THE F-1 ALGORITHM FOR EFFICIENT COMPUTATION OF THE HESSIAN MATRIX OF AN OBJECTIVE FUNCTION DEFINED IMPLICITLY BY THE SOLUTION OF A STEADY-STATE PROBLEM*

BENOÎT PASQUIER† AND FRANÇOIS PRIMEAU†

Abstract. Steady-state systems of nonlinear partial differential equations (PDEs) are common in engineering and the biogeosciences. These systems are typically controlled by parameters that can be estimated efficiently using second-order optimization algorithms. However, computing the gradient vector and Hessian matrix of a given objective function defined implicitly by the solution of large PDE systems is seldom economical. Here we present a fast and easy-to-use algorithm for computing the gradient and Hessian of an objective function implicitly constrained by a steady-state PDE system. We call the new algorithm, which is based on the use of hyperdual numbers, the F-1 algorithm, because it requires only one factorization of the constraint-equation Jacobian. Careful examination of the relationships that arise from differentiating the PDE system reveal analytical shortcuts that the F-1 algorithm leverages. We benchmark the F-1 algorithm against five numerical differentiation schemes in the context of optimizing a global steady-state model of the marine phosphorus cycle that depends explicitly on $m = 6$ parameters. In this context, the F-1 algorithm computes the Hessian 16 to 100 times faster than other algorithms, allowing for the entire optimization procedure to be performed 4 to 26 times faster. This is because other algorithms require $\mathcal{O}(m)$ to $\mathcal{O}(m^2)$ factorizations, which suggests even greater speedups for larger problems. To facilitate reproducibility and future benchmarks, all the code developed for this study was implemented as open-source Julia packages.

Key word. Global, Ocean, Marine biogeochemistry, Biogeochemistry, Model, Modeling, Inverse, Inverse model, Optimization, Uncertainty, Sensitivity, Differential equations, Partial differential equations, PDE, steady-state, Newton's method, Factorization, Automatic differentiation, Algorithmic differentiation, Autodiff, AD, Finite differences, Complex-step, Dual numbers, Hyperdual numbers, Nutrients, Nutrient cycles, Biogeochemical cycles, gradient, Hessian, Jacobian, Julia, Open source

AMS subject classifications. 15Axx, 15A06, 15A09, 15A23, 15A24, 15A29, 15A66, 15A69, 15A99, 26B10, 26A24, 13P99, 90C53, 90C90

1. Introduction.

The geosciences are rich with problems involving spatial data that can be modeled using partial differential equations (PDEs). In cases where steady-state or time-mean fields are of specific interest, such problems can be expressed generically as

$$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) = 0, \tag{1.1}$$

where $\boldsymbol{x}$ is the model state vector comprising one or more discretized field variables and $\boldsymbol{p}$ is a vector of adjustable parameters (see, e.g., [22, 50, 11, 24, 25, 44]).

A major modeling goal is then to find the value of $\boldsymbol{x}$ and $\boldsymbol{p}$ that are in the best possible agreement with available observational data while satisfying (1.1). Mathematically, this translates into the generic constrained optimization problem

$$\begin{cases} \underset{\boldsymbol{x},\boldsymbol{p}}{\text{minimize}} & f(\boldsymbol{x}, \boldsymbol{p}) \\ \text{subject to} & \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) = 0, \end{cases} \tag{1.2}$$

where $f(\boldsymbol{x}, \boldsymbol{p})$ is some measure of how far the state and parameter vectors are from the data and/or some assumed prior values. Here, we restrict ourselves to the case

*Submitted to the the SIAM Journal of Scientific Computing for review on 2019-07-09.

Funding: This work was funded by the US Department of Energy grant DE-SC0016539 and the National Science Foundation grant 1658380.

†Department of Earth System Science, University of California, Irvine, CA, United States (pasquieb@uci.edu, fprimeau@uci.edu).

where the solution to (1.1) defines $\boldsymbol{x}$ as an implicit function of $\boldsymbol{p}$, which we denote by $\boldsymbol{s}(\boldsymbol{p})$, the steady-state solution. The problem defined by (1.2) is then equivalent to finding the minimum of the objective function defined by

$$\hat{f}(\boldsymbol{p}) \equiv f\big(\boldsymbol{s}(\boldsymbol{p}), \boldsymbol{p}\big). \tag{1.3}$$

In a Bayesian formulation of the parameter estimation problem, $\hat{f}$ would correspond to the negative logarithm of the posterior probability distribution. Solving $\hat{\boldsymbol{p}} = \arg\min_{\boldsymbol{p}} \hat{f}(\boldsymbol{p})$ is then equivalent to finding the most probable parameter values. Efficient algorithms for minimizing $\hat{f}(\boldsymbol{p})$ in multidimensional parameter spaces make use of the gradient, $\nabla\hat{f}(\boldsymbol{p})$, and Hessian, $\nabla^2\hat{f}(\boldsymbol{p})$, to select the most promising search directions. Furthermore, in parameter estimation problems, the Hessian matrix, $\nabla^2\hat{f}(\boldsymbol{p})$, is of direct interest because its inverse evaluated at $\hat{\boldsymbol{p}}$ can be used to construct a useful approximation to the error covariance matrix for the parameters, which provides a useful summary of the parameter uncertainties (e.g., [51, 52, 55]).

The present study focuses on PDE problems with discretization schemes that lead to a sparse Jacobian matrix, $\nabla_{\boldsymbol{x}}\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$, that can be factored and stored in computer memory. For such problems Newton's method can be used to efficiently solve (1.1). Here, we show how to take advantage of this fact together with the application of recently developed hyperdual numbers (e.g., [12, 14, 13]) to simplify and greatly reduce the computational cost of evaluating $\nabla\hat{f}(\boldsymbol{p})$ and $\nabla^2\hat{f}(\boldsymbol{p})$.

The typical procedure for minimizing the objective function defined by (1.3) involves two nested iterative processes, as illustrated in Figure 1. The inner solver finds the steady-state solution, $\boldsymbol{s}(\boldsymbol{p})$, by iteratively updating the state, $\boldsymbol{x}$, until the norm of the state function, $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$, is sufficiently small. This is indicated by the "$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) \approx 0$?" condition, which determines when the inner-solver loop terminates. On the outside, the optimizer iteratively searches for a minimum of $\hat{f}$. The optimizer loop updates the parameters, $\boldsymbol{p}$, and terminates when the norm of the gradient of the objective function, $\nabla\hat{f}(\boldsymbol{p})$, is sufficiently small, which is indicated by the "$\nabla\hat{f}(\boldsymbol{p}) \approx 0$?" condition. The outer optimizer problem, like the inner solver problem, can be solved using Newton's method provided the search direction,

$$\Delta\boldsymbol{p} \equiv -\Big[\nabla^2\hat{f}(\boldsymbol{p})\Big]^{-1}\nabla\hat{f}(\boldsymbol{p}), \tag{1.4}$$

can be computed.

However, computing the derivatives required to evaluate $\nabla\hat{f}(\boldsymbol{p})$ and $\nabla^2\hat{f}(\boldsymbol{p})$ analytically is laborious, prone to errors, and potentially computationally expensive — see (2.5) for example, which involves five large third-order tensors. The evaluation of the gradient vector and Hessian matrix is therefore typically done using finite differences applied directly to $\hat{f}$. However, finite-difference approximations for computing the Hessian matrix is also computationally expensive when m is moderately large and suffers from both round-off and truncation errors [29], which can have detrimental effects on the convergence rate of the optimizer.

A recently developed alternative to finite differences, which does not suffer from round-off or truncation errors is the application of dual numbers to efficiently compute numerical derivatives (see, e.g., [37]). Dual numbers, like complex numbers, extend the real numbers by introducing a new unit, denoted ε, but with $\varepsilon^2 \equiv 0$ rather than $i^2 = -1$ as is the case for the imaginary unit. A more detailed description of dual numbers is given in subsection SM4.2 and references therein. Implementations of dual

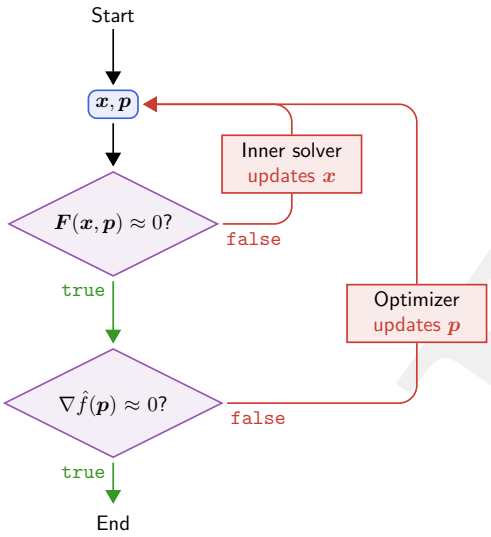

FIG. 1. *Schematic diagram representing the optimization procedure. Starting at the top with an initial choice for the state, $\boldsymbol{x}$, and parameters, $\boldsymbol{p}$, the procedure goes through two nested iterative loops. The outer loop is the optimizer, which iterates on $\boldsymbol{p}$ until it lies at the minimum of the objective function, $\hat{f}$ (i.e., until $\nabla \hat{f}(\boldsymbol{p}) \approx 0$). Nested inside the optimizer is the inner solver, which is executed for each update of $\boldsymbol{p}$. The inner-solver loop updates $\boldsymbol{x}$ until it approximately satisfies the steady-state condition, $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) \approx 0$, with is equivalent to $\boldsymbol{x} = \boldsymbol{s}(\boldsymbol{p})$. We note that the conditional statements to terminate the loops are approximations because of finite precision.*

numbers for efficiently and accurately computing derivatives are available for several scientific computing languages (see, e.g., [12, 14, 13, 56, 48, 37]).

To see how to compute the gradient using dual numbers let $\boldsymbol{e}_j$ be the jth vector of the natural basis of $\mathbb{R}^m$, i.e., a vector of m zeros except for a 1 in the jth entry. ($j = 1, \ldots, m$ indexes the m dimensions of the parameter space.) Then, the Taylor expansion of the objective function at $\boldsymbol{p}$ in the $\varepsilon \boldsymbol{e}_j$ direction is given by

$$\hat{f}(\boldsymbol{p} + \varepsilon \boldsymbol{e}_j) = \hat{f}(\boldsymbol{p}) + \varepsilon \nabla \hat{f}(\boldsymbol{p})\, \boldsymbol{e}_j, \tag{1.5}$$

where we express the gradient, $\nabla \hat{f}(\boldsymbol{p})$, as a row vector, so that the product $\nabla \hat{f}(\boldsymbol{p})\, \boldsymbol{e}_j$ yields its scalar jth entry. Thus, each entry of the gradient can be computed by evaluating the objective function with the dual-valued parameters $\boldsymbol{p} + \varepsilon \boldsymbol{e}_j$ and taking the dual part of the result. Rearranging each entry into a row vector gives a formula to compute the gradient in m dual-valued evaluations of the objective function,

$$\nabla \hat{f}(\boldsymbol{p}) = \mathfrak{D} \left(\begin{bmatrix} \hat{f}(\boldsymbol{p} + \varepsilon \boldsymbol{e}_1) \\ \hat{f}(\boldsymbol{p} + \varepsilon \boldsymbol{e}_2) \\ \vdots \\ \hat{f}(\boldsymbol{p} + \varepsilon \boldsymbol{e}_m) \end{bmatrix}^{\mathsf{T}} \right). \tag{1.6}$$

Note that the dual-step algorithm cannot be naively applied recursively to compute second-order derivatives because $\varepsilon^2 = 0$ ensures that terms of order two (and above) vanish in the Taylor expansion. To compute second-order derivatives, Fike and Alonso [12] have developed hyperdual numbers. Two distinct hyperdual units,

ε_1 and ε_2, are introduced, such that $\varepsilon_1^2 = \varepsilon_2^2 = 0$ but such that $\varepsilon_1\varepsilon_2 \neq 0$. Just like the dual unit, the hyperdual units play the role of infinitesimally small numbers. However, because they are independent and do not cancel each other out, they can propagate infinitesimal perturbations in two directions simultaneously. For more details on hyperdual numbers, see subsection SM4.3 and references therein.

