathrm{R},n}(\boldsymbol{x}_{\mathrm{R}}, \boldsymbol{x}_{\mathrm{I}}; \bar{\boldsymbol{y}}_{\mathrm{R}}, \bar{\boldsymbol{y}}_{\mathrm{I}}), \\ \bar{x}_{\mathrm{I},n} &= \sum_m \frac{\partial E}{\partial y_{\mathrm{R},m}} \frac{\partial y_{\mathrm{R},m}}{\partial x_{\mathrm{I},n}} + \sum_m \frac{\partial E}{\partial y_{\mathrm{I},m}} \frac{\partial y_{\mathrm{I},m}}{\partial x_{\mathrm{I},n}} \\ &= \sum_m \bar{y}_{\mathrm{R},m} \frac{\partial f_{\mathrm{R},m}(x_{\mathrm{R}}, x_{\mathrm{I}})}{\partial x_{\mathrm{I},n}} + \sum_m \bar{y}_{\mathrm{I},m} \frac{\partial f_{\mathrm{I},m}(x_{\mathrm{R}}, x_{\mathrm{I}})}{\partial x_{\mathrm{I},n}} \\ &\equiv \bar{f}_{\mathrm{I},n}(\boldsymbol{x}_{\mathrm{R}}, \boldsymbol{x}_{\mathrm{I}}; \bar{\boldsymbol{y}}_{\mathrm{R}}, \bar{\boldsymbol{y}}_{\mathrm{I}}). \end{aligned} \tag{39}$$

If we now define the complex extensions of the barred gradient variables as

$$\bar{\boldsymbol{y}} \equiv \bar{\boldsymbol{y}}_{\mathrm{R}} + i\bar{\boldsymbol{y}}_{\mathrm{I}}, \qquad \bar{\boldsymbol{x}} \equiv \bar{\boldsymbol{x}}_{\mathrm{R}} + i\bar{\boldsymbol{x}}_{\mathrm{I}}, \tag{40}$$

then we can define $\bar{f}(\boldsymbol{x},\bar{\boldsymbol{y}})$ by

$$\bar{f}(\boldsymbol{x},\bar{\boldsymbol{y}}) \equiv \bar{f}_{\mathrm{R}}(\boldsymbol{x}_{\mathrm{R}},\boldsymbol{x}_{\mathrm{I}};\bar{\boldsymbol{y}}_{\mathrm{R}},\bar{\boldsymbol{y}}_{\mathrm{I}}) + i\bar{f}_{\mathrm{I}}(\boldsymbol{x}_{\mathrm{R}},\boldsymbol{x}_{\mathrm{I}};\bar{\boldsymbol{y}}_{\mathrm{R}},\bar{\boldsymbol{y}}_{\mathrm{I}}). \tag{41}$$

Equations (39), (40), and (41) together define the meaning of Eq. (36) and provide the extension of Eq. (30) to complex variables. Our principal remaining task is to determine $\bar{f}(\boldsymbol{x},\bar{\boldsymbol{y}})$ for common operations of interest in phase retrieval error metrics; Eq. (39) defines a procedure for propagating derivatives for a general function, but we can derive a form of $\bar{f}(\boldsymbol{x},\bar{\boldsymbol{y}})$ based on the particular form of $f(\boldsymbol{x})$.

6. GRADIENT PROPAGATION RULES FOR COMMON OPERATIONS

In this section we will show, in Table 1, the gradient propagation rules for common operations. We will write the gradient propagation rules assuming that a particular intermediate variable appears only in the expression under consideration. However, in practice, this is often not the case. When the same intermediate variable is used in multiple parts of a calculation, it is necessary to add the gradient contributions from the different components. That is, given this situation,

$$\begin{aligned} \boldsymbol{y}_1 &= f_1(\boldsymbol{x}), \\ \boldsymbol{y}_2 &= f_2(\boldsymbol{x}), \end{aligned} \tag{42}$$

where E depends on both $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ in an unspecified general way, the gradient with respect to $\boldsymbol{x}$ is given by

$$\bar{\boldsymbol{x}} = \bar{f}_1(\boldsymbol{x},\bar{\boldsymbol{y}}_1) + \bar{f}_2(\boldsymbol{x},\bar{\boldsymbol{y}}_2). \tag{43}$$

In principle, it is possible to extend this to a general case in which a given variable appears at several places in the forward model by summing over gradient contributions for each place where the variable appears. In practice this may be an error-prone process: it may be difficult to track everywhere that a given variable appears in a computer program. We would like to preserve the correspondence from Eq. (34), that one step in the forward model corresponds to one step in the reverse model, without introducing this nonlocal summation. This can be accomplished by using what [4] refers to as the incremental update. For a step in the forward model

$$\boldsymbol{y} = f(\boldsymbol{x}), \tag{44}$$

we implement the gradient for this step in the reverse model using incremental updates as

$$\bar{\boldsymbol{x}} \leftarrow \bar{\boldsymbol{x}} + \bar{f}(\boldsymbol{x},\bar{\boldsymbol{y}}), \tag{45}$$

