SUPPLEMENTARY MATERIALS: 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†

SM1. Global marine phosphorus-cycling model. Here, we describe the discretization of the phosphorus-cycling model that we use to demonstrate the performance of the F-1 algorithm. Specifically, we detail how the physical and biogeochemical mechanisms are converted into the discretized generic equation (1.1) by explicitly defining the state function $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$. In practice, we use the AIBECS package [SM20] to download the ocean circulation and generate the state function, $\boldsymbol{F}$, the mismatch function f, and their derivatives.

SM1.1. The ocean circulation. We use the control (CTL) circulation output from the Ocean Circulation Inverse Model (OCIM1) of DeVries [SM3]. The OCIM optimizes a steady-state circulation by assimilating oceanographic tracer data (see [SM5]), which include potential temperature, salinity, radiocarbon, and CFC-11. The resulting circulation thus effectively represents an estimate of the mean state of the global ocean circulation and is formulated as a sparse matrix, denoted here by $\mathbf{T}_{\mathrm{DIP}}$, to facilitate rapid simulation of biogeochemical tracers. The circulation is embedded in an Arakawa B-grid with a $2^\circ \times 2^\circ$ resolution and 24 depth levels with thicknesses increasing from $\sim$25 m at the surface to $\sim$500 m for the deepest layer. (This amounts to 200 160 ocean grid cells.) In practice, we use the AIBECS package [SM20] to download the OCIM matrix.

With the state of the system represented by the column vector $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_{\mathrm{DIP}} \\ \boldsymbol{x}_{\mathrm{POP}} \end{bmatrix}$, the OCIM1 transport matrix is applied to the DIP vector only, $\boldsymbol{x}_{\mathrm{DIP}}$. That is, the matrix–vector product, $\mathbf{T}_{\mathrm{DIP}}\, \boldsymbol{x}_{\mathrm{DIP}}$, yields the flux divergence of DIP due to the ocean circulation. In other words, $\mathbf{T}_{\mathrm{DIP}}\, \boldsymbol{x}_{\mathrm{DIP}}$ is the discrete equivalent of $\nabla_{\boldsymbol{r}} \cdot [\boldsymbol{u} - \mathbf{K}\nabla]\, x_{\mathrm{DIP}}$ in (3.1).

SM1.2. The sinking particles. We now describe how $\mathbf{T}_{\mathrm{POP}}$, a sparse matrix representing the discrete equivalent of $\nabla \cdot \boldsymbol{w}$ in (3.1), is created. At the bottom of each model box, the sinking velocity is hence defined by

$$\boldsymbol{w} \equiv w' \boldsymbol{z}_{\mathrm{bot}} + w_0, \tag{SM1.1}$$

where $\boldsymbol{z}_{\mathrm{bot}}$ is a vector of the depths of the bottom of each grid box and $\boldsymbol{w}$ is the vector of the magnitudes of the downward particulate velocity. Note that the $\boldsymbol{w}$ symbol defined in (SM1.1) is different from the $\boldsymbol{w}$ symbol used for (3.1), which was the 3D velocity vector. The flux-divergence operator, $\mathbf{T}_{\mathrm{POP}}$, is defined such that $\mathbf{T}_{\mathrm{POP}}\, \boldsymbol{x}_{\mathrm{POP}}$ is the divergence of the sinking POP flux. The flux of sinking POP is approximated by $\boldsymbol{w}\, \boldsymbol{x}_{\mathrm{POP}}$ at the bottom of each model box.

*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).

The flux divergence is then approximated by the difference between the flux at the top and at the bottom of each box. Specifically, the sinking transport operator $\mathbf{T}_{\mathrm{POP}}$ is defined by

$$\mathbf{T}_{\mathrm{POP}} \equiv \Delta\mathbf{Z}^{-1}(\mathbf{U}_{\boldsymbol{x}_{\mathrm{POP}}} - \mathbf{I}_{\boldsymbol{x}_{\mathrm{POP}}})\mathbf{W}, \tag{SM1.2}$$

where $\Delta\mathbf{Z}$ is a diagonal matrix with diagonal $\Delta\mathbf{z}$, which is the vector of the depth thickness of each box, $\mathbf{I}_{\boldsymbol{x}_{\mathrm{POP}}}$ is the identity matrix, and $\mathbf{W}$ is the diagonal matrix with diagonal $\boldsymbol{w}$. The matrix $\mathbf{U}_{\boldsymbol{x}_{\mathrm{POP}}}$ consists of 1's on a superdiagonal similar and is thus upper triangular. It is defined similarly to a shift matrix, such that it shifts indices to the boxes above, so that $(\mathbf{U}_{\boldsymbol{x}_{\mathrm{POP}}} - \mathbf{I}_{\boldsymbol{x}_{\mathrm{POP}}})\mathbf{W}\boldsymbol{x}_{\mathrm{POP}}$ effectively yields the vector of the differences between the flux of POP at the top and at the bottom of each grid box. Thus, $\mathbf{T}_{\mathrm{POP}}$, is a function of the parameters w' and w_0 only.