By definition, hyperdual-valued Taylor expansions only extend to second order terms. In particular, for any pair $(\boldsymbol{e}_j, \boldsymbol{e}_k)$ of directions in parameter space (with j and k spanning the dimensions 1 to m of the parameter space), we have that

$$\hat{f}(\boldsymbol{p} + \varepsilon_1\boldsymbol{e}_j + \varepsilon_2\boldsymbol{e}_k) = \hat{f}(\boldsymbol{p}) + \varepsilon_1\nabla\hat{f}(\boldsymbol{p})\boldsymbol{e}_j + \varepsilon_2\nabla\hat{f}(\boldsymbol{p})\boldsymbol{e}_k + \varepsilon_1\varepsilon_2\boldsymbol{e}_j^\mathsf{T}\nabla^2\hat{f}(\boldsymbol{p})\boldsymbol{e}_k, \tag{1.7}$$

where the product $\boldsymbol{e}_j^\mathsf{T}\,\nabla^2\hat{f}(\boldsymbol{p})\,\boldsymbol{e}_k$ yields the entry in the jth row and the kth column of the Hessian matrix and where the product $\nabla\hat{f}(\boldsymbol{p})\,\boldsymbol{e}_j$ yields the jth entry of the gradient. Thus, one can compute the Hessian matrix with $m(m+1)/2$ hyperdual-valued evaluations of the objective function. Specifically, denoting the hyperdual parameters by $\boldsymbol{p}_{jk} \equiv \boldsymbol{p} + \varepsilon_1\boldsymbol{e}_j + \varepsilon_2\boldsymbol{e}_k$, the Hessian is given by

$$\nabla^2\hat{f}(\boldsymbol{p}) = \mathfrak{H}\left(\begin{bmatrix} \hat{f}(\boldsymbol{p}_{11}) & \hat{f}(\boldsymbol{p}_{12}) & \cdots & \hat{f}(\boldsymbol{p}_{1m}) \\ \hat{f}(\boldsymbol{p}_{12}) & \hat{f}(\boldsymbol{p}_{22}) & \cdots & \hat{f}(\boldsymbol{p}_{2m}) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{f}(\boldsymbol{p}_{1m}) & \hat{f}(\boldsymbol{p}_{2m}) & \cdots & \hat{f}(\boldsymbol{p}_{mm}) \end{bmatrix}\right), \tag{1.8}$$

where $\mathfrak{H}(x)$ is the $\varepsilon_1\varepsilon_2$ coefficient of x.

Although they provide an attractive alternative to the fully analytical approach, the numerical algorithms listed above come at a price. Indeed, in practice, these numerical methods suffer large computational costs on top of potential implementation pitfalls. Computing the gradient, $\nabla\hat{f}(\boldsymbol{p})$, via (1.6) seems straightforward and computationally efficient at face value but we note that each evaluation of $\hat{f}$ will generate a call to the inner solver. That is, each call will need to find the dual-valued steady-state solution, $\boldsymbol{s}(\boldsymbol{p}+\varepsilon\boldsymbol{e}_j)$, thus forcing the inner solver, which uses Newton's method, to perform at least one computationally-expensive factorization of each dual-valued matrix $\nabla_{\boldsymbol{x}}\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}+\varepsilon\boldsymbol{e}_j)$ and potentially multiple such factorizations. Similarly, computing the Hessian, $\nabla^2\hat{f}(\boldsymbol{p})$, via (1.8) will generate at a minimum $m(m+1)/2$ calls to the inner solver to find the hyperdual-valued steady-state solution, $\boldsymbol{s}(\boldsymbol{p}_{jk})$, thus forcing the inner solver to perform an even larger number of expensive factorizations of hyperdual-valued $\nabla_{\boldsymbol{x}}\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}_{jk})$ matrices. The additional calls to the inner solver also expose the user to potential implementation pitfalls if for some reason the solver does not handle non-real numbers properly. This would be the case for example if the inner solver invoked operations with non-real numbers internally in the first place, or did not check for convergence of non-real parts. (These pitfalls are discussed in section 6.)

Optimization problems defined generically in the form of (1.2) are common in physical sciences and engineering and practical solutions have been investigated and documented. For example, for aerospace engineering, Rumpfkeil and Mavriplis [49] suggested an efficient solution to a similar optimization problem to improve airfoil aerodynamism. They showed that taking the adjoint of the derivatives of their steady-state problem combined with algorithmic differentiation could lead to an optimally-efficient algorithm for computing the Hessian matrix. Here, in a similar approach, we show that a careful refactoring of the algorithm for computing $\nabla\hat{f}$ and $\nabla^2\hat{f}$ using

an adjoint formulation and hyperdual numbers can avoid all the calls to the inner solver. This leads to an algorithm, which we call F-1, for computing the gradient and Hessian that is simultaneously easy-to-use, fast, and accurate. The name, F-1, of the new algorithm relates to the fact that it is fast and to the fact that it requires only 1 factorization of the large Jacobian matrix for the PDE constraint equation

By using dual and hyperdual numbers, the accuracies of the gradient and Hessian computed by the F-1 algorithm are close to machine precision. Furthermore, the F-1 algorithm requires only a single factorization of the real-valued matrix $\mathbf{A} \equiv \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{s}, \boldsymbol{p})$ followed by $m + 1$ forward and back substitutions to compute both the gradient and Hessian — the minimum possible. (We use $\boldsymbol{s}$ instead of $\boldsymbol{s}(\boldsymbol{p})$ for brevity throughout.) Additionally, it does not require any call to the inner solver, avoiding the pitfalls of autodifferentiating through an iterative solver. Finally, the F-1 algorithm requires no analytical derivatives with respect to $\boldsymbol{p}$, making it simple to use.

We emphasize that an important requirement for the F-1 algorithm to be applicable is that it must be possible to create, store, and factorize the Jacobian matrix, $\mathbf{A}$. In other words, a generalization of the F-1 algorithm to problems for which the state is too large for the Jacobian to be factored or problems with millions of parameters is beyond the scope of this study. To illustrate the efficiency of the F-1 algorithm, we apply it to the optimization of a global marine phosphorus-cycling model. (In this model, described in detail in subsection 3.2, the Jacobian, $\mathbf{A}$, is a sparse $400\,320 \times 400\,320$ matrix with $3\,800\,846$ non-zero entries, and $\boldsymbol{p}$ is of length $m = 6$.)

Global nutrient-cycling models play a key role in our understanding of the Earth system. Photosynthetic microbes living in the sunlit upper ocean continuously remove dissolved carbon dioxide (CO_2) and nutrients from surface waters to produce organic matter that gets exported to depth in the form of sinking particles. The downward flux of these particles supplies the carbon and energy that sustain life in the dark subsurface waters. The respiration at depth of the exported organic matter particles also maintains a vertical gradient of CO_2 in the ocean against the tendency of mixing and overturning circulation to homogenize the concentration of dissolved constituents. As such, this "biological carbon pump" [54, 46, 1] sets the partitioning of CO_2 between the atmosphere and ocean with important consequences for the climate of the Earth. The strength of the biological pump is strongly regulated by the availability of nutrients such as phosphate (PO_4).

The biogeochemical mechanisms that control the cycling of nutrients, which involve hundreds of thousants of species, are complex. Additionally, the ocean is hard to sample with sufficient spatial and temporal coverage, hindering our capacity to understand and quantify marine processes. Oceanographers therefore often rely on mathematical models with biogeochemical parameterizations that are calibrated by requiring the model to reproduce available observations. Estimating these parameters through objective optimization is a task whose importance is increasingly recognized in marine biogeochemistry (see, e.g., [24, 25, 15, 8, 10, 44]) but whose implementation remains a formidable challenge for state-of-the-art Earth System Models.

We demonstrate the performance of the F-1 algorithm in the context of optimizing $m = 6$ parameters of a global marine biogeochemistry model of the phosphorus cycle, for which the state has size $n = 400\,320$. We benchmark the F-1 algorithm against other numerical differentiation algorithms by recording computation times in the same conditions (i.e., for the same state function, $\boldsymbol{F}$, and objective function, $\hat{f}$, with the same parameters, $\boldsymbol{p}$, on the same computer, and so forth). We show that the F-1 algorithm affords significant speedups. In fact, we show that the F-1 algorithm can compute gradient and Hessian at virtually no added cost relative to a fully analytical

approach, as has been suggested in [49].

In the phosphorus-cycling-model optimization context, the F-1 algorithm computes the Hessian matrix from 16 to 100 times faster than other algorithms, affording 4- to 26-fold computational speedups overall. Based on simple time-complexity arguments, we expect the computational cost benefits of the F-1 algorithm to scale with the size of the problem, because current state-of-the-art numerical differentiation algorithms require $\mathcal{O}(m)$ to $\mathcal{O}(m^2)$ factorizations, compared to a single factorization for the F-1 algorithm. In fact, for fixed n, we expect that computing the Hessian using the F-1 algorithm would be 3 orders of magnitude faster than finite-differences for $m \sim 20$ parameters, and 5 orders of magnitude for $m \sim 200$ parameters.

We start by describing the F-1 algorithm and its derivation in section 2, where we also give a short description of six other numerical differentiation algorithms. We describe our implementation in section 3 and show the results of the optimization of the phosphorus-cycling model and of the benchmarks of the F-1 algorithm in section 4. We conclude in section 5 and further discuss in section 6.

2. Theory.

2.1. Analytical formulas. The gradient, $\nabla \hat{f}(\boldsymbol{p})$, which we express as a row vector, is obtained by differentiating (1.3) via the chain rule:

$$\underbrace{\nabla \hat{f}(\boldsymbol{p})}_{1\times m} = \underbrace{\nabla_{\boldsymbol{x}} f(\boldsymbol{s},\boldsymbol{p})}_{1\times n}\,\underbrace{\nabla \boldsymbol{s}(\boldsymbol{p})}_{n\times m} + \underbrace{\nabla_{\boldsymbol{p}} f(\boldsymbol{s},\boldsymbol{p})}_{1\times m}, \tag{2.1}$$

where $\nabla_{\boldsymbol{x}} f$ and $\nabla_{\boldsymbol{p}} f$ are the partial derivatives of $f(\boldsymbol{x},\boldsymbol{p})$ with respect to $\boldsymbol{x}$ and $\boldsymbol{p}$, respectively, and $\nabla \boldsymbol{s}(\boldsymbol{p})$ is the derivative of $\boldsymbol{s}$ with respect to $\boldsymbol{p}$. The matrix size of each derivative, which defines the row/column orientation of vectors and matrices, is indicated below each term. Here, strictly speaking, the row vector $\nabla \hat{f}(\boldsymbol{p})$ is the transpose of the gradient of $\hat{f}(\boldsymbol{p})$, which we usually take to be an $m \times 1$ column vector. However, we refer to $\nabla \hat{f}(\boldsymbol{p})$ as the gradient (of the objective function) throughout for simplicity. The $\nabla \boldsymbol{s}(\boldsymbol{p})$ term in (2.1) is obtained by differentiating the steady-state equation, (1.1), and gives

$$\underbrace{\mathbf{A}}_{n\times n}\,\underbrace{\nabla \boldsymbol{s}(\boldsymbol{p})}_{n\times m} + \underbrace{\nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})}_{n\times m} = 0, \tag{2.2}$$

where $\nabla_{\boldsymbol{p}} \boldsymbol{F}$ is the partial derivative of $\boldsymbol{F}$ with respect to $\boldsymbol{p}$, and the Jacobian matrix $\mathbf{A} = \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})$ is the partial derivative of $\boldsymbol{F}(\boldsymbol{x},\boldsymbol{p})$ with respect to $\boldsymbol{x}$ evaluated at $\boldsymbol{x} = \boldsymbol{s}(\boldsymbol{p})$ and $\boldsymbol{p}$. A closed formula for the gradient, $\nabla \hat{f}(\boldsymbol{p})$ is then obtained by inserting the solution of (2.2) into (2.1), giving

$$\underbrace{\nabla \hat{f}(\boldsymbol{p})}_{1\times m} = -\underbrace{\nabla_{\boldsymbol{x}} f(\boldsymbol{s},\boldsymbol{p})}_{1\times n}\,\underbrace{\mathbf{A}^{-1}}_{n\times n}\,\underbrace{\nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})}_{n\times m} + \underbrace{\nabla_{\boldsymbol{p}} f(\boldsymbol{s},\boldsymbol{p})}_{1\times n}. \tag{2.3}$$

The computation time for evaluating (2.3) strongly depends on the size of the state, n, through the need to solve for the $n \times 1$ vector $\boldsymbol{s}$ and because of the need to evaluate and invert the $n \times n$ matrix $\mathbf{A}$. We note, however, that once the factors of $\mathbf{A}$ are available, they can be used to evaluate the first term on the right hand side for relatively little additional cost by first evaluating $\mathbf{A}^{-\mathsf{T}} \nabla_{\boldsymbol{x}} f(\boldsymbol{s},\boldsymbol{p})^{\mathsf{T}}$, which can be done with a single forward and backward substitution, and then multiplying its transpose by $\nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})$, rather than first evaluating $\mathbf{A}^{-1} \nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})$, which would consists of solving m linear systems instead of one.