where the left arrow indicates assignment. That is, the value of $\bar{\boldsymbol{x}}$ is updated with the contribution from the particular function in question. This update is interpreted as a concrete operation on mutable variables in a computer code, not as an abstract mathematical relationship between variables. The variable $\bar{\boldsymbol{x}}$ can be thought of in practice as an accumulator; every statement in the forward model involving $\boldsymbol{x}$ produces a gradient rule in the reverse model that updates the $\bar{\boldsymbol{x}}$ accumulator with the gradient contribution arriving from that forward-model term. The gradient in Eq. (43) would instead be written in two steps as

$$\begin{aligned} \bar{\boldsymbol{x}} &\leftarrow \bar{\boldsymbol{x}} + \bar{f}_2(\boldsymbol{x},\bar{\boldsymbol{y}}_2), \\ \bar{\boldsymbol{x}} &\leftarrow \bar{\boldsymbol{x}} + \bar{f}_1(\boldsymbol{x},\bar{\boldsymbol{y}}_1), \end{aligned} \tag{46}$$

which now correspond to the steps in Eq. (42). Note that strict use of the incremental update requires that $\bar{\boldsymbol{x}}$ be initialized. At the first place $\bar{\boldsymbol{x}}$ is used in the gradient calculation (corresponding to the last place $\boldsymbol{x}$ was used in the forward model), $\bar{\boldsymbol{x}}$ should be initialized to an array of zeros of the same size as $\boldsymbol{x}$. This initialization can be avoided by identifying the first use of $\bar{\boldsymbol{x}}$ and using the gradient formulas as written here for that point. Note that this use of the incremental update is, strictly speaking, optional; it is possible to instead track every case in which the same variable appears in more than one operation and extend the logic of Eq. (43) by explicitly summing over them.

In order to apply these methods in practice, the researcher needs to know which intermediate results are required to compute the gradient. We have formally written the gradient propagation function above as a function of $\boldsymbol{x}$ and $\bar{\boldsymbol{y}}$, but for some nonlinear functions the gradient can be computed more efficiently as a function of $\bar{\boldsymbol{y}}$ and $\boldsymbol{y}$, or of $\bar{\boldsymbol{y}}$, $\boldsymbol{x}$, and $\boldsymbol{y}$. Since $\boldsymbol{y}$ can be computed from $\boldsymbol{x}$, this makes no formal difference. In the following gradient forms, we will assume that both the input ($\boldsymbol{x}$) and output ($\boldsymbol{y}$) variables are available for the gradient if they are needed. We will use whichever affords the most efficient result: if a variable appears in the gradient propagation rule, that variable will need to be kept available for use in the gradient.

We will now consider various common operations. An asterisk in a superscript denotes complex conjugate. Unless noted otherwise, the variables written in bold in this section may be arrays of complex numbers. For simplicity, we will usually notate them as one-dimensional vectors of length N, but this includes multidimensional arrays with a total number of elements N as well. Note that operations that mix vectors and scalars require special care, as in Eq. (63). For addition and multiplication, the gradients for these mixed forms are given in Eqs. (48) and (51) in Table 1, but in general they can be derived from a vector form and the scalar broadcast rule in Eq. (61) in Table 1.

Finally, for gradient forms involving more than one term, like addition or multiplication, if one of the terms is a constant with respect to whatever variables are of interest, the gradient rule for that term can be simply ignored.

Note on Eq. (59) in Table 1: the incremental update in this particular case bears some additional consideration. Only the elements in the subset α need to be updated; that is, the incremental update is

$$\bar{\boldsymbol{x}}[\alpha] \leftarrow \bar{\boldsymbol{x}}[\alpha] + \bar{\boldsymbol{y}} \tag{62}$$

with the other elements of $\bar{\boldsymbol{x}}$ unchanged. Note that $\bar{\boldsymbol{x}}$ may require explicit initialization in this case, even if this indexing is the first appearance of the variable; $\bar{\boldsymbol{x}}$ should be an array of the same size as $\boldsymbol{x}$, not the size of the subset indexed by α.