SM1.3. The state function. The rate of change of the system is defined by

$$\boldsymbol{F}(\boldsymbol{x},\boldsymbol{p}) \equiv \begin{bmatrix} -\mathbf{T}_{\mathrm{DIP}}\,\boldsymbol{x}_{\mathrm{DIP}} - \boldsymbol{U} + \boldsymbol{R} + \boldsymbol{C} \\ -\mathbf{T}_{\mathrm{POP}}\,\boldsymbol{x}_{\mathrm{POP}} + \boldsymbol{U} - \boldsymbol{R} \end{bmatrix}, \tag{SM1.3}$$

where the column vectors $\boldsymbol{U}$ and $\boldsymbol{R}$ are the uptake and remineralization rates, respectively, converting DIP to POP and back (U and R in (3.1)).

The discrete equivalent of the uptake, remineralization, and geological restoring rates, as defined in (3.2), can be defined compactly by

$$\begin{cases} \boldsymbol{U} \equiv \dfrac{\boldsymbol{x}^+_{\mathrm{DIP}}}{\tau}\dfrac{\boldsymbol{x}^+_{\mathrm{DIP}}}{\boldsymbol{x}^+_{\mathrm{DIP}} + k}(\boldsymbol{z} \le z_0) \\ \boldsymbol{R} \equiv \kappa \boldsymbol{x}_{\mathrm{POP}} \\ \boldsymbol{C} \equiv \dfrac{\langle x^{\mathrm{geo}}\rangle - \boldsymbol{x}_{\mathrm{DIP}}}{\tau_{\mathrm{geo}}} \end{cases}, \tag{SM1.4}$$

where $\boldsymbol{x}^+_{\mathrm{DIP}} = \boldsymbol{x}_{\mathrm{DIP}}$ if $\boldsymbol{x}_{\mathrm{DIP}} \ge 0$ and $\boldsymbol{x}^+_{\mathrm{DIP}} = 0$ otherwise, ensuring that uptake only occurs for positive concentrations. (This is to avoid problems due to the hyperbolic term in the uptake when numerical noise generates negative concentrations.) Similarly, the $(\boldsymbol{z} \le z_0)$ term ensures that uptake only occurs in the euphotic zone. ($\boldsymbol{z}$ is the vector of box depths, oriented positively downwards, and $z_0 \equiv 75\,\mathrm{m}$ is the approximate depth of the bottom of the euphotic layer in the OCIM1 grid.)

Note that we used two shortcuts to simplify notation in (SM1.4) that we use throughout this section: (i) Boolean expressions are assumed to convert automatically to the real-valued 0 or 1, and (ii) binary vector operations are assumed to be elementwise. Thus, for example, in (SM1.4), each entry of $\boldsymbol{x}^+_{\mathrm{DIP}}$ is divided by the corresponding entry in $(\boldsymbol{x}^+_{\mathrm{DIP}} + k)$, and $(\boldsymbol{z} \ge z_0)$ is a vector of 0's and 1's with its entries multiplied by the corresponding entries of $\boldsymbol{x}^+_{\mathrm{DIP}}/(\boldsymbol{x}^+_{\mathrm{DIP}} + k)$.

SM2. Derivatives with respect to the state. The scope of the F-1 algorithm may appear restricted by the fact that it requires the user to supply the Jacobian function, $\nabla_{\boldsymbol{x}}\boldsymbol{F}$. However, the Jacobian function, $\nabla_{\boldsymbol{x}}\boldsymbol{F}$, which is required for Newton-like inner solvers, may be easy to derive analytically. Additionally, because many discretizations for PDEs use local low-order finite difference schemes that produce sparse Jacobian matrices, $\nabla_{\boldsymbol{x}}\boldsymbol{F}(\boldsymbol{x},\boldsymbol{p})$ is likely sparse, so that it can be computed numerically in only a few dual passes using a graph-coloring algorithm (see, e.g., [SM10]). Furthermore, it is likely that the Jacobian of the nonlinear part of $\boldsymbol{F}$ is

even sparser, e.g., diagonal or made of blocks that are diagonal, allowing for it to be computed in even less dual passes.

For instance, in the context of the global marine phosphorus cycling model (section SM1), the Jacobian of the state function $\boldsymbol{F}$ defined by (SM1.3) can be evaluated in just two dual evaluations. Consider the partition of $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$ into the sum of a linear part (with respect to $\boldsymbol{x}$), denoted by $-\mathbf{T}(\boldsymbol{p})\,\boldsymbol{x}$, where

$$\mathbf{T}(\boldsymbol{p}) \equiv \begin{bmatrix} \mathbf{T}_{\mathrm{DIP}} & 0 \\ 0 & \mathbf{T}_{\mathrm{POP}} \end{bmatrix} \tag{SM2.1}$$