We now turn to the Hessian, for which an analytical expression is obtained by differentiating (2.1). Using the compact tensor-product notation of Manton [27], we get

$$\begin{aligned}\nabla^2 \hat{f}(\boldsymbol{p}) = {} & \nabla_{\boldsymbol{xx}} f(\boldsymbol{s},\boldsymbol{p})\,(\nabla \boldsymbol{s} \otimes \nabla \boldsymbol{s}) + \nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})\,(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}}) \\ & + \nabla_{\boldsymbol{px}} f(\boldsymbol{s},\boldsymbol{p})\,(\mathbf{I}_{\boldsymbol{p}} \otimes \nabla \boldsymbol{s}) + \nabla_{\boldsymbol{pp}} f(\boldsymbol{s},\boldsymbol{p})\,(\mathbf{I}_{\boldsymbol{p}} \otimes \mathbf{I}_{\boldsymbol{p}}) \\ & + \nabla_{\boldsymbol{x}} f(\boldsymbol{s},\boldsymbol{p})\,\nabla^2 \boldsymbol{s},\end{aligned} \tag{2.4}$$

where we have omitted the $\boldsymbol{p}$ argument of $\nabla \boldsymbol{s}(\boldsymbol{p})$ and $\nabla^2 \boldsymbol{s}(\boldsymbol{p})$ for brevity. Here we give a brief explanation on how to interpret the tensor-product notation, and refer the interested reader to [27] for more details. In (2.4), the tensor products can be understood merely as separating the arguments for each combination of directions, which second-order derivatives are applied to. For example, evaluating $\nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}})$ is done by applying the $n \times m$ tensor, $\nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})$, to the combined direction $(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}})$. That is, one must contract the first dimension of $\nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})$ on the first dimension of $\nabla \boldsymbol{s}$ (which is a $n \times m$ matrix) and its second dimension on the first dimension of $\mathbf{I}_{\boldsymbol{p}}$ (which is the $m \times m$ identity). Effectively, $\nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}})$ results in a $m \times m$ matrix, which can be understood as the matrix product $\nabla \boldsymbol{s}^{\mathsf{T}} \nabla_{\boldsymbol{xp}} f(\boldsymbol{s},\boldsymbol{p})$.

Note that the evaluation of the expression on the right-hand side of (2.4) makes use of the second derivative of the steady-state solution, i.e., the $n \times m \times m$ tensor, $\nabla^2 \boldsymbol{s}(\boldsymbol{p})$. It is obtained by differentiating (2.2), which yields

$$\begin{aligned}0 = {} & \nabla_{\boldsymbol{xx}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,(\nabla \boldsymbol{s} \otimes \nabla \boldsymbol{s}) + \nabla_{\boldsymbol{xp}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}}) \\ & + \nabla_{\boldsymbol{px}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,(\mathbf{I}_{\boldsymbol{p}} \otimes \nabla \boldsymbol{s}) + \nabla_{\boldsymbol{pp}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,(\mathbf{I}_{\boldsymbol{p}} \otimes \mathbf{I}_{\boldsymbol{p}}) \\ & + \mathbf{A}\,\nabla^2 \boldsymbol{s}.\end{aligned} \tag{2.5}$$

In (2.5), note that every term in the sum is a $n \times m \times m$ tensor (none of which can be represented in matrix form, justifying our use of the tensor notation of [27]). For example, $\nabla_{\boldsymbol{xp}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,(\nabla \boldsymbol{s} \otimes \mathbf{I}_{\boldsymbol{p}})$ is the double contraction of the second dimension of the $n \times n \times m$ tensor $\nabla_{\boldsymbol{xp}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})$ on the first dimension of $\nabla \boldsymbol{s}$ and of its third dimension on the first dimension of $\mathbf{I}_{\boldsymbol{p}}$. Also note that $\mathbf{A}\,\nabla^2 \boldsymbol{s}$ must be understood as $\mathbf{A}$ multiplying each of the m^2 column vectors of dimension $n \times 1$ contained in the $n \times m \times m$ tensor $\nabla^2 \boldsymbol{s}$.

In principle, (2.5), can be substituted into the adjoint of (2.4) to compute the Hessian matrix by solving a single linear system involving the Jacobian matrix $\mathbf{A}$. Thus, at least one factorization of $\mathbf{A}$ and one forward and back substitutions are required to compute the Hessian with a closed analytical formula. However, numerically computing each term in (2.5) is tedious at best. Instead, a better solution is to compute the premultiplied terms arising from inserting (2.5) into (2.4) so as to avoid computing the third-order tensors, as suggested by [49].

2.2. The F-1 algorithm. For the gradient, the F-1 algorithm first uses (2.2) to compute $\nabla \boldsymbol{s}(\boldsymbol{p})$, which requires m forward and back substitutions and the factorization of $\mathbf{A}$. Once computed, $\nabla \boldsymbol{s}(\boldsymbol{p})$ is then inserted into (2.1) to evaluate the gradient. Importantly, the partial derivatives of $\boldsymbol{F}$ and f with respect to the parameters, $\boldsymbol{p}$, are computed numerically using dual numbers. Specifically, $\nabla_{\boldsymbol{p}} f(\boldsymbol{s},\boldsymbol{p})$ and $\nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})$ are computed in m dual-valued evaluations, via

$$\nabla_{\boldsymbol{p}} f(\boldsymbol{s},\boldsymbol{p}) = \mathfrak{D}\big[f(\boldsymbol{s},\boldsymbol{p}+\varepsilon \boldsymbol{e}_1),\ldots,f(\boldsymbol{s},\boldsymbol{p}+\varepsilon \boldsymbol{e}_m)\big] \tag{2.6}$$

and

$$\nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{s},\boldsymbol{p}) = \mathfrak{D}\big[\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p}+\varepsilon \boldsymbol{e}_1),\ldots,\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p}+\varepsilon \boldsymbol{e}_m)\big], \tag{2.7}$$

respectively.

For the Hessian, the F-1 algorithm uses hyperdual numbers, but exploits a combination of (2.4) and (2.5) that provides an optimally efficient analytical shortcut, which reduces the cost of computing the Hessian down to a single forward and back substitution. For $\boldsymbol{p}_{jk} = \boldsymbol{p} + \varepsilon_1 \boldsymbol{e}_j + \varepsilon_2 \boldsymbol{e}_k$, let $\boldsymbol{x}_{jk} \equiv \boldsymbol{s} + \varepsilon_1 \nabla \boldsymbol{s}\, \boldsymbol{e}_j + \varepsilon_2 \nabla \boldsymbol{s}\, \boldsymbol{e}_k$ denote a carefully chosen corresponding hyperdual-valued state. (We have dropped the $\boldsymbol{p}$ dependency on $\boldsymbol{s}(\boldsymbol{p})$ and on $\nabla \boldsymbol{s}(\boldsymbol{p})$ for brevity again.) Then, the entries of the Hessian matrix of the objective function are given by

$$[\nabla^2 \hat{f}(\boldsymbol{p})]_{jk} = \mathfrak{H}\big[f(\boldsymbol{x}_{jk}, \boldsymbol{p}_{jk})\big] - \mathfrak{H}\big[\boldsymbol{F}(\boldsymbol{x}_{jk}, \boldsymbol{p}_{jk})^\mathsf{T}\big]\, \mathbf{A}^{-\mathsf{T}}\, \nabla_{\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})^\mathsf{T}, \tag{2.8}$$

where $\mathfrak{H}(x)$ is the $\varepsilon_1\varepsilon_2$ coefficient of x. (A formal derivation of (2.8) is given later in this section.) With (2.8), a single forward and back substitution for $\mathbf{A}^{-\mathsf{T}}\, \nabla_{\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})^\mathsf{T}$ is required for all the entries of the Hessian because it is independent of j and k. Additionally, because the Hessian is symmetric, only $m(m+1)/2$ hyperdual-valued evaluations of f and $\boldsymbol{F}$ are necessary.

Hence, the F-1 algorithm computes the gradient in a single factorization of $\mathbf{A}$, m forward and back substitutions, and $\mathcal{O}(m)$ function evaluations, and computes the Hessian in a single forward and back substitution and $\mathcal{O}(m^2)$ function evaluations. Additionally, the F-1 algorithm requires only f, $\boldsymbol{F}$, $\nabla_{\boldsymbol{x}} f$, and $\nabla_{\boldsymbol{x}} \boldsymbol{F}$ from the user. We note that although the F-1 algorithm requires the user to supply the Jacobian, $\nabla_{\boldsymbol{x}} \boldsymbol{F}$, as well as $\nabla_{\boldsymbol{x}} f$, these may sometimes be easily derived analytically or computed numerically, as is the case for our phosphorus-cycling model (details in section SM2). Thus in some cases, with little extra work, the F-1 algorithm provides an automatic differentiation tool that requires no derivatives from the user — only $\boldsymbol{F}$ and f.

We note that (2.8) is similar to Equation (13) in the work of Rumpfkeil and Mavriplis [49], who suggest to premultiply the third-order tensors in (2.5) and use an algorithmic differention tool to compute the directional derivatives for each (j,k) direction. Here, we accomplish the same thing by simply evaluating f and $\boldsymbol{F}$ with appropriately chosen hyperdual-valued arguments. Thus, the F-1 algorithm provides an easy-to-implement and similarly optimally-efficient alternative.