Table 1. Gradient Rules

Name	Forward Model	Gradient Model	Parameter Domains	#
Addition	$z = x + y$	$\bar{x} = \bar{z},\ \bar{y} = \bar{z}$	$x, y, z \in \mathbb{C}^N$	(47)
Scalar–vector addition	$z = x + y$	$\bar{x} = \sum_n \bar{z}_n,\ \bar{y} = \bar{z}$	$z, y \in \mathbb{C}^N,\ x \in \mathbb{C}^1$	(48)
Element-wise product	$z = x \circ y$	$\bar{x} = y^* \circ \bar{z},\ \bar{y} = x^* \circ \bar{z}$	$x, y, z \in \mathbb{C}^N$	(49)
Matrix product	$z = x * y$	$\bar{x} = \bar{z} * y^{T*},\ \bar{y} = x^{T*} * \bar{z}$	$x \in \mathbb{C}^{N\times P},\ y \in \mathbb{C}^{P\times M},\ z \in \mathbb{C}^{N\times M}$	(50)
Scalar–vector product	$z = xy$	$\bar{x} = \sum_n y_n^* \bar{z}_n,\ \bar{y} = x^* \bar{z}$	$z, y \in \mathbb{C}^N,\ x \in \mathbb{C}^1$	(51)
Raising to a constant power	$y = x^c$	$\bar{x} = c \circ (x^*)^{c-1} \circ \bar{y}$	$x, y \in \mathbb{C}^N,\ c \in \mathbb{R}^N$	(52)
Element-wise square magnitude	$y = \lvert x\rvert^2$	$\bar{x} = 2x \circ \Re\{\bar{y}\}$	$x \in \mathbb{C}^N,\ y \in \mathbb{R}^N$	(53)
Real part	$y = \Re\{x\}$	$\Re\{\bar{x}\} = \Re\{\bar{y}\}$	$x \in \mathbb{C}^N,\ y \in \mathbb{R}^N$	(54)
Imaginary part	$y = \Im\{x\}$	$\Im\{\bar{x}\} = \Re\{\bar{y}\}$	$x \in \mathbb{C}^N,\ y \in \mathbb{R}^N$	(55)
Complex exponential	$y = e^x$	$\bar{x} = \bar{y} \circ y^*$	$x, y \in \mathbb{C}^N$	(56)
Imaginary complex exponential	$y = e^{ix}$	$\bar{x} = \Im\{\bar{y} \circ y^*\}$	$x \in \mathbb{R}^N,\ y \in \mathbb{C}^N$	(57)
Basis function expansion	$y = \sum_n a_n \boldsymbol{B_n}$	$\bar{a}_n = \sum_{u,v} \bar{y}_{u,v} B_n(u, v)$	$y \in \mathbb{R}^{N\times M},\ \boldsymbol{B} \in \mathbb{R}^{P\times N\times M},\ \boldsymbol{a} \in \mathbb{R}^P$	(58)
Array indexing	$y = x[\![\alpha]\!]$	$\bar{x}[\![\alpha]\!] = \bar{y}$	$x, y \in \mathbb{C}^N,\ \alpha$ is an index	(59)
Array assignment	$y[\![\alpha]\!] = x$	$\bar{x} = \bar{y}[\![\alpha]\!]$	$x, y \in \mathbb{C}^N,\ \alpha$ is an index	(60)
Scalar broadcasting	$y_n = x, \quad \forall\ n$	$\bar{x} = \sum_n \bar{y}_n$	$y \in \mathbb{C}^N,\ x \in \mathbb{C}^1$	(61)

7. MIXING COMPLEX AND REAL VARIABLES

Most of the derivative rules derived in the previous section are written for complex-valued quantities. This can lead to complications when an expression combines both real and complex quantities. For example, the forward-model expression

$$z = x + iy \tag{63}$$

with real x and y is a potential problem. A direct application of the gradient rules for addition and multiplication would give

$$\bar{x} = \bar{z}, \qquad \bar{y} = -i\bar{z}, \tag{64}$$

where in general $\bar{z}$ is complex. However, this cannot be correct; since x and y are real, the imaginary parts of $\bar{x}$ and $\bar{y}$ must be zero (since the error metric may not depend on the imaginary part of a strictly real quantity). The issue here arises from the mixing of real and complex quantities in Eq. (63). In order to evaluate the expression we implicitly extended x and y into complex numbers, but our gradient rule did not take this into account. Equation (63) could be written more explicitly as

$$x' = x \to \mathbb{C}, \qquad y' = y \to \mathbb{C}, \qquad z = x' + iy', \tag{65}$$

where we use the prime notation to denote that x' and y' are temporary intermediate variables created from x and y, respectively, and we use the $\to \mathbb{C}$ notation to denote embedding a real number into the complex plane on the real axis, e.g.,

$$\Re\{x'\} = x, \qquad \Im\{x'\} = 0. \tag{66}$$

Although it is common practice to consider all numbers to be implicitly complex numbers with the imaginary part being zero for real numbers, this does not reflect how real numbers are actually stored and processed in a computer. In order to understand what is actually evaluated, it is more accurate to consider real and complex numbers as distinct, with an explicit extension operator. Clearly, the gradient derived in Eq. (64) does not account for this extension into the complex plane. We can rectify this by noting the gradient pair for the complex extension as

$$x' = x \to \mathbb{C}, \qquad \bar{x} = \Re\{\bar{x}'\}. \tag{67}$$

Including this directly in the gradient rule yields a modified version of Eq. (64):

$$\bar{x} = \Re\{\bar{z}\}, \qquad \bar{y} = -\Re\{i\bar{z}\}. \tag{68}$$

Although the implicit promotion of real variables to complex variables may appear somewhat abstract, the practical implication is quite simple: the gradients of real variables should themselves be real, which can be achieved by taking the real part before performing the gradient update.

8. LINEAR OPERATORS

We have discussed the gradient propagation of several common elementary operations, but arbitrary linear operators are of particular interest in optics and phase retrieval algorithms. Regardless of how it is numerically implemented, any linear operator can be formally written as a matrix product (assuming the input and output arrays are represented in lexicographic order):

$$y_m = \sum_n A_{mn} x_n, \qquad \boldsymbol{y} = \boldsymbol{A} * \boldsymbol{x}. \tag{69}$$