is a large block-diagonal sparse matrix, and a local part, $\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{p})$, which consists of the uptake, $\boldsymbol{U}$, remineralization, $\boldsymbol{R}$, and geological restoring, $\boldsymbol{C}$ (see section SM1). The Jacobian of the local part, $\nabla_{\boldsymbol{x}}\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{p})$, is then made of blocks that are diagonal, i.e.,

$$\nabla_{\boldsymbol{x}}\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{p}) \equiv \begin{bmatrix} \nabla_{\mathrm{DIP}}\boldsymbol{C} - \nabla_{\mathrm{DIP}}\boldsymbol{U} & \nabla_{\mathrm{POP}}\boldsymbol{R} \\ \nabla_{\mathrm{DIP}}\boldsymbol{U} & -\nabla_{\mathrm{POP}}\boldsymbol{R} \end{bmatrix} \tag{SM2.2}$$

where $\nabla_{\mathrm{DIP}}\boldsymbol{C}$, $\nabla_{\mathrm{DIP}}\boldsymbol{U}$, and $\nabla_{\mathrm{POP}}\boldsymbol{R}$ are the derivatives of $\boldsymbol{C}$, $\boldsymbol{U}$, and $\boldsymbol{R}$ with respect to $\boldsymbol{x}_{\mathrm{DIP}}$ or $\boldsymbol{x}_{\mathrm{POP}}$, as indicated, and are diagonal matrices because $\boldsymbol{C}$, $\boldsymbol{U}$, and $\boldsymbol{R}$ are local rates. Because these are diagonals, $\nabla_{\boldsymbol{x}}\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{p})$ can be computed in two dual evaluations, one where ε is added to each entry of $\boldsymbol{x}_{\mathrm{DIP}}$, and one where ε is added to each entry of $\boldsymbol{x}_{\mathrm{POP}}$. The full Jacobian, $\nabla_{\boldsymbol{x}}\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$ is then simply given by $-\mathbf{T}(\boldsymbol{p}) + \nabla_{\boldsymbol{x}}\boldsymbol{G}(\boldsymbol{x}, \boldsymbol{p})$. (This is for example implemented under the hood in the AIBECS package to generate $\nabla_{\boldsymbol{x}}\boldsymbol{F}$ automatically from $\mathbf{T}_{\mathrm{DIP}}$, $\mathbf{T}_{\mathrm{POP}}$, and the local sources and sinks [SM20].)

A similar potential restriction to the scope of the F-1 algorithm is that it requires the user to supply $\nabla_{\boldsymbol{x}} f$. However, an analytical formula for $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{p})$ is straightforwardly available if $f(\boldsymbol{x}, \boldsymbol{p})$ is expressed as the sum of the quadratics, as is the case for the objective function of our phosphorus-cycling model. That is, the state and parameter parts of $f(\boldsymbol{x}, \boldsymbol{p})$ are usually separable, so that $f(\boldsymbol{x}, \boldsymbol{p}) = f(\boldsymbol{x}) + f(\boldsymbol{p})$, and the state part, $f(\boldsymbol{x})$, usually takes the generic form $f(\boldsymbol{x}) = \frac{1}{2}\delta\boldsymbol{x}^{\mathsf{T}}\,\boldsymbol{\Omega}\,\delta\boldsymbol{x}$, where the precision matrix $\boldsymbol{\Omega}$ is diagonal, so that

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{p}) = \delta\boldsymbol{x}^{\mathsf{T}}\boldsymbol{\Omega} \tag{SM2.3}$$

can be readily used as the analytical formula supplied to the F-1 algorithm. (Here we have abused notation to distinguish the state and parameter parts of $f(\boldsymbol{x}, \boldsymbol{p})$ depending on its arguments, similar to the multiple-dispatch paradigm of the Julia language.) Hence, we argue that in many scientific applications, the F-1 algorithm can be used as a fully automatic differentiation tool with little additional effort, e.g., compared to the HYPER or FD2 algorithms.

SM3. Comment on the computation time partition. We note that in Figure 4, the time spent by the inner solver is attributed to time spent on the gradient. This is because the Newton-Trust-Region algorithm we use for the optimization computes the gradient first, then the Hessian, and finally the objective, in that specific order. Thus, the inner solver is always invoked by the gradient first, making $\boldsymbol{s}(\boldsymbol{p})$ (and the factors of $\mathbf{A}$ in the case of the F-1 algorithm) available for the subsequent Hessian and objective computations. The objective function is computed last because it is not needed in Newton's method for minimization and is merely evaluated for recording the progress of the optimization. Hence, the time attributed to the objective function, $\hat{f}(\boldsymbol{p})$, is merely the time of a single evaluation of $f(\boldsymbol{s}, \boldsymbol{p})$, which is negligibly

small relative to the other computations (the corresponding bars are not visible in Figure 4).

SM4. Extending real numbers for differentiation.

