High-Order Automatic Differentiation of Unmodified Linear Algebra Routines via Nilpotent Matrices

by

Benjamin Z Dunham

B.A., Carroll College, 2009

M.S., University of Colorado, 2011

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Aerospace Engineering Sciences

2017

This thesis entitled:
High-Order Automatic Differentiation of Unmodified Linear Algebra Routines via Nilpotent Matrices
written by Benjamin Z Dunham
has been approved for the Department of Aerospace Engineering Sciences

Prof. Kurt K. Maute

Prof. Alireza Doostan

Date

Dunham, Benjamin Z (Ph.D., Aerospace Engineering Sciences)

High-Order Automatic Differentiation of Unmodified Linear Algebra Routines via Nilpotent Matrices

Thesis directed by Prof. Kurt K. Maute

This work presents a new automatic differentiation method, Nilpotent Matrix Differentiation (NMD), capable of propagating any order of mixed or univariate derivative through common linear algebra functions – most notably third-party sparse solvers and decomposition routines, in addition to basic matrix arithmetic operations and power series – without changing data-type or modifying code line by line; this allows differentiation across sequences of arbitrarily many such functions with minimal implementation effort. NMD works by enlarging the matrices and vectors passed to the routines, replacing each original scalar with a matrix block augmented by derivative data; these blocks are constructed with special sparsity structures, termed “stencils,” each designed to be isomorphic to a particular multidimensional hypercomplex algebra. The algebras are in turn designed such that Taylor expansions of hypercomplex function evaluations are finite in length and thus exactly track derivatives without approximation error.

Although this use of the method in the “forward mode” is unique in its own right, it is also possible to apply it to existing implementations of the (first-order) discrete adjoint method to find high-order derivatives with lowered cost complexity; for example, for a problem with N inputs and an adjoint solver whose cost is independent of N – i.e., $\mathcal{O}(1)$ – the $N \times N$ Hessian can be found in $\mathcal{O}(N)$ time, which is comparable to existing second-order adjoint methods that require far more problem-specific implementation effort. Higher derivatives are likewise less expensive – e.g., a $N \times N \times N$ rank-three tensor can be found in $\mathcal{O}\left(N^2\right)$. Alternatively, a Hessian-vector product can be found in $\mathcal{O}(1)$ time, which may open up many matrix-based simulations to a range of existing optimization or surrogate modeling approaches. As a final corollary in parallel to the NMD-adjoint hybrid method, the existing complex-step differentiation (CD) technique is also shown to be capable of finding the Hessian-vector product. All variants are implemented on a stochastic diffusion problem and compared in-depth with various cost and accuracy metrics.

Contents

Figures

Part I

Motivation and Background

Chapter 1

Overview

In broad terms, this dissertation concerns itself with the automatic differentiation of medium- to large-scale engineering codes – that is, given an existing program, the goal herein is to compute derivatives of simulation output(s) with respect to system parameters, and to do it with an absolute minimum of work on the part of the code developer. Typical expected usage cases include those where the cost of human labor required to implement the differentiation routines is likely to swamp the actual run cost; those where different program runs will need different orders or different numbers of derivatives, preferably without major code re-writes for each case; and those where the precision of the common finite difference method is simply too limited.

More specifically, this work focuses on arbitrary-order derivatives propagated through low-level matrix algorithms implemented in third-party packages, with the most attention being paid to matrix arithmetic and linear algebra solvers. Matrix routines deserve special consideration for the simple fact that, despite their wide use, they are not amenable to most existing automatic differentiation (AD) techniques. This is because their back-end executables and source code can rarely be modified directly; this puts them in conflict with almost all standard AD packages (e.g., [1–7]), which require recompilation after either a change in data-type or line-by-line code modification.

1.1 Contributions

To address such difficulties, this work presents a new method, termed “Nilpotent Matrix Differentiation” (NMD), to semi-automatically find precise, any-order derivatives of unaltered third-party linear algebra routines like those included in BLAS/LAPACK or MATLAB; “any-order,” here, means the ability to find simple first derivatives, mixed second derivatives (i.e., entries of the Hessian matrix), or even third, fourth, or N^{th} mixed derivatives[1]. Hybrid modifications are also introduced for existing first-order differentiation approaches – namely, the popular discrete adjoint method and the complex-step differentiation (CD) technique – which quickly extend them to higher orders.

The key insight to the new method is that very specific matrix sparsity structures, designed to be isomorphic to certain non-real scalar algebras, can both store derivative data and compute the chain rule with most matrix operations. Before entering any sequence of linear algebra routines, input matrices and vectors are mapped to new, larger matrices that introduce derivative data in the proper structures; these will then self-propagate through successive matrix operations with little or no further code modification on the part of the user[2] . After all such operations are complete, derivative information can easily be extracted from the resulting matrices or solution vectors. Crucially, the new information provided by NMD will be just as precise and accurate[3] as the output of the original program. Compare this to the typical first-approximation tool of choice, the finite difference (FD) method, which might find only a few significant figures or be entirely incorrect for higher orders.