Using Eq. (50) in Table 1, we could write down the gradient rule for this case:

$$\bar{\boldsymbol{x}} = \boldsymbol{A}^{T*} * \bar{\boldsymbol{y}}. \tag{70}$$

However, since this particular case is of such centrality, we will present an explicit derivation of its gradient propagation rule here. First, we consider the real part of the gradient vector,

$$\begin{aligned}\frac{\partial E}{\partial x_{\mathrm{R},l}} &= \sum_m \frac{\partial E}{\partial y_{\mathrm{R},m}}\frac{\partial y_{\mathrm{R},m}}{\partial x_{\mathrm{R},l}} + \sum_m \frac{\partial E}{\partial y_{\mathrm{I},m}}\frac{\partial y_{\mathrm{I},m}}{\partial x_{\mathrm{R},l}} \\ &= \sum_m \bar{y}_{\mathrm{R},m}\frac{\partial}{\partial x_{\mathrm{R},l}}\Re\left\{\sum_n A_{mn}x_n\right\} + \sum_m \bar{y}_{\mathrm{I},m}\frac{\partial}{\partial x_{\mathrm{R},l}}\Im\left\{\sum_n A_{mn}x_n\right\} \\ &= \sum_m \bar{y}_{\mathrm{R},m}\frac{\partial}{\partial x_{\mathrm{R},l}}\sum_n (A_{\mathrm{R},mn}x_{\mathrm{R},n} - A_{\mathrm{I},mn}x_{\mathrm{I},n}) \\ &\quad + \sum_m \bar{y}_{\mathrm{I},m}\frac{\partial}{\partial x_{\mathrm{R},l}}\sum_n (A_{\mathrm{R},mn}x_{\mathrm{I},n} + A_{\mathrm{I},mn}x_{\mathrm{R},n}) \\ &= \sum_m \bar{y}_{\mathrm{R},m}A_{\mathrm{R},ml} + \sum_m \bar{y}_{\mathrm{I},m}A_{\mathrm{I},ml}. \end{aligned} \tag{71}$$

Next we consider the imaginary part,

$$\begin{aligned}\frac{\partial E}{\partial x_{\mathrm{I},l}} &= \sum_m \frac{\partial E}{\partial y_{\mathrm{R},m}}\frac{\partial y_{\mathrm{R},m}}{\partial x_{\mathrm{I},l}} + \sum_m \frac{\partial E}{\partial y_{\mathrm{I},m}}\frac{\partial y_{\mathrm{I},m}}{\partial x_{\mathrm{I},l}} \\ &= \sum_m \bar{y}_{\mathrm{R},m}\frac{\partial}{\partial x_{\mathrm{I},l}}\sum_n (A_{\mathrm{R},mn}x_{\mathrm{R},n} - A_{\mathrm{I},mn}x_{\mathrm{I},n}) \\ &\quad + \sum_m \bar{y}_{\mathrm{I},m}\frac{\partial}{\partial x_{\mathrm{I},l}}\sum_n (A_{\mathrm{R},mn}x_{\mathrm{I},n} + A_{\mathrm{I},mn}x_{\mathrm{R},n}) \\ &= -\sum_m \bar{y}_{\mathrm{R},m}A_{\mathrm{I},ml} + \sum_m \bar{y}_{\mathrm{I},m}A_{\mathrm{R},ml}. \end{aligned} \tag{72}$$

Next, we can assemble the whole complex gradient vector:

$$\begin{aligned}\bar{x}_l &= \frac{\partial E}{\partial x_{\mathrm{R},l}} + i\frac{\partial E}{\partial x_{\mathrm{I},l}} \\ &= \sum_m \bar{y}_{\mathrm{R},m}A_{\mathrm{R},ml} + \sum_m \bar{y}_{\mathrm{I},m}A_{\mathrm{I},ml} - i\sum_m \bar{y}_{\mathrm{R},m}A_{\mathrm{I},ml} \\ &\quad + i\sum_m \bar{y}_{\mathrm{I},m}A_{\mathrm{R},ml} \\ &= \sum_m (\bar{y}_{\mathrm{R},m} + i\bar{y}_{\mathrm{I},m})(A_{\mathrm{R},ml} - iA_{\mathrm{I},ml}) = \sum_m \bar{y}_m A^*_{ml} \\ &= \sum_m (\boldsymbol{A}^{T*})_{lm}\bar{y}_m \end{aligned} \tag{73}$$

or, in matrix form,

$$\bar{\boldsymbol{x}} = \boldsymbol{A}^{T*} * \bar{\boldsymbol{y}}, \tag{74}$$

which agrees with the expected result given in Eq. (70). For a general matrix A, the transpose conjugate is not equal to the inverse, and thus for a linear operator the inverse is not necessarily the correct operation to propagate gradients. Although we may speak casually of the gradient calculation as the "inverse" of the forward model, this is not correct mathematically: the gradient calculation does not require that the forward model be invertible. Therefore, valid analytic gradients can be obtained even for operators that are not invertible (i.e., the corresponding matrix for the operator is singular). An example is a multiplane diffraction problem in which an optical field passes through a mask in an intermediate plane. Given the field just after the mask we cannot invert the problem and find the field just before the mask, but we can find an analytic gradient for the field just before the mask given the field just after the mask. This is a difference between Gerchberg–Saxton-type iterative transform phase retrieval algorithms and nonlinear-optimization phase retrieval algorithms: the iterative-transform-type algorithms require an explicit inverse and may become unstable if the forward model is not invertible, although this may be addressed by "patching" the inverse model to stabilize it. For example, in the masking example, the masked-off portions can be saved in the forward model and added back into the inverse model to account for the "missing" energy as was shown in [3].