SM4.1. Complex numbers. Since its discovery [SM12], and despite its elegance and efficiency, it took nearly 30 years before the complex-step algorithm for differentiation was described in the literature, by Squire and Trapp [SM24]. More recently, Cleve Moler wrote a simple blog post [SM16] describing the technique on the MathWorks website in 2013, and Martins et al. [SM13] wrote an article dedicated to it. Consider a simple generic real-valued function g, which depends on a real-valued scalar variable. The Taylor expansion of g at x in the ih direction, where i is the imaginary unit and h is a small real-valued scalar, is given by

$$g(x+ih) = g(x) + ih\nabla g(x) + \underbrace{\frac{(ih)^2}{2}\nabla^2 g(x) + \dots}_{\to 0 \quad \text{if} \quad h \to 0} \tag{SM4.1}$$

In practice, if h is small enough, floating-point arithmetic would ensure that only the first two terms remain (specifically, if $g(x)$ is indistinguishable from $g(x) - h^2\nabla^2 g(x)/2$ for the finite-precision machine, assuming the higher-order terms are even smaller).

This way, one can approximate the first derivatives to numerical precision, by taking the imaginary part of $g(x+ih)$, i.e.,

$$\nabla g(x) \approx \frac{\Im\big[g(x+ih)\big]}{h}, \tag{SM4.2}$$

where $\Im(x)$ is the imaginary part of x. In particular, because there is no difference involved and because the relative sizes of the real and imaginary parts do not interfere within the complex-number type, there is no need to worry about truncation error, so that one can take an arbitrarily small h (e.g., $h = 10^{-100}$).

The complex-step algorithm is straightforward to implement in many scientific computing languages, including Julia, MATLAB, Octave, Scilab, Python, IDL, and Fortran, which all have a built-in complex-number type. And we note that Martins et al. [SM13] suggested that complex-step differentiation is equivalent to algorithmic differentiation (also known as automatic differentiation).

SM4.2. Dual Numbers. Dual numbers were first introduced by Clifford [SM2], with the "dual" term being coined by Study [SM25]. A dual unit number, denoted ε, is introduced, and defined by the simple rule that $\varepsilon^2 = 0$ (instead of $i^2 = -1$ for complex numbers) and is used here as an efficient numerical tool to compute first derivatives (see, e.g., [SM17]).

Using dual numbers to evaluate a derivative is cleaner and simpler than the complex-step differentiation of (SM4.2), as

$$\nabla g(x) = \mathfrak{D}\big[g(x+\varepsilon)\big], \tag{SM4.3}$$

where $\mathfrak{D}(x)$ is the dual part (the ε part) of x. The advantage of using dual numbers for our approach lies in the fact that Taylor expansions of the first order in the non-real direction are exact, i.e.,

$$g(x+\varepsilon) = g(x) + \varepsilon\nabla g(x) + \underbrace{\frac{\varepsilon^2}{2}\nabla^2 g(x) + \dots}_{=0} \tag{SM4.4}$$

is a strict equality. (A property that is not true for Taylor expansions in the complex plane.) In other words, dual numbers provides a robust mathematical tool to represent infinitesimal quantities with ε.

Another advantage compared to the complex step algorithm is that there is no need to introduce a small step size, h, although we note that the behavior of dual numbers can be reproduced using complex numbers by choosing h sufficiently small for the complex Taylor series to automatically truncate terms of order greater than 1. However, we note that the dual-step algorithm offers a cleaner, more modern numerical differentiation tool that can be applied to a broader set of functions (e.g., functions that internally invoke complex numbers).

We note the dual numbers have a formal algebraic interpretation, whereby they extend the real numbers to an algebra over the reals of dimension 2, where ε is the image of the polynomial $X \mapsto X$ in the quotient of the univariate polynomial ring with real-valued coefficients, $\mathbb{R}[X]$, over the ideal generated by the $X \mapsto X^2$ polynomial. In other words, the dual numbers can be identified with $\mathbb{R}[X]/(X^2)$ just like the complex numbers can be identified with $\mathbb{R}[X]/(X^2+1)$.

SM4.3. Hyper Dual Numbers. Hyper dual numbers are an extension of dual numbers used to evaluate second derivatives (e.g., [SM8, SM6, SM7, SM17]). The nonreal components are defined by ε_1 and ε_2 such that $\varepsilon_1^2 = \varepsilon_2^2 = 0$ and $\varepsilon_1\varepsilon_2 \neq 0$. Consider $g(x, y)$, a real-valued function of two variables. Its Taylor expansion in the $(\varepsilon_1, \varepsilon_2)$ direction is given by

$$g(x+\varepsilon_1, y+\varepsilon_2) = g(x,y) + \varepsilon_1\nabla_x g(x,y) + \varepsilon_2\nabla_y g(x,y) + \varepsilon_1\varepsilon_2\nabla_{xy} g(x,y), \tag{SM4.5}$$

and its Taylor expansion in the $(\varepsilon_1\varepsilon_2, 0)$ direction is given by

$$g(x+\varepsilon_1, y+\varepsilon_2) = g(x,y) + \varepsilon_1\nabla_x g(x,y) + \varepsilon_2\nabla_x g(x,y) + \varepsilon_1\varepsilon_2\nabla_{xx} g(x,y), \tag{SM4.6}$$