Most real-world usage examples in this document are only summarized, and the derivation of NMD itself is largely abstract. To give the reader a conceptual framework to work with, the mechanics of the approach will first be introduced with a stylized example of how it might be used. The underlying theory will be saved for Part II.

1.2 A Motivating Example

Consider a case where a user wishes to quickly interface his or her code with a black-box optimization suite (e.g., [8–10]), and to this end wishes to use automatic differentiation to provide the suite with second derivatives of the code. The execution order of a generic, undifferentiated user program is shown in Figure 1.1(a). Such a program loosely proceeds in the following way: first, user-written code performs any initialization computation, then discretizes

[1] Various uses of second derivatives will be discussed throughout Chapter 2, and the use of third or higher derivatives will reviewed in Section 2.4.

[2] In this context, "user" refers to the coder who must call linear algebra routines, not necessarily a user of the final program. The phrase "code developer" could be used interchangeably.

[3] Throughout this work, results from NMD or other methods will at times be referred to as "exact". The word is reserved for situations where a computed value *is* the derivative (at least as it would otherwise be given by a literal formula), rather than an approximation thereof; however, it should formally be taken to mean "exact to within the limitations of computer architecture (machine epsilon) or of the original matrix algorithm being modified".

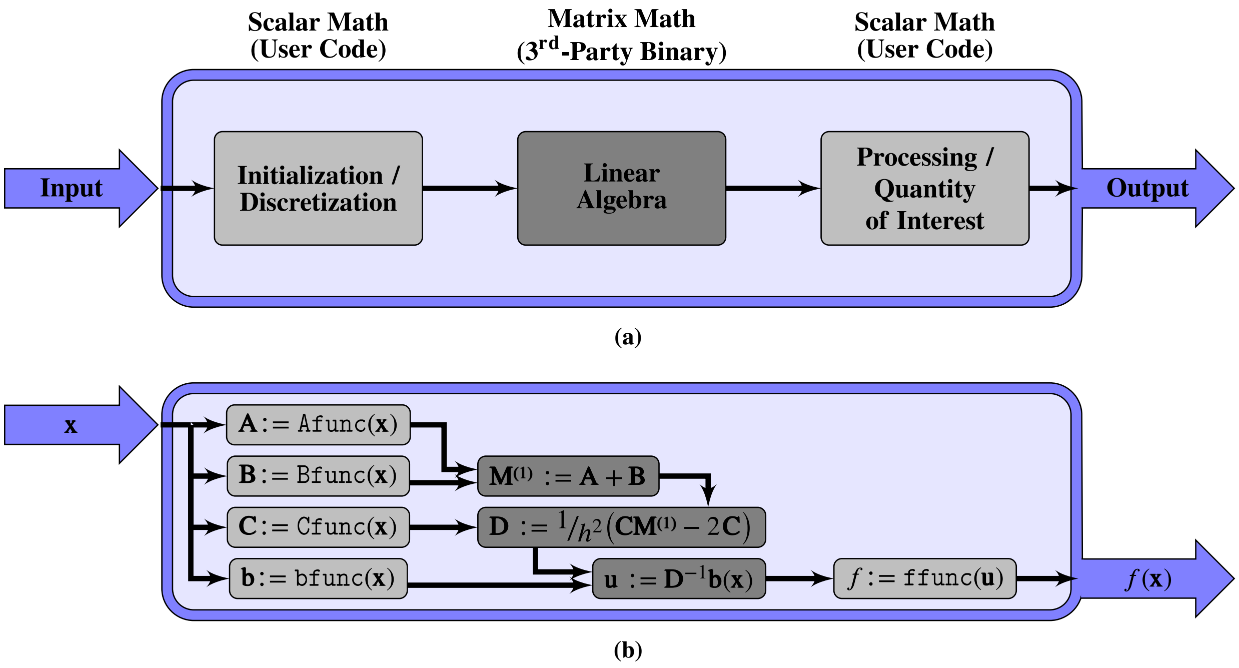

Scalar Math
(User Code)
Matrix Math
(3rd-Party Binary)
Scalar Math
(User Code)
Input
Initialization / Discretization
Linear Algebra
Processing / Quantity of Interest
Output
(a)
x
$\mathbf{A} := \texttt{Afunc}(\mathbf{x})$
$\mathbf{B} := \texttt{Bfunc}(\mathbf{x})$
$\mathbf{C} := \texttt{Cfunc}(\mathbf{x})$
$\mathbf{b} := \texttt{bfunc}(\mathbf{x})$
$\mathbf{M}^{(1)} := \mathbf{A} + \mathbf{B}$
$\mathbf{D} := {}^{1}/_{h^2}(\mathbf{C}\mathbf{M}^{(1)} - 2\mathbf{C})$
$\mathbf{u} := \mathbf{D}^{-1}\mathbf{b}(\mathbf{x})$
$f := \texttt{ffunc}(\mathbf{u})$
$f(\mathbf{x})$
(b)