The discrete Fourier transform (DFT) (typically computed using FFT algorithms) is of particular interest in phase retrieval, so we will consider a DFT as an example of a linear operator. For the general case of an arbitrarily normalized DFT,

$$\boldsymbol{y} = \mathrm{DFT}\{\boldsymbol{x}\}, \qquad y_m = a\sum_{n=0}^{N-1} x_n \exp\left(\frac{-i2\pi nm}{N}\right), \tag{75}$$

we identify the matrix element as

$$A_{mn} = a\exp\left(\frac{-i2\pi nm}{N}\right), \tag{76}$$

so we can write the gradient propagation as

$$\begin{aligned}\bar{x}_n &= \sum_{m=0}^{N-1} (A^{T*})_{nm}\bar{y}_m \\ &= a\sum_{m=0}^{N-1} \exp\left(\frac{i2\pi nm}{N}\right)\bar{y}_m. \end{aligned} \tag{77}$$

If the inverse DFT is likewise arbitrarily normalized, so that

$$\mathrm{IDFT}\{x\}_m = b\sum_{n=0}^{N-1} x_n \exp\left(\frac{i2\pi nm}{N}\right), \tag{78}$$

then the gradient rule can be written as

$$\bar{\boldsymbol{x}} = \frac{a}{b}\mathrm{IDFT}\{\bar{\boldsymbol{y}}\}. \tag{79}$$

Essentially the same derivation holds if we reverse the roles of DFT and IDFT. Note that although the FFT algorithm is different from the matrix multiplication form used in this proof, these results apply to FFTs and IFFTs as well, since they ultimately compute the same numerical results as the DFT and IDFT, respectively. In the form used in the previous section, we can note the gradient rules for the DFT and IDFT are as follows:

$$\text{Forward DFT} \quad \boldsymbol{y} = \mathrm{DFT}\{\boldsymbol{x}\} \quad \bar{\boldsymbol{x}} = \frac{a}{b}\mathrm{IDFT}\{\bar{\boldsymbol{y}}\} \quad \boldsymbol{x},\boldsymbol{y} \in \mathbb{C}^N \tag{80}$$

$$\text{Inverse DFT} \quad \boldsymbol{y} = \mathrm{IDFT}\{\boldsymbol{x}\} \quad \bar{\boldsymbol{x}} = \frac{b}{a}\mathrm{DFT}\{\bar{\boldsymbol{y}}\} \quad \boldsymbol{x},\boldsymbol{y} \in \mathbb{C}^N \tag{81}$$

This is consistent with the earlier claim that in general for linear operators the transpose conjugate (rather than the inverse) is the correct model to propagate gradients. In the case of the DFT and IDFT operators, the transpose conjugate is the inverse only if they are defined such that the transforms are unitary.

9. PIXEL BINNING

Phase retrieval and image reconstruction algorithms often must simulate images as they would be sensed on a physical detector. The values of the image pixels we obtain are essentially a definite integral of the underlying continuous intensity. For images that are adequately sampled, this integration effect can be ignored, but when simulating undersampled images or when high fidelity is required, it should be taken into account. When working with discrete arrays in a computer, one can approximate the definite integral over a finite pixel area by simulating a forward-model image with an integer-factor-finer sampling than the final desired sampling and summing together blocks of adjacent pixels to approximate a single larger pixel; this is a form of downsampling, although distinct from the downsampling by decimation that is often used. We will illustrate this with an example of a 6×4 image being downsampled to a 2×2 image. We will call the original high-resolution image a and the downsampled image b, where

$$\boldsymbol{a}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03}\\ a_{10} & a_{11} & a_{12} & a_{13}\\ a_{20} & a_{21} & a_{22} & a_{23}\\ a_{30} & a_{31} & a_{32} & a_{33}\\ a_{40} & a_{41} & a_{42} & a_{43}\\ a_{50} & a_{51} & a_{52} & a_{53}\end{bmatrix}. \tag{82}$$

The lower-resolution image is given by

$$\begin{aligned}\boldsymbol{b}&=\begin{bmatrix} b_{00} & b_{10}\\ b_{01} & b_{11}\end{bmatrix}=\underset{3\times 2}{\operatorname{bin}}(\boldsymbol{a})\\ &=\begin{bmatrix} \sum_{n=0}^{2}\sum_{m=0}^{1} a_{nm} & \sum_{n=0}^{2}\sum_{m=2}^{3} a_{nm}\\ \sum_{n=3}^{5}\sum_{m=0}^{1} a_{nm} & \sum_{n=3}^{5}\sum_{m=2}^{3} a_{nm}\end{bmatrix}.\end{aligned} \tag{83}$$

If we now presume to have the array of gradients with respect to the elements of $\boldsymbol{b}$ and wish to compute the gradients with respect to the elements of $\boldsymbol{a}$, we note that each element of $\boldsymbol{a}$ contributes linearly to a single element of $\boldsymbol{b}$, so each element of $\bar{\boldsymbol{a}}$ should depend on a single element of $\bar{\boldsymbol{b}}$ as well. If we consider one term only,

$$b_{00}=a_{00}+a_{01}+a_{10}+a_{11}+a_{20}+a_{21}, \tag{84}$$

we can use the chain rule to find $\bar{a}_{00}$,

$$\bar{a}_{00}=\frac{\partial E}{\partial a_{00}}=\sum_{nm}\frac{\partial E}{\partial b_{nm}}\frac{\partial b_{nm}}{\partial a_{00}}=\frac{\partial E}{\partial b_{00}}=\bar{b}_{00}, \tag{85}$$