Hence, the terms of the Hessian of g can be evaluated via

$$\nabla^2 g(x,y) = \mathfrak{H}\left(\begin{bmatrix} g(x+\varepsilon_1+\varepsilon_2, y) & g(x+\varepsilon_1, y+\varepsilon_2) \\ g(x+\varepsilon_2, y+\varepsilon_1) & g(x, y+\varepsilon_1+\varepsilon_2) \end{bmatrix}\right), \tag{SM4.7}$$

where $\mathfrak{H}(x)$ is the hyperdual part of x (the $\varepsilon_1\varepsilon_2$ coefficient). Note that hyperdual numbers can also be used to evaluate the terms of the gradient just like dual numbers, e.g., via

$$\nabla g(x,y) = \mathfrak{H}_1\left(\begin{bmatrix} g(x+\varepsilon_1, y) \\ g(x, y+\varepsilon_1) \end{bmatrix}^\mathsf{T}\right), \tag{SM4.8}$$

where $\mathfrak{H}_1(x)$ is the "first hyperdual" part of x (the ε_1 part). Generalizing these formulas to the case of functions of more than two variables is straightforward.

We note that the hyperdual numbers also have an algebraic interpretation. They form an algebra over the reals of dimension 4, as the quotient of the multivariate polynomial ring with real-valued coefficients, $\mathbb{R}[X, Y]$, over the ideal generated by the $(X, Y) \mapsto X^2$ and $(X, Y) \mapsto Y^2$ polynomials. In that quotient, ε_1 and ε_2 are the images of the polynomials $(X, Y) \mapsto X$ and $(X, Y) \mapsto Y$. In other words, the hyperdual numbers can be identified with $\mathbb{R}[X, Y]/(X^2, Y^2)$.

SM5. Solving dual and hyperdual linear systems. As noted in section 2 and section 3, the CSD, DUAL, and HYPER algorithms must invoke the inner solver

with complex-valued, dual-valued, and hyperdual-valued state and parameters. However, assuming the inner solver invokes Newton-type steps, it must solve complex, dual, or hyperdual linear systems, respectively. This Appendix describes an efficient strategy for solving dual-valued and hyperdual-valued linear systems, and illustrates its execution in Julia using packages that were developed specifically for this study: DualMatrixTools [SM18] and HyperDualMatrixTools [SM19].

At first, it would appear that in order to solve the linear systems involving a dual-valued matrix of the type $\mathbf{M} = \mathbf{A} + \varepsilon\mathbf{B}$, the factorization of the dual-valued matrix $\mathbf{M}$ is necessary. However, doing so is not straightforward and is likely not efficient: (i) there is no guarantee that the underlying package that performs factorizations can handle dual-valued sparse matrices (e.g., UMFPACK can only handle real and complex numbers), and (ii) performing dual-valued factorizations is computationally more expensive than performing real-valued factorizations. We note that this is also the case for solving complex-valued linear systems, and that a similar technique — avoiding complex-valued factorizations — can be used for solving complex-valued linear systems, as long as the imaginary part is small (as is the case for the complex-step differentiation method).

Instead, a better strategy is to only factorize $\mathbf{A}$, the real part of $\mathbf{M}$. Because we are using dual numbers, the Taylor expansion of the inverse of $\mathbf{A} + \varepsilon\mathbf{B}$ is given by

$$(\mathbf{A} + \varepsilon\mathbf{B})^{-1} = (\mathbf{I}_{\boldsymbol{x}} - \varepsilon\mathbf{A}^{-1}\mathbf{B})\mathbf{A}^{-1}, \tag{SM5.1}$$

where $\mathbf{I}_{\boldsymbol{x}}$ is the identity matrix (with its size, $n \times n$, indicated by the $\boldsymbol{x}$ superscript). Equation (SM5.1) allows to solve dual-valued linear systems using only the factors of the real part, $\mathbf{A}$, via two real-valued forward and back substitutions.

For the DUAL algorithm to work in our implementation, i.e., for the DUAL algorithm to be able to invoke the inner solver with dual-valued state and parameters, we developed DualMatrixTools [SM18], to perform the shortcut provided by (SM5.1) under the hood. That is, it allows to factorize the real part only of $\mathbf{M}$ (using the `factorize` function) and to solve the corresponding dual-valued linear system (using the backslash function, \) via (SM5.1) in a single line of code for each of those two operations. We note that this package can be used in other applications and, again, emphasize its ease of use.