We now derive (2.8). The hyperdual-valued Taylor expansion of f at $(\boldsymbol{s}, \boldsymbol{p})$ in the $(\varepsilon_1 \nabla \boldsymbol{s}\, \boldsymbol{e}_j + \varepsilon_2 \nabla \boldsymbol{s}\, \boldsymbol{e}_k, \varepsilon_1 \boldsymbol{e}_j + \varepsilon_2 \boldsymbol{e}_k)$ direction gives exactly the cross terms of (2.4) as its $\varepsilon_1\varepsilon_2$ coefficient. That is,

$$\begin{aligned} \mathfrak{H}\big[f(\boldsymbol{x}_{jk}, \boldsymbol{p}_{jk})\big] = {} & \boldsymbol{e}_j^\mathsf{T}\, \nabla \boldsymbol{s}^\mathsf{T}\, \nabla_{\boldsymbol{x}\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})\, \nabla \boldsymbol{s}\, \boldsymbol{e}_k + \boldsymbol{e}_j^\mathsf{T}\, \nabla \boldsymbol{s}^\mathsf{T}\, \nabla_{\boldsymbol{x}\boldsymbol{p}} f(\boldsymbol{s}, \boldsymbol{p})\, \boldsymbol{e}_k \\ & + \boldsymbol{e}_j^\mathsf{T}\, \nabla_{\boldsymbol{p}\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})\, \nabla \boldsymbol{s}\, \boldsymbol{e}_k + \boldsymbol{e}_j^\mathsf{T}\, \nabla_{\boldsymbol{p}\boldsymbol{p}} f(\boldsymbol{s}, \boldsymbol{p})\, \boldsymbol{e}_k, \end{aligned} \tag{2.9}$$

where $\mathfrak{H}(x)$ is the $\varepsilon_1\varepsilon_2$ coefficient of x. Mathematically, $\mathfrak{H}\big[f(\boldsymbol{x}_{jk}, \boldsymbol{p}_{jk})\big]$ is simply the second-order directional derivative of f at $(\boldsymbol{s}, \boldsymbol{p})$ in the combined $(\nabla \boldsymbol{s}\, \boldsymbol{e}_j, \boldsymbol{e}_j)$ and $(\nabla \boldsymbol{s}\, \boldsymbol{e}_k, \boldsymbol{e}_k)$ directions.

The entry-wise version of (2.4) can be rearranged and expressed as

$$[\nabla^2 \hat{f}(\boldsymbol{p})]_{jk} = \mathfrak{H}\big[f(\boldsymbol{x}_{jk}, \boldsymbol{p}_{jk})\big] + \nabla_{\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})\, [\nabla^2 \boldsymbol{s}]_{jk}, \tag{2.10}$$

where $[\nabla^2 \boldsymbol{s}]_{jk}$ is the $n \times 1$ column vector given by the second partial derivative of $\boldsymbol{s}(\boldsymbol{p})$ with respect to the jth and kth parameters (which contracts on the $1 \times n$ row vector multiplied to its left, $\nabla_{\boldsymbol{x}} f(\boldsymbol{s}, \boldsymbol{p})$, as per the tensor notation of [27] resulting in a scalar entry).

Similarly, the hyperdual-valued Taylor expansion of $\boldsymbol{F}$, also taken at $(\boldsymbol{s}, \boldsymbol{p})$ and in the same $(\varepsilon_1 \nabla \boldsymbol{s}\, \boldsymbol{e}_j + \varepsilon_2 \nabla \boldsymbol{s}\, \boldsymbol{e}_k, \varepsilon_1 \boldsymbol{e}_j + \varepsilon_2 \boldsymbol{e}_k)$ direction, gives exactly the cross terms

of (2.5) as its $\varepsilon_1\varepsilon_2$ coefficient. That is,

$$\begin{aligned}\mathfrak{H}\big[\boldsymbol{F}(\boldsymbol{x}_{jk},\boldsymbol{p}_{jk})\big] = \boldsymbol{e}_j^\mathsf{T}\,\nabla\boldsymbol{s}^\mathsf{T}\,\nabla_{\boldsymbol{xx}}\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,\nabla\boldsymbol{s}\,\boldsymbol{e}_k + \boldsymbol{e}_j^\mathsf{T}\,\nabla\boldsymbol{s}^\mathsf{T}\,\nabla_{\boldsymbol{xp}}\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,\boldsymbol{e}_k \\ + \boldsymbol{e}_j^\mathsf{T}\,\nabla_{\boldsymbol{px}}\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,\nabla\boldsymbol{s}\,\boldsymbol{e}_k + \boldsymbol{e}_j^\mathsf{T}\,\nabla_{\boldsymbol{pp}}\boldsymbol{F}(\boldsymbol{s},\boldsymbol{p})\,\boldsymbol{e}_k,\end{aligned} \tag{2.11}$$

which can be interpreted as the second-order directional derivative of $\boldsymbol{F}$ at $(\boldsymbol{s},\boldsymbol{p})$ in the combined $(\nabla\boldsymbol{s}\,\boldsymbol{e}_j,\boldsymbol{e}_j)$ and $(\nabla\boldsymbol{s}\,\boldsymbol{e}_k,\boldsymbol{e}_k)$ directions. Thus, the entry-wise version of (2.5) can be recast as

$$[\nabla^2\boldsymbol{s}]_{jk} = -\mathbf{A}^{-1}\,\mathfrak{H}\big[\boldsymbol{F}(\boldsymbol{x}_{jk},\boldsymbol{p}_{jk})\big]. \tag{2.12}$$

Inserting (2.12) into (2.10) and taking the adjoint yields (2.8).

3. Implementation. We chose the Julia language [3] for the implementation of this study because it affords, among many other features, (i) state-of-the-art factorization of sparse matrices and solving of sparse linear systems using the standard-library LinearAlgebra, SparseArrays, and SuiteSparse packages [5, 6], (ii) state-of-the-art optimizations using the Optim package [30, 31], and (iii) efficient implementations of the dual- and hyperdual-number types using the DualNumbers and HyperDualNumbers packages. We emphasize that different implementations are possible, so that one could use another scientific computing language or other packages. However, this combined choice of Julia and relevant packages allows for a simple yet fast, modern, and open-source implementation of the algorithms benchmarked in this study.

3.1. Algorithms for the gradient and Hessian.

3.1.1. F-1 algorithm. The Julia implementation of the F-1 algorithm is publicly available online as the F1Method package [42], which was developed for this study and used in the optimization benchmarks of section 4. Thanks to the expressivity and syntax of the Julia language, the code for the entire F1Method package requires just a few lines of code that closely match (2.1), (2.2), and (2.6)–(2.8). The F1Method package essentially defines five functions, which are called to update the steady-state solution, update a memory cache, and compute the objective, its gradient, and its Hessian, respectively.

The memory cache, denoted `mem`, used by the F1Method package, is an instance of a custom Julia type, `Mem`, which is used to store the results of reusable computations, i.e., the steady-state solution, $\boldsymbol{s}(\boldsymbol{p})$, the factors of the Jacobian, $\mathbf{A}$, the derivatives $\nabla\boldsymbol{s}(\boldsymbol{p})$ and $\nabla_{\boldsymbol{x}} f(\boldsymbol{s},\boldsymbol{p})$, and the corresponding parameter values, $\boldsymbol{p}$. When calling either of the objective, the gradient, or the Hessian functions, the contents of `mem` are only updated if the parameters, $\boldsymbol{p}$, are modified.

3.1.2. Other algorithms. We chose to benchmark the F-1 algorithm against five algorithms that apply their respective numerical scheme to either the analytical gradient function, $\nabla\hat{f}$, as defined by (2.3), or directly to the objective function, $\hat{f}$, as defined by (1.3), taken as black boxes. The algorithms that use the analytical gradient formula are (i) the FD1 algorithm, which applies a second-order finite-difference scheme, (ii) the CSD algorithm, which applies the complex-step scheme, and (iii) the DUAL algorithm, which applies the dual-step scheme. The algorithms that directly use the objective function to compute both gradient and Hessian are (iv) the FD2 algorithm, which applies second-order finite-difference schemes for both the gradient and Hessian, and (v) the HYPER algorithm, which applies the dual-step scheme for the gradient and the hyperdual-step scheme for the Hessian. These algorithms,

together with the F-1 algorithm, are collected in Table 1, with a link to their Julia implementation and a short description with an estimate of their computational costs. Overall, we thus benchmark six algorithms that are run in the same conditions and on the same machine, allowing for a fair set of comparisons.

TABLE 1

Collection of the algorithms benchmarked in this study. The FD1, CSD, and DUAL algorithms compute the Hessian, $\nabla^2 \hat{f}(\boldsymbol{p})$, by numerically differentiating the gradient, $\nabla \hat{f}$, using the analytical formula, (2.3). They require the user to supply f, $\boldsymbol{F}$, $\nabla_{\boldsymbol{x}} f$, $\nabla_{\boldsymbol{x}} \boldsymbol{F}$, $\nabla_{\boldsymbol{p}} f$, and $\nabla_{\boldsymbol{p}} \boldsymbol{F}$. The F-1 algorithm requires f, $\boldsymbol{F}$, $\nabla_{\boldsymbol{x}} f$, and $\nabla_{\boldsymbol{x}} \boldsymbol{F}$. The FD2 and HYPER algorithms compute both gradient and Hessian directly from the objective function and thus require only $\hat{f}$. The "cost" column shows the number of factorizations as a function of the number of parameters, m. Names in the first column are clickable URLs that point to their implementation in GitHub.

Algorithm	Cost	Definition
F-1	$\mathcal{O}(1)$	The algorithm presented in this study. Computes $\nabla \hat{f}(\boldsymbol{p})$ and $\nabla^2 \hat{f}(\boldsymbol{p})$ using the analytical shortcuts described in subsection 2.2. Overall requires 1 factorization and $m+1$ forward and back substitutions.
FD1	$\mathcal{O}(m)$	Central finite-differences algorithm. Computes $\nabla^2 \hat{f}(\boldsymbol{p})$ as the Jacobian of $\nabla \hat{f}$. Executes $2m$ calls to $\nabla \hat{f}$ (i.e., $2m$ factorizations) and to the inner solver (about 1 iteration each time). Overall requires about $4m$ additional factorizations.
CSD	$\mathcal{O}(m)$	Complex-step differentiation algorithm. Computes $\nabla^2 \hat{f}(\boldsymbol{p})$ by evaluating $\nabla \hat{f}$ with complex parameters. Executes m calls to the complex $\nabla \hat{f}$ (i.e., m complex factorizations) and to the inner solver (2 iterations each time). Overall requires about $2m$ additional complex factorizations. (Note that complex-valued operations can be more expensive than real-valued ones.)
DUAL	$\mathcal{O}(m)$	Dual-step differentiation algorithm. Computes $\nabla^2 \hat{f}(\boldsymbol{p})$ by evaluating $\nabla \hat{f}$ with dual parameters. Executes m calls to the dual $\nabla \hat{f}$ and to the inner solver (2 iterations each time). Overall requires about $2m$ additional dual factorizations. (Note the DUAL method applies (SM5.1) for solving dual-valued linear systems).
FD2	$\mathcal{O}(m^2)$	Second-order finite-differences algorithm. Computes $\nabla \hat{f}(\boldsymbol{p})$ and $\nabla^2 \hat{f}(\boldsymbol{p})$ directly from $\hat{f}$. Executes $2m$ calls to the inner solver for the gradient and $2m^2$ for the Hessian. Overall requires about $2m^2 + 2m$ additional factorizations. (Note that the low accuracy of the FD2 algorithm slows the convergence of the optimizer.)
HYPER	$\mathcal{O}(m^2)$	Dual- and Hyperdual-step differentiation algorithm. Computes $\nabla \hat{f}(\boldsymbol{p})$ and $\nabla^2 \hat{f}(\boldsymbol{p})$ by evaluating $\hat{f}$ with dual-valued and hyperdual parameters, respectively. Executes m calls to the inner solver with dual parameters (2 iterations each time), and $m(m+1)/2$ calls with hyperdual-valued parameters (3 iterations each time). Overall requires about m additional dual factorizations and $m(m+1)/2$ additional hyperdual factorizations. (Note that the HYPER algorithm applies (SM5.1) and (SM5.2) for dual and hyperdual linear systems.)

Importantly, we note that the CSD, DUAL, and HYPER algorithms invoke the inner solver with non-real number types. Thus, the inner solver must be able to check the convergence of the imaginary, dual, and hyperdual parts. In practice, the solver that we use applies the Shamanskii method, a quasi-Newton method that computes Newton steps [19, 20]. We thus add a conditional statement on the relative size of the non-real parts of the Newton step for the inner-solver loop to terminate.