where the sum collapses because a_{00} appears only in the sum for b_{00}. We could also have arrived at this directly using Eq. (47) in Table 1. Essentially, we can construct $\bar{a}$ by inspection,

$$\bar{\boldsymbol{a}}=\underset{3\times 2}{\operatorname{tile}}(\bar{\boldsymbol{b}})=\begin{bmatrix} \bar{b}_{00} & \bar{b}_{00} & \bar{b}_{01} & \bar{b}_{01}\\ \bar{b}_{00} & \bar{b}_{00} & \bar{b}_{01} & \bar{b}_{01}\\ \bar{b}_{00} & \bar{b}_{00} & \bar{b}_{01} & \bar{b}_{01}\\ \bar{b}_{10} & \bar{b}_{10} & \bar{b}_{11} & \bar{b}_{11}\\ \bar{b}_{10} & \bar{b}_{10} & \bar{b}_{11} & \bar{b}_{11}\\ \bar{b}_{10} & \bar{b}_{10} & \bar{b}_{11} & \bar{b}_{11}\end{bmatrix}. \tag{86}$$

We can define the bin and tile operators in a more general case as

$$\left(\underset{p\times q}{\operatorname{bin}}(\boldsymbol{a})\right)_{rs}=\sum_{n=rp}^{(r+1)p}\sum_{m=sq}^{(s+1)q} a_{nm}, \tag{87}$$

where blocks of size p by q from $\boldsymbol{a}$ are binned together to form the final image and

$$\left(\underset{p\times q}{\operatorname{tile}}(\boldsymbol{b})\right)_{nm}=b_{\lfloor n/p\rfloor,\lfloor m/q\rfloor}, \tag{88}$$

where the bottom side bracket denotes rounding down. Both these definitions assume zero-based indexing for $\boldsymbol{a}$ and $\boldsymbol{b}$. The gradient rules for the tile and bin are as follows:

$$\text{Pixel binning}\quad \boldsymbol{y}=\underset{p\times q}{\operatorname{bin}}(\boldsymbol{x})\quad \bar{\boldsymbol{x}}=\underset{p\times q}{\operatorname{tile}}(\bar{\boldsymbol{y}})\quad \boldsymbol{x}\in\mathbb{R}^{N\times M}\boldsymbol{y}\in\mathbb{R}^{pN\times qM} \tag{89}$$

$$\text{Pixel tiling}\quad \boldsymbol{y}=\underset{p\times q}{\operatorname{tile}}(\boldsymbol{x})\quad \bar{\boldsymbol{x}}=\underset{p\times q}{\operatorname{bin}}(\bar{\boldsymbol{y}})\quad \boldsymbol{x}\in\mathbb{R}^{pN\times qM}\boldsymbol{y}\in\mathbb{R}^{N\times M} \tag{90}$$

where the first rule follows from the same argument advanced in Eq. (85). The second rule follows from noting that, for the tiling operation, a p by q block of output pixels depends on a single input pixel, so all of their gradient contributions need to be added together. Note that although the bin and tile play symmetric roles in their respective gradient formulations, they are not inverses of one another in general (indeed bin has no inverse). While for all $\boldsymbol{x}$,

$$\underset{p\times q}{\operatorname{bin}}\left[\underset{p\times q}{\operatorname{tile}}(x)\right]=pqx, \tag{91}$$

in general,

$$\underset{p\times q}{\operatorname{tile}}\left[\underset{p\times q}{\operatorname{bin}}(\boldsymbol{x})\right]\neq pq\boldsymbol{x}. \tag{92}$$

The bin and tile are linear operators, though bin is not invertible; this illustrates the claim that the steps in a nonlinear-optimization forward model need not be invertible to have tractable reverse-mode gradient models.

10. SIMPLE PHASE RETRIEVAL ALGORITHM REVISITED

Now that we have developed the necessary derivative relationships, we revisit the analytic gradient derivation of Eq. (9). We will go through the forward-model steps in reverse, in the order in which they would be executed in the reverse-mode strategy. The error metric in Eq. (7) yields

$$\bar{\boldsymbol{I}}=2\boldsymbol{w}\circ(\boldsymbol{I}-\boldsymbol{D}) \tag{93}$$

through the application of Eqs. (47), (49), and (52) in Table 1. The squared magnitude in Eq. (6) yields

$$\overline{\boldsymbol{G}}=2\bar{\boldsymbol{I}}\circ\boldsymbol{G} \tag{94}$$

from Eq. (53) in Table 1. We have dropped the real part since $\bar{\boldsymbol{I}}$ is always real. The FFT in Eq. (5) gives us

$$\bar{\boldsymbol{g}} = \mathrm{IFFT}(\overline{\boldsymbol{G}}) \tag{95}$$

by Eq. (80). The complex pupil formation in Eq. (4) gives us

$$\overline{\boldsymbol{W}} = \frac{2\pi}{\lambda}\Im[\bar{\boldsymbol{g}} \circ \boldsymbol{g}^*] \tag{96}$$

according to Eqs. (49), (51), and (57) in Table 1. The basis expansion in Eq. (3) yields

$$\bar{a}_n = \sum_p \overline{W}_p Z_{n,p} \tag{97}$$

following Eq. (58) in Table 1, if we take p as indexing both dimension of the arrays. We can obtain Eq. (9) by simply back-substituting Eqs. (9394)–(95) into Eq. (96), but for the practical purposes of implementing an phase retrieval algorithm, we need not do so. Instead we may choose to implement the individual steps as pairs of subroutines or classes in an object-oriented design so that they can be reused in other projects.