In the case of the HYPER algorithms, we provide a hyperdual equivalent of (SM5.1), which shows that only the inverse of $\mathbf{A}$ is required for computing the inverse of a hyperdual-valued matrix $\mathbf{M} = \mathbf{A} + \varepsilon_1\mathbf{B} + \varepsilon_2\mathbf{C} + \varepsilon_1\varepsilon_2\mathbf{D}$. In fact, the inverse of $\mathbf{M}$ is given by

$$\mathbf{M}^{-1} = \left[\mathbf{I}_{\boldsymbol{x}} - \varepsilon_1\mathbf{A}^{-1}\mathbf{B} - \varepsilon_2\mathbf{A}^{-1}\mathbf{C} - \varepsilon_1\varepsilon_2\mathbf{A}^{-1}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{B}\mathbf{A}^{-1}\mathbf{C})\right]\mathbf{A}^{-1}. \tag{SM5.2}$$

Just like for dual-valued linear systems, we developed a Julia package, HyperDualMatrixTools [SM19], to solve hyperdual-valued systems under the hood. In a single line of code, one can factorize the real part of $\mathbf{M}$ using `factorize` or solve a hyperdual-valued linear system using backslash (\). This package has been optimized to compute hyperdual-valued forward and back substitutions in just four real-valued forward and back substitutions, as can be derived from (SM5.2).

SM6. The pitfalls of black-box approaches. In this appendix, we describe what happens when invoking the solver with complex, dual or hyperdual parameters as a black box. Here we show that not only is this naive approach slower, it directly introduces numerical errors. That is, we describe what happens when using the

DUAL, HYPER, and CSD algorithms. We emphasize that when using an iterative algorithm that was written for real-valued numbers, it is imperative to set additional tolerances on the non-real part(s) of the iterates. Setting these tolerances, which control when the loop terminates, can be complicated. But before diving in the details, we first recall Newton's method in the real case.

Starting from an initial estimate $\boldsymbol{x}_0$, Newton's method is based on the recursion relation

$$\boldsymbol{x}_{l+1} = \boldsymbol{x}_l - \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}_l, \boldsymbol{p})^{-1} \boldsymbol{F}(\boldsymbol{x}_l, \boldsymbol{p}), \tag{SM6.1}$$

where $\nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}_l, \boldsymbol{p})$ is the Jacobian evaluated at the lth iterate of the state, $\boldsymbol{x}_l$. If all goes well, the iterates converge to the steady state, i.e., $\lim_{l\to\infty} \boldsymbol{x}_l = \boldsymbol{s}(\boldsymbol{p})$.

In practice, when invoked for real-valued parameters, the solver we use applies Shamanskii's method, which improves on Newton's method by not updating the Jacobian at each iterate, increasing computational performance. Additionally, the solver we use also performs an Armijo line search if the Shamanskii step overshoots the solution (the solver we use was adapted from the `nsold` solver from Kelley [SM11]). That is, the Armijo line search looks for the minimum of the norm of $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p})$ along the Shamanskii-step direction via a quadratic approximation. However, for the sake of clarity, here we only detail the classical Newton's method. (The solver we actually use is available in the AIBECS package [SM20].)

We now explore the convergence of the Newton's method when the parameters are dual-valued (i.e., what happens when using the DUAL algorithm). We thus replace $\boldsymbol{p}$ with $\boldsymbol{p} + \varepsilon \boldsymbol{e}_j$. Although we start from the real-valued $\boldsymbol{x}_0 = \boldsymbol{s}(\boldsymbol{p})$, the iterates $\boldsymbol{x}_l$ include a potentially non-trivial dual part for $l > 0$, since the dual part of $\boldsymbol{p} + \varepsilon \boldsymbol{e}_j$ will spread to $\boldsymbol{x}_l$ at each Newton iteration. We thus partition $\boldsymbol{x}_l$ into its real and dual parts, such that

$$\boldsymbol{x}_l = \boldsymbol{a}_l + \varepsilon \boldsymbol{b}_l, \tag{SM6.2}$$

where we omit the $\boldsymbol{p} + \varepsilon \boldsymbol{e}_j$ argument for brevity.

The dual-valued $\boldsymbol{F}(\boldsymbol{a}_l + \varepsilon \boldsymbol{b}_l, \boldsymbol{p} + \varepsilon \boldsymbol{e}_j)$ can be expressed via the Taylor expansion of $\boldsymbol{F}$ at $(\boldsymbol{a}_l, \boldsymbol{p})$ in the $(\varepsilon \boldsymbol{b}_l, \varepsilon \boldsymbol{e}_j)$ direction, which gives

$$\boldsymbol{F}(\boldsymbol{a}_l + \varepsilon \boldsymbol{b}_l, \boldsymbol{p} + \varepsilon \boldsymbol{e}_j) = \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p}) + \varepsilon \mathbf{A}_l \boldsymbol{b}_{jk} + \varepsilon \nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p}) \boldsymbol{e}_j, \tag{SM6.3}$$

where $\mathbf{A}_l \equiv \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p})$ is a real-valued Jacobian matrix. Similarly, the dual-valued Jacobian matrix $\nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{a}_l + \varepsilon \boldsymbol{b}_l, \boldsymbol{p} + \varepsilon \boldsymbol{e}_j)$ can also be expressed by a dual Taylor expansion. However, we need not detail all its terms, and instead simply denote its real part by $\mathbf{A}_l$ and its dual part by the unspecified matrix $\mathbf{B}_l$, respectively, so that

$$\nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{a}_l + \varepsilon \boldsymbol{b}_l, \boldsymbol{p} + \varepsilon \boldsymbol{e}_j) = \mathbf{A}_l + \varepsilon \mathbf{B}_l. \tag{SM6.4}$$

Note that $\mathbf{A}_l = \nabla_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p})$ is also the real part of the dual-valued Jacobian.