Additionally, for the CSD, DUAL, and HYPER algorithms, the inner-solver Newton steps require solving complex-valued, dual-valued, or hyperdual-valued linear systems. While UMFPACK [5], which is the C package called by Julia's SuiteSparse package [6] for the LU factorization of unsymmetric sparse matrices (like $\mathbf{A}$ in our phosphorus-cycling model), can handle complex numbers, it cannot deal with dual or

hyperdual valued number types. Thus an important advantage of the F-1 algorithm in this regard is that despite using dual and hyperdual numbers, the F-1 algorithm does not require the solution to any dual-valued or hyperdual-valued linear systems. For the DUAL and HYPER algorithms — specifically, for the inner solver to handle dual-valued and hyperdual-valued factorizations and forward and back substitutions — we developed two Julia packages, DualMatrixTools [39] and HyperDualMatrixTools [40]. These packages afford efficient factorization of dual and hyperdual sparse matrices, as well as solving dual and hyperdual linear systems with a minimum number of forward and back substitutions. Both packages rely on analytical identities for the inverse of dual-valued and hyperdual-valued matrices that are derived in section SM5.

3.2. Phosphorus cycling model. To generate the global marine phosphorus cycling model, we use the AIBECS package (for Algebraic Implicit Biogeochemical Elemental Cycling System, [41]), which was developed in parallel to this study. In the AIBECS, steady-state problems are built as objects of the `SteadyStateProblem` type as defined by the DiffEqBase package [45] in Julia. The steady-state solution is computed via a state-of-the-art quasi-Newton algorithm implementing the Shamanskii method, derived from the work of [19, 20] and coded inside the AIBECS. We reiterate that the solver invoked by the AIBECS has been carefully designed to handle real-, complex-, dual-, and hyperdual-valued state and parameters, and integrates the DualMatrixTools [39] and HyperDualMatrixTools [40] packages for dual- and hyperdual-valued factorizations and forward and back substitutions of sparse linear systems.

Our phosphorus-cycling model consists of two marine tracers, dissolved inorganic phosphorus (DIP), i.e., phosphate, and particulate organic phosphorus (POP). DIP is transported by water currents and turbulent eddies, and is taken up by phytoplankton in the euphotic layer (where light is available for photosynthesis to occur) and converted to sinking POP. As it sinks, POP is remineralized into DIP. The sequence of uptake, sinking, and remineralization, provides the downward transport mechanism (the biological pump [54, 46, 1]). The system is in steady state when the circulation, which brings nutrients back to the surface, balances the biological pump. The model only explicitly tracks DIP and POP, thus the state of the system is entirely determined by the concentration fields of DIP and POP, denoted by $x_{\mathrm{DIP}}(\boldsymbol{r})$ and $x_{\mathrm{POP}}(\boldsymbol{r})$ at location $\boldsymbol{r}$.

The evolution of the system is determined by the coupled mass-conservation equations for DIP and POP. They are is given by

$$\begin{cases} \left(\partial_t + \nabla_{\boldsymbol{r}} \cdot [\boldsymbol{u} - \mathbf{K}\nabla_{\boldsymbol{r}}]\right) x_{\mathrm{DIP}} = -U(x_{\mathrm{DIP}}) + R(x_{\mathrm{POP}}) \\ \left(\partial_t + \nabla_{\boldsymbol{r}} \cdot \boldsymbol{w}\right) x_{\mathrm{POP}} = +U(x_{\mathrm{DIP}}) - R(x_{\mathrm{POP}}) \end{cases}, \tag{3.1}$$

where $\nabla_{\boldsymbol{r}}$ is the classical three-dimensional gradient operator. (We have ommitted the $\boldsymbol{r}$ dependency of the DIP and POP fields for brevity.) Equation (3.1) defines a system of two coupled PDEs where we assume no-flux (Neumann) boundary conditions at the land–ocean and atmosphere–ocean interfaces. On the left hand sides of (3.1), $\boldsymbol{u}$ is the 3D water velocity vector field, $\mathbf{K}$ is the 3×3 eddy-diffusivity tensor, and $\boldsymbol{w}$ is the particulate sinking velocity. The flux divergence of DIP due to the ocean's currents and eddies is thus represented by the action of the advective–eddy-diffusive transport operator, $\nabla_{\boldsymbol{r}} \cdot [\boldsymbol{u} - \mathbf{K}\nabla_{\boldsymbol{r}}]$, on x_{DIP}.

The flux divergence of sinking POP is represented by the action of $\nabla_{\boldsymbol{r}} \cdot \boldsymbol{w}$ on x_{POP}. The remineralization profile of POP is assumed to follow a power-law with depth after the observations of Martin [28]. Following [23], this is equivalent to assuming that

TABLE 2
Parameters for the global marine phosphorus-cycling model with their prior and posterior means.

Symbol	Description	Prior mean	Posterior mean	Unit
x^{geo}	Mean DIP concentration	2.17	2.12	$\text{mmol}\,\text{m}^{-3}$
k	Half-saturation constant (Michaelis-Menten)	10.00	6.62	$\mu\text{mol}\,\text{m}^{-3}$
w_0	Sinking velocity at surface	1.00	0.64	$\text{m}\,\text{d}^{-1}$
w'	Vertical gradient of sinking velocity	0.22	0.13	d^{-1}
κ	Dissolution rate constant (POP to DIP)	0.19	0.19	d^{-1}
τ	Maximum uptake rate timescale	30.00	236.52	d

the magnitude of the sinking velocity, w, increases linearly with depth, an approach we adopt here, such that $w = w'z + w_0$, where w' and w_0 are optimizable parameters.

On the right hand sides of (3.1), U and R are the local uptake and remineralization rates, respectively. The specific phosphate uptake by phytoplankton in the euphotic layer is modeled according to a simple Monod term [32] with maximum set by the timescale τ and half-saturation rate constant k, and the POP remineralization is modeled after a first order reaction depending only on the POP concentration. Hence, U and R are defined by

$$\begin{cases} U(x_{\text{DIP}}) \equiv \dfrac{x_{\text{DIP}}}{\tau}\dfrac{x_{\text{DIP}}}{x_{\text{DIP}}+k} & \text{where } z \le z_0 \\ R(x_{\text{POP}}) \equiv \kappa\, x_{\text{POP}} \end{cases}, \tag{3.2}$$

where τ, k, and κ are optimizable parameters. (For our discrete model grid, the depth of the bottom of the euphotic layer, z_0, lies at the bottom of the second layer, i.e., at about 73 m, below which $U \equiv 0$.)

Because (3.1) does not contain external sources and sinks to the system, the global means are not constrained and could be chosen arbitrarily (see, e.g., [25]). We prescribe the global mean phosphate concentration by slowly restoring the DIP concentration everywhere to a mean value, x^{geo}, with a timescale of $\tau_{\text{geo}} = 1\,\text{Myr}$ that is larger than the typical timescale for a tracer to be homogeneously mixed in the ocean. Thus, in practice, $(x^{\text{geo}} - x_{\text{DIP}})/\tau_{\text{geo}}$ is added to the right hand side of the DIP equation in (3.1), where x^{geo} is an optimizable parameter. We note that estimating the value of x^{geo} is of interest because the total inventory of DIP in the ocean, which is uncertain, is given by x^{geo} multiplied by the total volume of the ocean. The $m = 6$ optimizable parameters are collected in Table 2.

The continuous equations in (3.1) are discretized onto a 3D grid of the ocean. Specifically, the steady-state version of (3.1) is recast into (1.1) by rearranging the 3D concentration fields of DIP and POP into a state vector $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_{\text{DIP}} \\ \boldsymbol{x}_{\text{POP}} \end{bmatrix}$ and by replacing the linear operators for the flux divergences by large sparse matrices. For the advective–diffusive transport operator, we use the Ocean Circulation Inverse Model (OCIM1, [9, 7]), which defines the 3D grid of the ocean. With two tracers and the 200 160 boxes of the OCIM1 grid, the state vector, $\boldsymbol{x}$, has length $n = 400\,320$. (More details on creating the discrete model and on OCIM1 are given in section SM1.)

For the optimization, we define the objective function as the sum of the squared mismatch of the modeled state against observations and the posterior parameters against their prior mean. Specifically, the objective function is defined by

$$f(\boldsymbol{x}, \boldsymbol{p}) \equiv \frac{\omega_{\boldsymbol{x}}}{2}\delta\boldsymbol{x}^{\mathsf{T}}\boldsymbol{\Omega}_{\boldsymbol{x}}\delta\boldsymbol{x} + \frac{\omega_{\boldsymbol{p}}}{2}\delta\boldsymbol{\lambda}^{\mathsf{T}}\boldsymbol{\Omega}_{\boldsymbol{\lambda}}\delta\boldsymbol{\lambda}, \tag{3.3}$$

where $\omega_{\boldsymbol{x}}$ and $\omega_{\boldsymbol{p}}$ are hyper parameters that control the relative weights of the state and the parameters, respectively. In (3.3), $\delta\boldsymbol{x}$ is the difference between the modeled and observed DIP concentrations, where the observations are from the World Ocean Atlas (WOA18, [16, 43]). The diagonal matrix $\boldsymbol{\Omega}_{\boldsymbol{x}}$ is taken as the inverse of the covariance matrix from regridding the WOA18 data onto the OCIM1 grid. For $\delta\boldsymbol{\lambda}$, assuming prior log-normal distributions for the parameters, we use $\delta\boldsymbol{\lambda} \equiv \log(\boldsymbol{p}) - \boldsymbol{\mu}$, where $\boldsymbol{\mu}$ is the prior mean in logarithmic space. In practice, we feed $\boldsymbol{\lambda} = \log(\boldsymbol{p})$ to the optimizer instead of $\boldsymbol{p}$ so that the parameters remain positive throughout. (We note that while our parameters are necessarily positive, they need not be positive in general and in such a case one would forgo the logarithmic transformation.) The diagonal matrix $\boldsymbol{\Omega}_{\boldsymbol{\lambda}}$ is taken as the inverse of the prior covariance matrix in logarithmic space. (The non-logarithmic prior variances of the parameters are prescribed as the square of their non-logarithmic prior means.)

3.3. Optimizer. For the optimization, we use the Trust-Region Newton algorithm of Julia's Optim package [30, 31], which we use to optimize the parameters in logarithmic space. For the initial choice of parameters, we chose their prior means as collected in Table 2. The optimizer is deemed to have converged when the norm of the gradient of the objective function is less than 10^{-8}. The initial state is chosen to be equal to x^{geo} everywhere. Accurate measurements of computation times for the benchmarks are performed with the BenchmarkTools and TimerOutputs packages.

4. Results.

4.1. Optimized model. The prior values of the parameters, which we use as the initial guess in the optimization, are given in Table 2, along with the posterior values (i.e., the optimized values). In our phosphorus-cycling model, at the start of the optimization, for the first steady-state solution, $\boldsymbol{s}_0$, such that $\boldsymbol{F}(\boldsymbol{s}_0, \boldsymbol{p}_0) = 0$, the DIP field has a large mismatch with observations, with a root-mean-square error (RMSE) relative to the mean observed DIP of about 41 %. Our careful choice of the weights $\omega_{\boldsymbol{x}}$ and $\omega_{\boldsymbol{p}}$ ensures that the optimization effectively reduces the mismatch of the modeled state, i.e., the first term of (3.3), such that, at the end of the optimization, the RMSE of the modeled DIP field of the optimal steady-state solution, $\boldsymbol{s}(\hat{\boldsymbol{p}})$, for the optimal parameters $\hat{\boldsymbol{p}}$, is of only about 6 %. Figure 2 shows a number of diagnostics of the DIP field of this optimized steady-state solution.