11. EXAMPLE: NONLINEAR DETECTOR RESPONSE

To illustrate the utility of this approach, we will consider phase retrieval in a system with unknown second-order detector nonlinearity and Gaussian noise. The model for the detected intensity $\boldsymbol{D}$ is

$$\boldsymbol{D} = a\boldsymbol{I} \circ \boldsymbol{I} + b\boldsymbol{I} + c + \boldsymbol{n}, \tag{98}$$

where the constants a, b, and c are not known exactly and must be estimated, and $\boldsymbol{n}$ is noise. We will use a weighted sum of squared differences of intensity error metric, in which $\boldsymbol{M}$ is the model for the data, and $\boldsymbol{w}$ is the weighting function. The phase retrieval forward model is then

$$\begin{aligned}
\boldsymbol{W} &= \sum_n a_n \boldsymbol{Z}_n, \\
\boldsymbol{\theta} &= \frac{2\pi}{\lambda}\boldsymbol{W}, \\
\boldsymbol{g} &= \boldsymbol{A} \circ \exp(i\boldsymbol{\theta}), \\
\boldsymbol{G} &= \mathrm{FFT}\{\boldsymbol{g}\}, \\
\boldsymbol{I} &= |\boldsymbol{G}|^2 = \boldsymbol{G} \circ \boldsymbol{G}^*, \\
\boldsymbol{M} &= a\boldsymbol{I} \circ \boldsymbol{I} + b\boldsymbol{I} + c, \\
E &= \sum_n [\boldsymbol{w} \circ (\boldsymbol{M} - \boldsymbol{D})^2]_n,
\end{aligned} \tag{99}$$

where $\boldsymbol{Z}$ is a three-dimensional array containing Zernike basis functions, $\boldsymbol{A}$ is a known pupil amplitude, and λ is the system wavelength. For the intermediate quantities, $\boldsymbol{\theta}$ is the pupil phase, $\boldsymbol{g}$ is the complex field in the pupil plane, $\boldsymbol{G}$ is the complex field in the image plane, and $\boldsymbol{I}$ is the image intensity. Notice that with the exception of the nonlinear intensity response in the second to last line, this forward model is essentially identical to that of Eqs. (3)–(7). Recall from Eq. (2) that the square in the error metric is interpreted as the scalar square applied to the array quantity in parentheses. We apply the derivative propagation rules step by step to compute the gradient with respect to the a_n coefficients:

$$\begin{aligned}
\overline{\boldsymbol{M}} &= 2\boldsymbol{w} \circ (\boldsymbol{M} - \boldsymbol{D}), \\
\bar{\boldsymbol{I}} &= 2a\boldsymbol{I} \circ \overline{\boldsymbol{M}} + b\overline{\boldsymbol{M}} = \overline{\boldsymbol{M}} \circ (2a\boldsymbol{I} + b), \\
\overline{\boldsymbol{G}} &= 2\boldsymbol{G} \circ \bar{\boldsymbol{I}}, \\
\bar{\boldsymbol{g}} &= \mathrm{IFFT}\{\overline{\boldsymbol{G}}\}, \\
\bar{\boldsymbol{\theta}} &= \Im\{\boldsymbol{g}^* \bar{\boldsymbol{g}}\}, \\
\overline{\boldsymbol{W}} &= \frac{2\pi}{\lambda}\bar{\boldsymbol{\theta}}, \\
\bar{a}_n &= \sum_m (\overline{\boldsymbol{W}} \circ \boldsymbol{Z}_n)_m,
\end{aligned} \tag{100}$$

where we can make reference to Eqs. (93)–(97) for everything except the nonlinear response to intensity. The gradients with respect to a, b, and c are

$$\bar{a} = \sum_m \overline{M}_m I_m^2, \qquad \bar{b} = \sum_m \overline{M}_m I_m, \qquad \bar{c} = \sum_m \overline{M}_m. \tag{101}$$

The sums in the equations above arise from the implicit broadcasting of the scalars a, b, and c to sizes compatible with $\boldsymbol{I}$ using the rule from Eq. (61) in Table 1, not from the final sum that forms the error metric.

Observe that we were not required to write an explicit function for the error metric as a function of $\boldsymbol{W}$, a, b, and c in order to obtain the gradients; specifying the steps to compute the forward model was sufficient. Also notice that, in the gradient derivation, only the second line in Eq. (100) relates to the nonlinearity; the rest is identical to a phase retrieval algorithm for a linear detector. Finally, the addition of the gradients with respect to a, b, and c has no impact on the form of the gradients for $\boldsymbol{W}$. This highlights the modularity enabled by this approach; code relating to one part of a model can be kept largely separate from code relating to other parts, and a change in one part of the forward model requires a change in only the corresponding part of the gradient.