Evaluating the iterations of (SM6.1) with dual-valued parameters, it appears that the real part has exactly the same form as the original real-valued iterations, except with $\boldsymbol{a}_l$ symbols in place of $\boldsymbol{x}_l$ symbols. Assuming we start with $\boldsymbol{a}_0 = \boldsymbol{s}(\boldsymbol{p})$ means $\boldsymbol{a}_l = \boldsymbol{s}(\boldsymbol{p})$ throughout. This is expected because the dual-valued call to the solver will not improve the real-valued solution.

The non-real part however contains additional terms, with

$$\begin{aligned} \boldsymbol{b}_{l+1} = \boldsymbol{b}_l &- \mathbf{A}_l^{-1} \mathbf{A}_l \boldsymbol{b}_l - \mathbf{A}_l^{-1} \nabla_{\boldsymbol{p}} \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p}) \boldsymbol{e}_j \\ &+ \mathbf{A}_l^{-1} \mathbf{B}_l \mathbf{A}_l^{-1} \boldsymbol{F}(\boldsymbol{a}_l, \boldsymbol{p}), \end{aligned} \tag{SM6.5}$$

where we have used the analytical formula for the inverse of a dual-valued matrix, (SM5.1). Note the first and second terms, which should cancel each other. These were intentionally left for the reader to appreciate how a black-box approach will execute unnecessary computations, both slowing down the computation and potentially introducing additional numerical errors ($\boldsymbol{b}_l$ would seldom be strictly exactly equal to $\mathbf{A}_l^{-1}\mathbf{A}_l\boldsymbol{b}_l$ without an infinite precision computer.) Additionally, the last term should vanish because $\boldsymbol{a}_l = \boldsymbol{s}(\boldsymbol{p})$ (by definition of $\boldsymbol{s}(\boldsymbol{p})$). However, this term will not vanish exactly with finite precision machines, so that the black-box approach will compute this term and potentially introduce errors while doing so.

In our implementation, the tolerance for the dual part is decided by the ratio of the norm of the dual-part of the Newton step to the norm of the dual part. Specifically, the solver loop terminates if $\|\boldsymbol{b}_l - \boldsymbol{b}_{l-1}\|/\|\boldsymbol{b}_{l-1}\|$ is smaller than 10^{-7}. Starting from the solution $\boldsymbol{s}(\boldsymbol{p})$, the dual-type black-box approaches will thus invoke the inner solver with one iteration to converge to the dual-valued solution, and another iteration to check that the dual-part has converged, thus requiring two iterations overall.

We note that in this specific context, one could get away with forgetting to set a tolerance for the dual part only because the inner solver effectively converges in a single iteration. However, this is not the case for all iterative algorithms taken as a black box, as illustrated by the ∼100-iterations lag of the non-real part described by Martins et al. [SM13]. Therefore we consider that omitting these tolerances is *not* a reasonable approach. Conversely, we also note that setting too tight a tolerance for the non-real part could potentially result in a large (or even infinite) number of iterations for the solver loop to terminate, when in fact it oscillates over a small neighborhood of the solution until a small enough dual-valued step is randomly taken. We emphasize that all the arguments brought up in this appendix so far also apply to the complex-step algorithm. That is, the exact same phenomenon occurs when using the CSD algorithm. In comparison, the F-1 algorithm completely alleviates the need to set a tolerance for the dual part of the inner solver, since it does not invoke it.

We now show that, in our context, the inner solver invoked with hyperdual-valued parameters converges in exactly two iterations (i.e., when using the HYPER algorithm). We give a brief outline of how the derivation goes, but leave its details out for brevity. Again, we consider the hyperdual parameters defined by $\boldsymbol{p}_{jk} = \boldsymbol{p} + \varepsilon_1\boldsymbol{e}_j + \varepsilon_2\boldsymbol{e}_k$. Starting from the real-valued solution, $\boldsymbol{s}(\boldsymbol{p})$, just like in the dual case, the real part of the hyperdual state remains constant, equal to $\boldsymbol{s}(\boldsymbol{p})$, throughout the Newton iterations. The first iteration permeates hyperdual-values from the parameters into the first state iterate, $\boldsymbol{x}_1$. In fact, the ε_1 and ε_2 parts of the state converge in that first iteration, so that the ε_1 and ε_2 parts of $\boldsymbol{s}(\boldsymbol{p}_{jk})$ are the ε_1 and ε_2 parts of $\boldsymbol{x}_1$. However, a second iteration is required for the $\varepsilon_1\varepsilon_2$ part to converge, because it requires that the previous state (in this case $\boldsymbol{x}_1$) contains the converged ε_1 and ε_2 parts. Thus, convergence of all the non-real parts requires exactly two iterations. This is because, unlike for the dual case, some of the non-real terms that include non-real parts of the state (the terms including $\boldsymbol{b}_l$ in (SM6.5)) do not cancel out in the hyperdual case. This is also because the $\varepsilon_1\varepsilon_2$ part of $\boldsymbol{x}_{l+1}$ only depends on the real, ε_1, and ε_1 parts of $\boldsymbol{x}_l$. Just like in the dual case, we emphasize that the inner solver must check that the non-real parts have converged, so that the HYPER algorithm requires one additional Newton step to terminate, bringing the total number of iterations to three.