The DIP field of the optimized solution is shown at a depth of about 919 m in Figure 2a. For the same depth, the relative mismatch with observations is shown in Figure 2b, revealing the mismatch range of approximately ±20 %. Most of the mismatch lies within ±5 % although there are some significant positive biases in the Arctic and negative biases at low-latitudes.

In order to investigate the mismatch at different depths, Figure 2c shows the horizontally-averaged DIP concentrations of the optimized solutions and of the observations for each of the Atlantic (ATL), Pacific (PAC), and Indian (IND) basins. The excellent fit shows that the optimized model captures the global vertical gradients, which quantify the strength of the biological pump, remarkably well. This is confirmed by Figure 2d, which allows us to evaluate the DIP mismatch at every location by showing the cost-weighed cumulative joint probability density function of the modeled and observed DIP fields. The joint distribution, which is concentrated on the 1 : 1 line, shows that the optimized model strongly agrees with the observations over the entire ocean.

We emphasize that the phosphorus-cycling model we use is simplistic in the sense

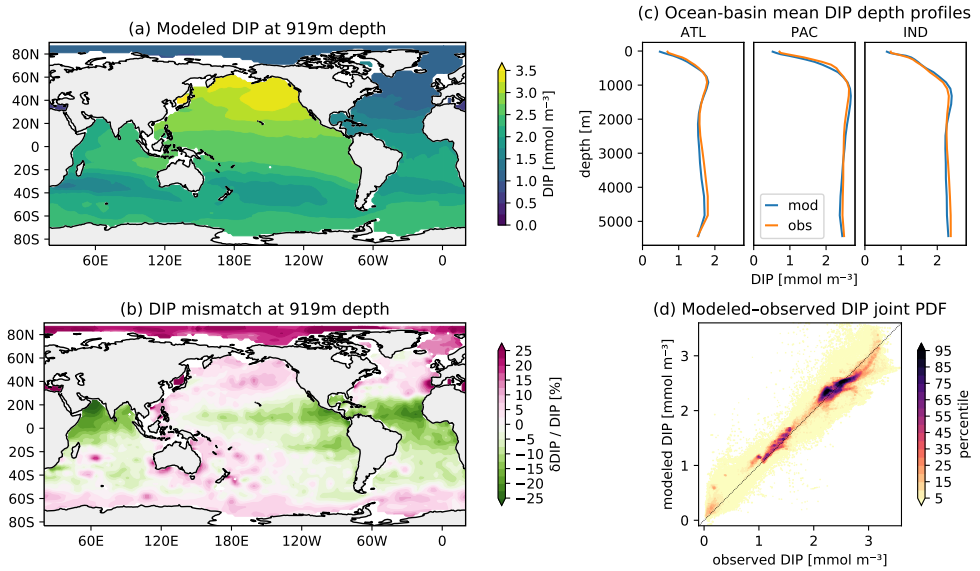

FIG. 2. *(a) Modeled DIP at about* 919 m *depth. (b) Relative mismatch between modeled DIP and observations at the same depth as (a). (c) Horizontally averaged (volume-weighted) concentrations of the modeled and observed DIP for the Atlantic (ATL), Pacific (PAC), and Indian (IND) oceans. (d) Volume- and inverse-variance-weighted cumulative joint probability density function of the modeled and observed DIP. (i% of the data lies outside of the contour of the ith percentile.)*

that it does not contain any information on light availability or other nutrient limitations, which are important controls on the distribution of DIP. Hence, the quality of the fit of the optimized steady-state solution to observations is remarkably good.

4.2. Benchmarks. We now compare the computation times afforded by using the F-1 algorithm against the other algorithms collected in Table 1 in the context of optimizing our global marine phosphorus-cycling model. We run the entire optimization procedure as illustrated in Figure 1. (Details of the implementation, such as the initial state and parameters, are described in section 3.)

Figure 3 shows the convergence rate of the optimization as the norm of the gradient, $\|\nabla \hat{f}(\boldsymbol{p})\|$, versus computation time. Recall that the gradient, $\nabla \hat{f}(\boldsymbol{p})$, must be equal to zero where the parameters are optimal. For all the algorithms, convergence is achieved in 9 iterations of the optimizer, except in the case of the FD2 algorithm, for which 10 iterations are needed. Using the FD2 algorithm requires more optimizer-loop iterations because both its gradient and Hessian are inaccurate. However, an accurate Hessian is not as important as an accurate gradient [18], so that the optimization run using the FD1 algorithm also converges in 9 iterations despite an inaccurate Hessian.

The optimization using the F-1 algorithm converges in about 7 min and is, by far, the fastest. Using the DUAL, FD1, and CSD algorithms, convergence is achieved in about 28 min, 40 min and 47 min, respectively. The HYPER algorithm converges in almost 2 h while the FD2 algorithm takes almost 3 h. The F-1 algorithm is about 24 times faster than the FD2 algorithm, which is the most common numerical differentiation algorithm.

We note that each optimization run includes unavoidable computations, regardless

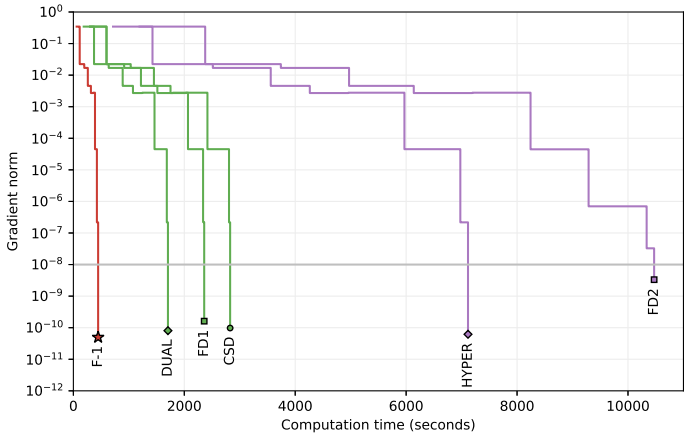

FIG. 3. *Convergence for all the algorithms from Table 1 quantified by the gradient norm, $\|\nabla \hat{f}(\boldsymbol{p})\|$, which is zero at the minimum, versus computation time. The algorithm is indicated at the end of each curve. The fastest algorithm is the F-1 algorithm (red star). Other Hessian-only computing algorithms (DUAL, FD1, and CSD) are shown in green, and gradient-and-Hessian-computing algorithms (HYPER and FD2) are shown in purple. The horizontal gray line indicates the tolerance of the optimizer, which terminates when $\|\nabla \hat{f}(\boldsymbol{p})\| \leq 10^{-8}$.*

of the algorithm used. For instance, every time the (real-valued) parameters, $\boldsymbol{p}$, are updated by the optimizer, a non-negligible fraction of the computation time is spent finding the corresponding steady-state solution, $\boldsymbol{s}(\boldsymbol{p})$, by invoking the inner solver. Below, we partition the computation time for the entire optimization into the time spent for the the gradient and Hessian. This allows to further untangle the differences in performance for each algorithm.

Figure 4 shows the partition of the time spent computing the gradient (in gray) and the Hessian (purple) during the same optimization runs as for Figure 3. In optimization problems, most of the time is usually spent computing the highest-order derivative, i.e., the Hessian in our context. This is true of all the algorithms except the F-1 algorithm. In fact, the optimization run using the F-1 algorithm is the only case where the time spent computing the Hessian is smaller than for computing the gradient.

Importantly, the cost of computing the Hessian only by the F-1 algorithm is spectacularly low. Specifically, the F-1 algorithm is about 16 times, 24 times, and 32 times faster than the DUAL, FD1, and CSD algorithms, for computing the Hessian. Furthermore, based on the number of factorizations of these algorithms, which scales like $\mathcal{O}(m)$, one should expect these performance ratios to roughly scale with the number of parameters, m. (Recall that $m = 6$ in our benchmarks.) For example, one would reasonably expect speedups of about two orders of magnitude with $m = 25$ parameters, and of about three orders of magnitude with $m = 250$. This is remarkable, considering the F-1 algorithm only requires the partials $\nabla_{\boldsymbol{x}} \boldsymbol{F}$ and $\nabla_{\boldsymbol{x}} f$ from the user, compared to the DUAL, FD1, and CSD algorithms, which are all state-of-the-art (although naive) applications of numerical differentiation, and which additionally require the analytical formula for $\nabla_{\boldsymbol{p}} \boldsymbol{F}$ and $\nabla_{\boldsymbol{p}} f$. We note however that for large m, the cost of function evaluations for the F-1 algorithm, which scales like $\mathcal{O}(m^2)$, may become predominant.

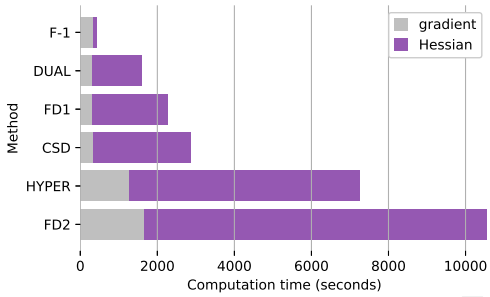

FIG. 4. *Cumulated computation times for the entire optimization run for all the algorithms. The computation time for the gradient, $\nabla \hat{f}$, and the Hessian, $\nabla^2 \hat{f}$, are indicated in gray and purple, respectively. (Computation times for the objective function are negligible in the optimization context — see details in section SM3.)*

A fairer comparison is against the HYPER and FD2 algorithms. Normalized by the number of calls (which is greater for the FD2 algorithm), the F-1 algorithm is effectively 76 times faster than the HYPER algorithm, and 100 times faster than the FD2 algorithm, for computing the Hessian. This is also spectacular, even more so when considering that the number of factorizations required by the HYPER and FD2 algorithms scales as $\mathcal{O}(m^2)$. In fact, one should expect the F-1 algorithm to be about 3 orders of magnitude faster than the HYPER and FD2 algorithms for $m \sim 20$ parameters, and 5 orders of magnitude faster for $m \sim 200$.

5. Conclusions. We presented a computationally efficient method, the F-1 algorithm, to numerically evaluate the gradient and Hessian of an objective function, $\hat{f}(\boldsymbol{p})$, which quantifies a model's skill (its ability to match observations) as a function of its parameters, $\boldsymbol{p}$. The algorithm is applicable to steady-state problems represented by a system of discretized nonlinear PDEs, $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) = 0$, for which the steady-state solution, $\boldsymbol{x} = \boldsymbol{s}(\boldsymbol{p})$, can be efficiently computed using a Newton-type solver. Additionally, the F-1 algorithm requires that the Jacobian matrix of the problem, $\nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$, can be created, stored, and factored. Requiring minimal input from the user, the F-1 algorithm performs significantly better than other state-of-the-art differentiation algorithms.

The F-1 algorithm relies on existing numerical differentiation schemes that are often-overlooked, even in advanced scientific applications. These techniques are based on the concepts of dual numbers and hyperdual numbers, which allow numerical differentiation of first and second derivatives, respectively, with machine-precision accuracy (see, e.g., [12, 14, 13, 37]). In addition to providing increased accuracy, using dual and hyperdual numbers is essential for the F-1 algorithm to be both fast and easy to implement.

While it builds on existing autodifferentiation tools [12, 14, 13, 37] and concepts [49], the F-1 algorithm elegantly combines them to leverage analytical shortcuts that we derive in this study. These shortcuts eliminate expensive calculations that are unavoidable when differentiating a black-box steady-state solver. In particular, they avoid redundant factorizations of the Jacobian. Specifically, the F-1 algorithm computes both gradient and Hessian in a single factorization, $\mathcal{O}(m)$ forward and back substitutions, and $\mathcal{O}(m^2)$ inexpensive function evaluations. Because factorizations are typically computationally expensive, the single-factorization feature of the F-1 algorithm affords large computational savings.