12. CONCLUSIONS

We have shown how the standard technique of reverse-mode algorithmic differentiation can be applied to computer programs making use of multidimensional arrays of complex numbers. We have provided the complex forms of the gradient propagation rules for various elementary operations in a format suitable for application via "manual" algorithmic differentiation, i.e., without requiring the use of an automated software tool to produce the gradient calculations. The derivation of analytic gradients for phase retrieval obtained in the traditional way (by hand, using symbolic differentiation) requires a substantial investment of time for the researcher and limits the ease with which novel forward-model formulations can be explored; the use of the techniques discussed here can greatly reduce that amount of time. Additionally, the structure of the reverse-mode algorithmic differentiation process allows us to separate code and variables into related data structures and subroutines relating to different steps in a forward model. This allows modularity and reuse of code in related families of error metrics.

APPENDIX A: REVIEW OF CURRENT SOFTWARE TOOLS

Although there are many software packages that can perform some aspects of the reverse-mode algorithmic differentiation discussed in this paper, we found at the time of this writing none that were fully suitable to our needs—hence the "manual" approach taken in this paper. For research purposes we are primarily interested in high-level interpreted languages such as MATLAB and Python that allow rapid development and interactive exploration. This rules out the use of tools developed for Fortran or C/C++, including Tapenade [8], ADOL-C [9], and CppAD [10]. We prefer free and open source tools both so that their working can be understood and to avoid encumbering research software with additional software licensing requirements and costs. Additionally, automatic/algorithmic differentiation tools support only "forward-mode" differentiation, a process similar to that described in this paper, which is beneficial for algorithms in which a small number of inputs yields a large number of outputs, but not optimization problems with a single scalar error metric.

For MATLAB, although there are several tools listed in [11], we found none that were both free and supported reverse-mode calculations. For Python there are more options, including [12–14]. All three of these libraries work on an operator overloading approach (see [4] for a discussion of operator overloading), which makes them incompatible with external numerical libraries that are not written to be aware of their own special data types and limits their application for large array oriented calculations. Theano [15], while not specifically an algorithmic differentiation tool, does support array oriented calculations and is able to produce gradient functions programmatically; however, it requires adopting a very different style from that typically used in Python numerical programs and still requires the user to supply gradient rules like the ones discussed in this paper for any library functions outside of its own core features. The AlgoPy project [16] appears to be addressing some of these limitations, but is still in very early development as of this writing.

One might also considering using a traditional symbolic mathematics tool such as SymPy [17] to compute gradient expressions. While these tools are certainly able to compute the symbolic derivatives of a given expression, they are mainly intended to work with a small number of named continuous scalar variables rather than a discrete calculation where the variables are entries in large arrays. As such they lack the ability to compactly express the phase retrieval and image reconstruction calculations we are interested in. Furthermore, in many cases the reverse-mode algorithmic differentiation gradient calculation benefits greatly from reusing intermediate values computed in the forward model, which inherently requires the two to be computed together, whereas a symbolic computing tool typically will try to produce an expression for the gradient in isolation.

In summary, though there are many software packages that touch on parts of the gradient calculation problem, it is the view of the authors that none of the present packages meet the needs of phase retrieval and image reconstruction algorithms without requiring substantial additional work. Therefore we put forth the manual algorithmic differentiation approach discussed in this paper.

ACKNOWLEDGMENTS

Thanks to Matthew Bergkoetter for finding several errors in early drafts of Table 1 and for pointing out the tricky nature of gradients for calculations that mix real and complex variables. This research was supported by NASA Goddard Space Flight Center.

REFERENCES

1. S. J. Wright and J. Nocedal, "Quasi-Newton methods," in *Numerical Optimization* (Springer, 1999), Sect. 8.1.
2. J. R. Fienup, "Phase retrieval algorithms: a comparison," Appl. Opt. **21**, 2758–2769 (1982).
3. J. R. Fienup, "Phase-retrieval algorithms for a complicated optical system," Appl. Opt. **32**, 1737–1746 (1993).
4. A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. (SIAM, 2008).
5. J. R. Fienup, "Phase retrieval for undersampled broadband images," J. Opt. Soc. Am. A **16**, 1831–1837 (1999).
6. R. G. Paxman, T. J. Schulz, and J. R. Fienup, "Joint estimation of object and aberrations by using phase diversity," J. Opt. Soc. Am. A **9**, 1072–1085 (1992).
7. K. F. Riley, M. P. Hobson, and S. J. Bence, *Mathematical Methods for Physics and Engineering* (Cambridge University, 2006).
8. L. Hascoet and V. Pascual, "The tapenade automatic differentiation tool: principles, model, and specification," ACM Trans. Math. Softw. **39**, 1–43 (2013).
9. A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++," ACM Trans. Math. Softw. **22**, 131–167 (1996).
10. B. Bell, "CppAD: a package for C++ algorithmic differentiation," in Computational Infrastructure for Operations Research (2012), http://www.coin-or.org/CppAD/.
11. "Selected AD tools," http://www.autodiff.org.
12. A. Lee, "ad 1.2.2," https://pypi.python.org/pypi/ad.
13. N. C. Domingo, "adol-Py," https://pypi.python.org/pypi/adol-Py/0.1.
14. B. M. Bell, "pycppad," http://www.seanet.com/~bradbell/pycppad/index.xml.
15. J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)* (2010).
16. S. F. Walter and L. Lehmann, "Algorithmic differentiation in Python with AlgoPy," J. Comput. Sci. **4**, 334–344 (2013).
17. SymPy Development Team, "SymPy: Python library for symbolic mathematics" (2014), http://www.sympy.org.