Code and data availability. The state function, $\boldsymbol{F}$, and the mismatch function, f, as well as their derivatives, were automatically generated using the AIBECS package [SM20] or manually generated in the FastBGCParameterOptimization GitHub

repository. The ocean circulation was taken from the output of the control (CTL) run of the Ocean Circulation Inverse Model (OCIM1 [SM4]) and is available online in a MAT file (the MATLAB data format) on the website of Tim DeVries. The OCIM1 grid and transport matrix have been converted to Julia to be downloaded and used by the AIBECS package [SM20]. The mismatch of the state used the objectively analyzed mean field of the phosphate concentration from the World Ocean Atlas 2018 (WOA18 [SM9]). The WOA18 data was downloaded and fitted to the OCIM grid using the WorldOceanAtlasTools package [SM22]. The F-1 algorithm is implemented in the F1Method package [SM21]. The CSD, FD1, DUAL, HYPER, and FD2 algorithms, the figures, and the underlying data that are shown in this study are contained in the FastBGCParameterOptimization GitHub repository. Dual- and hyperdual-valued factorizations and forward and back substitutions of sparse linear systems were performed using the DualMatrixTools [SM18] and HyperDualMatrixTools [SM19] packages, respectively. The optimizations were performed using the Optim package [SM14, SM15]. The concrete types defining the steady-state problems and the steady-state solutions were built upon the DiffEqBase package [SM23]. Benchmarks were performed with the BenchmarkTools and TimerOutputs packages. The diagram was created using the LaTeXTikZ package [SM26]. Result figures were created using the PyPlot package. All the computations were performed using the Julia language [SM1].

REFERENCES

[SM1] 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.

[SM2] W. Clifford, *Preliminary sketch of bi-quaternions*, Proceedings of London Mathematical Society, 4 (1873), pp. 361–395.

[SM3] 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.

[SM4] T. DeVries, J.-H. Liang, and C. Deutsch, *A mechanistic particle flux model applied to the oceanic phosphorus cycle*, Biogeosciences, 11 (2014), pp. 5381–5398, https://doi.org/10.5194/bg-11-5381-2014.

[SM5] 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.

[SM6] 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.

[SM7] 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.

[SM8] 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.

[SM9] 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.

[SM10] A. H. Gebremedhin, F. Manne, and A. Pothen, *What color is your jacobian? graph coloring for computing derivatives*, SIAM review, 47 (2005), pp. 629–705.

[SM11] 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.
[SM12] J. N. Lyness and C. B. Moler, *Numerical differentiation of analytic functions*, SIAM Journal on Numerical Analysis, 4 (1967), pp. 202–210, https://doi.org/10.1137/0704019.
[SM13] 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.
[SM14] 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.
[SM15] 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.
[SM16] C. Moler, *Cleve's Corner: Cleve Moler on Mathematics and Computing — Complex Step Differentiation*, 2013, https://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation (accessed 2019-04-29).
[SM17] 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.
[SM18] B. Pasquier, *DualMatrixTools: A Julia package for solving dual-valued linear systems*, Nov. 2018, https://doi.org/10.5281/zenodo.1493571.
[SM19] B. Pasquier, *HyperDualMatrixTools: A Julia package for solving hyperdual-valued linear systems*, Nov. 2018, https://doi.org/10.5281/zenodo.1670841.
[SM20] B. Pasquier, *AIBECS.jl: The ideal tool for exploring global marine biogeochemical cycles*, May 2019, https://doi.org/10.5281/zenodo.2864051.
[SM21] 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.
[SM22] B. Pasquier, *Worldoceanatlastools.jl: Download and process world ocean data in julia*, May 2019, https://doi.org/10.5281/zenodo.2677666.
[SM23] 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.
[SM24] W. Squire and G. Trapp, *Using complex variables to estimate derivatives of real functions*, SIAM Review, 40 (1998), pp. 110–112, https://doi.org/10.1137/S003614459631241X.
[SM25] E. Study, *Geometrie der dynamen: Die zusammensetzung von kräften und verwandte gegenstände der geometrie*, 1903.
[SM26] T. Tantau, *The TikZ and PGF Packages*, 2013, http://sourceforge.net/projects/pgf/.