Naturally, the computational costs of computing gradients and Hessians depend on multiple variables, including the structure of the model itself. While we do not make any definitive statement on how the time complexity depends on this structure, it is reasonable to assume that it predominantly depends on the sparsity pattern of the Jacobian given by $\nabla_{\boldsymbol{x}}\boldsymbol{F}$. Furthermore, it is also reasonable to assume that computational costs scale as a function of the number of state variables and the number of parameters, i.e., as a function of n and m. However, the algorithms benchmarked in this study mainly differ by the number of factorizations they require, which is essentially a function of m only. Thus, throughout, we qualitatively estimated the expected computational costs as a function of m to explain the effective differences in performance and to extrapolate our performance estimates to problems of different sizes.

From our experience, the computational costs depend primarily on the number of factorizations, secondarily on the number of forward and back substitutions, and tertiarily on the number of function evaluations. Because the FD1, CSD, and DUAL algorithms apply a numerical differentiation scheme to the gradient, they require $\mathcal{O}(m)$ factorizations, forward and back substitutions, and function evaluations. Being applied to the objective function, the HYPER and FD2 algorithms are more expensive, requiring $\mathcal{O}(m^2)$ factorizations, forward and back substitutions, and function evaluations. Furthermore, the computational costs of each of these algorithms likely scale with the size of the state, n. Based on these considerations alone, the F-1 algorithm, which already outperforms the other algorithms by a large margin (in the context of our global marine phosphorus-cycling model), should outperform the other algorithms even more for larger problems, i.e., for larger m and n.

We demonstrated the computational performance of the F-1 algorithm by benchmarking it against five other state-of-the-art numerical differentiation algorithms (the DUAL, FD1, CSD, HYPER, and FD2 algorithms) in the context of optimizing a global marine phosphorus-cycling model, embedded in a global steady-state data-assimilated ocean circulation. The performance of the F-1 algorithm benefits from being used in the context of an optimization, during which previous computations, such as that of the factors of the Jacobian matrix, $\mathbf{A}$, can be reused. Overall, optimizing these $m = 6$ parameters takes about 7 minutes with the F-1 algorithm, against 28 to 47 minutes using the DUAL, CSD, or FD1 algorithms, about 2 hours using the HYPER algorithm, and almost 3 hours using the FD2 algorithm.

We further investigated performance by recording the time spent computing the gradient and Hessian. In particular, we found that for computing the Hessian only, the F-1 algorithm is about 16 to 32 times faster than the DUAL, CSD, and FD1 algorithms, which have access to the analytical gradient. Furthermore, the F-1 algorithm is 76 and 100 times faster than the HYPER and FD2 algorithms, respectively, for only $m = 6$ parameters.

We extrapolated our results to different problem sizes (different m) based on qualitative algorithmic complexity arguments. Assuming computation times scale primarily with the number of factorizations, we find that the performance of the F-1 algorithm would be amplified for larger problems. In particular, we expect the F-1 algorithm to be about 2 orders of magnitude faster than the DUAL, CSD, and FD1 algorithms with $m = 25$ parameters, and 3 orders of magnitude faster with $m = 250$. Furthermore, we expect the F-1 algorithm to be about 3 and 5 orders of magnitude faster than the HYPER and FD2 algorithms with $m = 20$ and $m = 200$, respectively.

In summary, we have presented an optimally efficient algorithm for computing the gradient and Hessian of the objective function for problems defined implicitly by the

steady-state solution of a system of discretized nonlinear PDEs. Our algorithm outperforms other state-of-the art algorithms for numerical differentiation, spectacularly so in the context of optimization. This is because classical numerical differentiation methods invoke nested iterative algorithms as black boxes, effectively repeating redundant computations, which incur significant computational costs. Instead, our algorithm leverages analytical shortcuts that are not available with naive black-box approaches. The performance gains likely scale with the size, n, of the system of PDEs, and with the number, m, of parameters, such that larger models, e.g., with finer resolution or more detailed mechanisms, would benefit even more from our algorithm than the benchmarks presented in this study.

The F-1 algorithm is ideally suited to a number of geoscientific model optimizations, provided the models can be represented by a steady-state PDE system of which the Jacobian can be stored and factored. However, the F-1 algorithm could potentially be extended to a larger scope of problems: (i) For very large m, where the optimizer does not create the full Hessian matrix but instead uses a matrix-free approach (i.e., only evaluates matrix–vector products). This is the case when one wishes to optimize a 3D field with a large number of entries, rather than a few scalar parameters. (ii) For very large n, where the inner solver similarly does not create the Jacobian matrix $\mathbf{A}$, but only evaluates matrix–vector products, e.g., using a Newton–Krylov type of solver (as in, e.g., [26, 49]). Other avenues of research include exploring potentially faster strategies for constrained optimization problems, for which the solver is not nested inside the optimizer, allowing for updates of the state, $\boldsymbol{x}$, outside of the manifod of steady-state solutions (i.e., not satisfying the steady-state condition at every update of the parameters, $\boldsymbol{p}$) [38]. Exploring the potential generalization of the F-1 algorithm to non-steady problems, as in [49] is also a promising research direction. Finally, the question remains whether the F-1 algorithm is applicable to problems that can leverage a distributed structure. In the case where the state vector can be separated into chunks that the solver can update in parallel, the computation times would be set by the sizes of the submatrices of the Jacobian now separated into smaller blocks. A parrallel solver and F-1 algorithm could be combined, e.g., in the offline optimization of much larger models than the phosphorus-cycling model presented here [2]. Such models are common in global marine biogeochemistry, e.g., in order to simulate a large number of marine tracers, like the Biogeochemical Elemental Cycling (BEC) model [33, 35, 34].

6. Discussion. Although our approach applies to a vast range of steady-state models defined through the implicit solution of a discretized system of nonlinear PDEs, it does not apply to all optimization problems of that form. Specifically, we focused on the cases where one is interested in computing the Hessian, with a particular focus on optimization algorithms that use quasi-Newton's methods (see, e.g., [53]). However, we should point out that there are different algorithms that can be used to minimize $\hat{f}$ that do not require the Hessian matrix. For example, some require only evaluations of the objective function, like the Simulated Annealing (e.g., [21]) and Nelder-Mead [36] algorithms. Others, like the Broyden-Fletcher-Goldfarb-Shanno (e.g., [38]), the gradient descent [4], and the conjugate gradient (e.g., [17]) algorithms, require evaluations of the gradient. It might be the case that the problem at hand is not suitable for a Newton-like method for the optimization algorithm, in which case the F-1 algorithm would not be needed.

Because it does not invoke the inner solver to compute derivatives, the F-1 algorithm avoids a number of implementation pitfalls that all other available algorithms

fall into. First, the F-1 algorithm allows for inner solvers that use non-real operations. For example, if the inner solver used the complex-step algorithm to compute the Jacobian, $\mathbf{A} = \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$, then the CSD algorithm could not be naively applied to compute the Hessian matrix because of pertubration confusion. Similarly, if the inner solver used the dual-step algorithm to compute the Jacobian, then the DUAL and HYPER algorithms would not work either. We note that it is possible *in theory* to carefully chose the size of the complex, dual, or hyperdual steps so that the CSD, DUAL, or HYPER algorithms work with an inner solver that uses non-real operations internally. However, we do not recommend such an approach because it comes at the risk of failing silently. We also note that perturbation confusion can be avoided by carerul autodifferentiation implementations [48, 47].

Second, the F-1 algorithm avoids having to carefully chose the step sizes. This is particularly important in the case of finite-difference methods (e.g., FD1 and FD2), for which if the step size h is too small, the inner-solver loop may not execute a single iteration, potentially causing large errors. In our implementation, an optimal choice for h was about $[\boldsymbol{p}]_j/10^4$ for the FD1 algorithm, and $[\boldsymbol{p}]_j/10^2$ for the non-diagonal terms of the Hessian for the FD2 algorithm, as can be seen in the code, accessible from the URLs in Table 1. ($[\boldsymbol{p}]_j$ denotes the jth optimizable parameter value.) Such a large relative step is likely the reason for the slower convergence (in terms of number of optimizer iterations) of the FD2 algorithm.

Third, the F-1 algorithm avoids having to carefully chose the tolerances of the iterative algorithms. In the case of the DUAL, CSD, and HYPER algorithms, an additional tolerance for each non-real part must be added to the inner solver, as detailed in section SM6, and the choice of said tolerance will matter. Although one may get away with forgetting to set the non-real tolerances, in this case the choice of the tolerance on the real part will determine when the inner loop terminates and may cause large errors.

Fourth, the F-1 algorithm does not need an inner solver that can handle complex, dual, or hyper-valued parameter or state inputs. In contrast, this is the case for the CSD, DUAL, and HYPER algorithms. In order for these algorithms to work, the inner-solver Newton steps go through solving complex-, dual-, and hyperdual-valued linear systems. As mentioned in subsection 3.1.2, the calls to underlying UMFPACK allow for complex-valued linear systems, but fails on dual- and hyperdual-valued systems. These failures compelled us to develop the DualMatrixTools and HyperDualMatrixTools packages specifically designed to overcome this likely common shortcoming: there is no guarantee that dual or hyperdual-valued numbers will be handled correctly by underlying package that solves linear systems.

Naturally, one should always avoid repeating expensive computations. This is what is accomplished by the memory cache in the implementation of the F-1 algorithm, which stores, e.g., the factors of $\mathbf{A}$, for multiple subsequent forward and back substitutions. While the other algorithms do not store all the information that the F-1 algorithm stores, they still keep the real-valued steady-state solution, $\boldsymbol{s}(\boldsymbol{p})$, in memory. A common strategy in computer sciences that would benefit the other algorithms is memoization of the factorization function, such that the factors of $\mathbf{A}$ would be computed only once, just like for the F-1 algorithm. However, while this seems like a good strategy at face value, it turns out that there is numerical noise as the state, $\boldsymbol{x}_l$, gets close to the theoretical steady-state solution, $\boldsymbol{s}(\boldsymbol{p})$. That is, the factors of $\mathbf{A}$ would be updated with a high probability at every update of either the state or the parameters, regardless of how small the change is. Additionally, storing a large number of factors of a large sparse matrix would likely cause memory issues.

In comparison, by leveraging exact analytical shortcuts, the F-1 algorithm provides a finely-tuned storage-and-reuse approach that avoids redundant computations in an optimal way.

Acknowledgments. FP and BP acknowledge funding from DOE grant DE-SC0016539 and NSF grant 1658380. The computations were performed using the Linux computational cluster Katana supported by the Faculty of Science, University of New South Wales, Australia.

REFERENCES

[1] D. E. Archer, G. Eshel, A. Winguth, W. Broecker, R. Pierrehumbert, M. Tobis, and R. Jacob, *Atmospheric pCO_2 sensitivity to the biological pump in the ocean*, Global Biogeochem. Cycles, 14 (2000), pp. 1219–1230, https://doi.org/10.1029/1999GB001216.

[2] A. Bardin, F. Primeau, and K. Lindsay, *An offline implicit solver for simulating prebomb radiocarbon*, Ocean Modelling, 73 (2014), pp. 45–58, https://doi.org/10.1016/j.ocemod.2013.09.008.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah, *Julia: A fresh approach to numerical computing*, SIAM Review, 59 (2017), pp. 65–98, https://doi.org/10.1137/141000671.

[4] A. Cauchy, *Méthode générale pour la résolution des systemes d'équations simultanées*, Comp. Rend. Sci. Paris, 25 (1847), pp. 536–538.

[5] T. A. Davis, *Algorithm 832: UMFPACK v4.3 — an unsymmetric-pattern multifrontal method*, ACM Trans. Math. Softw., 30 (2004), pp. 196–199, https://doi.org/10.1145/992200.992206.

[6] T. A. Davis, *Direct methods for sparse linear systems*, vol. 2, SIAM, 2006.

[7] T. DeVries, *The oceanic anthropogenic CO_2 sink: Storage, air-sea fluxes, and transports over the industrial era*, Global Biogeochem. Cycles, 28 (2014), pp. 631–647, https://doi.org/10.1002/2013GB004739.

[8] T. DeVries, C. Deutsch, P. A. Rafter, and F. Primeau, *Marine denitrification rates determined from a global 3-D inverse model*, Biogeosciences, 10 (2013), pp. 2481–2496, https://doi.org/10.5194/bg-10-2481-2013.

[9] T. DeVries and F. Primeau, *Dynamically and observationally constrained estimates of water-mass distributions and ages in the global ocean*, J. Phys. Oceanogr., 41 (2011), pp. 2381–2401, https://doi.org/10.1175/JPO-D-10-05011.1.

[10] T. DeVries and T. Weber, *The export and fate of organic matter in the ocean: New constraints from combining satellite and oceanographic tracer observations*, Global Biogeochemical Cycles, 31 (2017), pp. 535–555, https://doi.org/10.1002/2016GB005551.

[11] H. A. Dijkstra and W. Weijer, *Stability of the global ocean circulation: Basic bifurcation diagrams*, Journal of Physical Oceanography, 35 (2005), pp. 933–948, https://doi.org/10.1175/JPO2726.1.

[12] J. Fike and J. Alonso, *The development of hyper-dual numbers for exact second-derivative calculations*, in 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 2011, p. 886.

[13] J. A. Fike and J. J. Alonso, *Automatic differentiation through the use of hyper-dual numbers for second derivatives*, in Recent Advances in Algorithmic Differentiation, S. Forth, P. Hovland, E. Phipps, J. Utke, A. Walther, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, and T. Schlick, eds., vol. 87 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2012, pp. 163–173, https://doi.org/10.1007/978-3-642-30023-3_15.

[14] J. A. Fike, S. Jongsma, J. J. Alonso, and E. van der Weide, *Optimization with gradient and Hessian information calculated using hyper-dual numbers*, in AIAA paper 2011-3807, 29th AIAA Applied Aerodynamics Conference, 2011.

[15] M. Frants, M. Holzer, T. DeVries, and R. Matear, *Constraints on the global marine iron cycle from a simple inverse model*, Journal of Geophysical Research: Biogeosciences, 121 (2016), pp. 28–51, https://doi.org/10.1002/2015jg003111.

[16] H. E. Garcia, K. Weathers, C. R. Paver, I. Smolyar, T. P. Boyer, R. A. Locarnini, M. M. Zweng, A. V. Mishonov, O. K. Baranova, D. Seidov, and J. R. Reagan, *World Ocean Atlas 2018*, Volume 4: Dissolved Inorganic Nutrients (phosphate, nitrate and nitrate+nitrite, silicate) (in preparation). A. Mishonov Technical Ed.

[17] W. W. Hager and H. Zhang, *Algorithm 851: CG_DESCENT, a Conjugate Gradient Method with Guaranteed Descent*, ACM Trans. Math. Softw., 32 (2006), pp. 113–137, https://doi.org/10.1145/1132973.1132979.

[18] C. T. Kelley, *Iterative Methods for Optimization*, Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1999.

[19] C. T. Kelley, *Solving Nonlinear Equations with Newton's Method*, SIAM, 2003, ch. 1. Introduction, pp. 1–25, https://doi.org/10.1137/1.9780898718898.ch1.

[20] C. T. Kelley, *Solving Nonlinear Equations with Newton's Method*, SIAM, 2003, ch. 2. Finding the Newton Step with Gaussian Elimination, pp. 27–55, https://doi.org/10.1137/1.9780898718898.ch2.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science, 220 (1983), pp. 671–680, https://doi.org/10.1126/science.220.4598.671.

[22] O. Kolditz and H.-J. Diersch, *Quasi-steady-state strategy for numerical simulation of geothermal circulation in hot dry rock fractures*, International Journal of Non-Linear Mechanics, 28 (1993), pp. 467–481, https://doi.org/10.1016/0020-7462(93)90020-L.

[23] I. Kriest and A. Oschlies, *On the treatment of particulate organic matter sinking in large-scale models of marine biogeochemical cycles*, Biogeosciences, 5 (2008), pp. 55–72, https://doi.org/10.5194/bg-5-55-2008.

[24] E. Y. Kwon and F. Primeau, *Sensitivity and optimization study of a biogeochemistry ocean model using an implicit solver and in-situ phosphate data*, Global Biogeochem. Cycles, 20 (2006), GB4009, https://doi.org/10.1029/2005GB002631.

[25] E. Y. Kwon and F. Primeau, *Optimization and sensitivity of a global biogeochemistry ocean model using combined in situ DIC, alkalinity, and phosphate data*, Journal of Geophysical Research: Oceans, 113 (2008), https://doi.org/10.1029/2007JC004520. C08011.

[26] X. Li and F. W. Primeau, *A fast Newton–Krylov solver for seasonally varying global ocean biogeochemistry models*, Ocean Modelling, 23 (2008), pp. 13–20, https://doi.org/10.1016/j.ocemod.2008.03.001.

[27] J. H. Manton, *Differential Calculus, Tensor Products and the Importance of Notation*, ArXiv e-prints, (2012), https://arxiv.org/abs/1208.0197v2, https://arxiv.org/abs/1208.0197.

[28] J. W. Martin, G. A. Knauer, D. M. Karl, and W. W. Broenkow, *VERTEX: Carbon cycling in the NE Pacific*, Deep-Sea Research, 34 (1987), pp. 267–285.

[29] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso, *The complex-step derivative approximation*, ACM Trans. Math. Softw., 29 (2003), pp. 245–262, https://doi.org/10.1145/838250.838251.

[30] P. K. Mogensen and A. N. Riseth, *Optim: A mathematical optimization package for Julia*, Journal of Open Source Software, 3 (2018), p. 615, https://doi.org/10.21105/joss.00615.

[31] P. K. Mogensen, J. M. White, A. N. Riseth, T. Holy, M. Lubin, C. Stocker, A. Levitt, C. Ortner, B. Johnson, A. Noack, Y. Yu, K. Carlsson, D. Lin, T. R. Covert, R. Rock, J. Regier, B. Kuhn, A. Williams, Ryan, K. Sato, D. Smith, R. Anantharaman, M. Gomez, J. Revels, I. Dunning, D. MacMillen, C. Rackauckas, B. Legat, and A. Stukalov, *JuliaNLSolvers/Optim.jl: Bugfix for Fminbox, better docs, and experimental maximize function*, Sept. 2018, https://doi.org/10.5281/zenodo.1412092.

[32] J. Monod, *Microbiologie: Recherches sur la croissance des cultures bactériennes. I*, Actualités scientifiques et industrielles, Hermann & cie, 1942.

[33] J. Moore, S. C. Doney, J. A. Kleypas, D. M. Glover, and I. Y. Fung, *An intermediate complexity marine ecosystem model for the global domain*, Deep Sea Research Part II: Topical Studies in Oceanography, 49 (2001), pp. 403 – 462, https://doi.org/10.1016/S0967-0645(01)00108-4. The US JGOFS Synthesis and Modeling Project: Phase 1.

[34] J. K. Moore and O. Braucher, *Sedimentary and mineral dust sources of dissolved iron to the world ocean*, Biogeosciences, 5 (2008), pp. 631–656, https://doi.org/10.5194/bg-5-631-2008.

[35] J. K. Moore, S. C. Doney, and K. Lindsay, *Upper ocean ecosystem dynamics and iron cycling in a global three-dimensional model*, Global Biogeochem. Cycles, 18 (2004), GB4028, https://doi.org/10.1029/2004GB002220.

[36] J. A. Nelder and R. Mead, *A simplex method for function minimization*, The Computer Journal, 7 (1965), pp. 308–313, https://doi.org/10.1093/comjnl/7.4.308.

[37] M. Neuenhofen, *Review of theory and implementation of hyper-dual numbers for first and second order automatic differentiation*, CoRR, abs/1801.03614 (2018), http://arxiv.org/abs/1801.03614, https://arxiv.org/abs/1801.03614.

[38] J. Nocedal and S. Wright, *Numerical optimization*, Springer Science & Business Media, 2006.

[39] B. Pasquier, *DualMatrixTools: A Julia package for solving dual-valued linear systems*, Nov. 2018, https://doi.org/10.5281/zenodo.1493571.
[40] B. Pasquier, *HyperDualMatrixTools: A Julia package for solving hyperdual-valued linear systems*, Nov. 2018, https://doi.org/10.5281/zenodo.1670841.
[41] B. Pasquier, *AIBECS.jl: The ideal tool for exploring global marine biogeochemical cycles*, May 2019, https://doi.org/10.5281/zenodo.2864051.
[42] B. Pasquier, *F1Method.jl: A julia package for computing the gradient and Hessian of an objective function defined implicitly by the solution to a steady-state problem*, May 2019, https://doi.org/10.5281/zenodo.2667835.
[43] B. Pasquier, *Worldoceanatlastools.jl: Download and process world ocean data in julia*, May 2019, https://doi.org/10.5281/zenodo.2677666.
[44] B. Pasquier and M. Holzer, *Inverse-model estimates of the ocean's coupled phosphorus, silicon, and iron cycles*, Biogeosciences, 14 (2017), pp. 4125–4159, https://doi.org/10.5194/bg-14-4125-2017.
[45] C. Rackauckas and Q. Nie, *Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia*, Journal of Open Research Software, 5 (2017), https://doi.org/10.5334/jors.151.
[46] J. A. Raven and P. G. Falkowski, *Oceanic sinks for atmospheric CO2*, Plant, Cell & Environment, 22 (1999), pp. 741–755, https://doi.org/10.1046/j.1365-3040.1999.00419.x.
[47] J. Revels, T. Besard, M. J. Innes, R. Deits, M. Piibeleht, M. Schauer, L. White, and K. Fischer, *Cassette.jl*, Jan. 2019, https://doi.org/10.5281/zenodo.2549715.
[48] J. Revels, M. Lubin, and T. Papamarkou, *Forward-mode automatic differentiation in julia*, arXiv:1607.07892 [cs.MS], (2016), https://arxiv.org/abs/1607.07892.
[49] M. P. Rumpfkeil and D. J. Mavriplis, *Efficient hessian calculations using automatic differentiation and the adjoint method with applications*, AIAA Journal, 48 (2010), pp. 2406–2417, https://doi.org/10.2514/1.J050451.
[50] G. A. Schmidt and L. A. Mysak, *Can increased poleward oceanic heat flux explain the warm cretaceous climate?*, Paleoceanography, 11 (1996), pp. 579–593, https://doi.org/10.1029/96PA01851.
[51] D. Sivia and J. Skilling, *Data analysis: a Bayesian tutorial*, OUP Oxford, 2006.
[52] Y. Teng, F. W. Primeau, J. K. Moore, M. W. Lomas, and A. Martiny, *Global-scale variations of the ratios of carbon to phosphorus in exported marine organic matter*, Nature Geosci., 7 (2014), pp. 895–898, https://doi.org/10.1038/NGEO2303.
[53] G. Venter, *Review of optimization techniques*, Encyclopedia of aerospace engineering, (2010).
[54] T. Volk and M. Hoffert, *Ocean carbon pumps: analysis of relative strengths and efficiencies in ocean-driven atmospheric CO_2 changes.*, American Geophysical Union; Geophysical Monograph 32, 1985, pp. 99–110.
[55] W.-L. Wang, C. Lee, and F. W. Primeau, *A bayesian statistical approach to inferring particle dynamics from in-situ pump poc and chloropigment data from the mediterranean sea*, Marine Chemistry, (2019), https://doi.org/10.1016/j.marchem.2019.04.006.
[56] W. Yu and M. Blair, *DNAD, a simple tool for automatic differentiation of Fortran codes using dual numbers*, Computer Physics Communications, 184 (2013), pp. 1446–1452, https://doi.org/10.1016/j.cpc.2012.12.025.