Figure 1.1: Data-flow of (a) a generic program and (b) the example specified by Equation 1.1 presented with an intermediate calculation of the matrix $\mathbf{M}^{(1)}$ (which makes the construction of $\mathbf{D}$ compatible with basic BLAS operations). User source code (light gray) computes scalar (non-matrix) operations; it is accessible to the code developer and can be modified. The linear algebra routines (dark gray), in contrast, either come in the form of precompiled binaries or else are simply too large to modify.

a partial differential equation to build a matrix representation of the problem. Second, the program calls a series of third-party linear algebra routines (e.g., in BLAS/LAPACK, MATLAB, or any of the many other numerical computing packages); these are assumed to be inaccessible to traditional AD approaches. (This is generally the case unless the user designed the program from the very start around a specialized matrix library meant for AD.) Third and finally, the final matrix or vector result is examined by more user-written code, which performs any post-processing and outputs a quantity of interest.

For the sake of argument, assume this specific user program solves only the following[4] simple problem:

[4] Chapter 8 will feature a much more elaborate program which includes some very similar operations.

$$\mathbf{D} = \left(^{1}/_{h^2}\right)\Big(\mathbf{C}(\mathbf{x})\left(\mathbf{A}(\mathbf{x}) + \mathbf{B}(\mathbf{x})\right) - 2\mathbf{C}(\mathbf{x})\Big) \tag{1.1}$$

$$\mathbf{u} = \mathbf{D}^{-1}\mathbf{b}(\mathbf{x}) \tag{1.2}$$

where h is a constant (undifferentiated) real scalar; $\mathbf{x}$ is the set of input parameters to be optimized; the matrics $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ and the vector $\mathbf{b}$ are user-coded functions of $\mathbf{x}$; and the final program output $f(\mathbf{x}) = f(\mathbf{u}(\mathbf{x}))$ is a user-coded function of $\mathbf{u}$. Note that $\mathbf{u}$ can be considered a vector of internal state-variables, e.g., for an adjoint formulation as discussed in Section 3.1.2. The data-flow of this specific example is visualized in Figure 1.1(b).

Assume the black-box optimization suite implements one of the many second-order optimization methods (some of which will be further discussed in Section 2.1) that utilize both the function gradient $\mathbf{g}(\mathbf{x}) \equiv \nabla f(\mathbf{x})$ and second derivative information contained in the Hessian matrix $\mathbf{H}(\mathbf{x}) \equiv \nabla^2 f(\mathbf{x})$. In particular, assume it requires the nonlinear conjugate gradient (NCG) update coefficients that will be presented in Equations 2.11 and 2.13 (pages 17 and 18), reprinted here for convenience:

$$\alpha = -\frac{\mathbf{g}^T\mathbf{d}}{\mathbf{d}^T\mathbf{H}\mathbf{d}} \qquad\qquad \beta = \frac{\mathbf{g}^T\mathbf{H}\mathbf{d} - \mathbf{g}^T\mathbf{d}}{\mathbf{d}^T\mathbf{H}\mathbf{d}} \tag{1.3}$$

where $\mathbf{d}$ is the search direction and α and β are used to move along $\mathbf{d}$ and update it, respectively.

1.3 Differentiation Approaches

As with all NCG variants, $\mathbf{g}$ must be provided by the user; presumably, this is done with a manual implementation of the common adjoint method (Section 3.1.2), which in many cases can compute the entire N-input gradient with a cost complexity of $\mathcal{O}(1)$ – that is, independent of N and proportional only to the cost of the original program. Second derivatives, however, are more difficult to compute. Second-order adjoint methods (Section 3.1.3) with complexity $\mathcal{O}(N)$ exist to build the Hessian; these are, however, application-specific and far more time-consuming for most users to implement. Meanwhile, standard forward-mode (non-adjoint) automatic methods (matrix-compatible or not), while less difficult to apply, find $\mathbf{H}$ in $\mathcal{O}\left(N^2\right)$ time; for systems with large N, this can be impractically expensive, even when compared to human labor.

However, it is important to realize that the full Hessian is not always required – instead, algorithms often (e.g., [11–17]) only call for the *product* of the matrix with one or two vectors. This becomes more than a semantic difference if such a product can easily be found directly, without ever building[5] the full $\mathbf{H}$. (Admittedly, this glosses over several difficulties – many methods will require a pre-conditioner, or else an optimization algorithm that includes a detection and solution strategy for negative curvature. [18]) To this end, Section 3.1.4 will discuss an existing manual approach [19] for scalar codes – originally developed for neural network backpropagation algorithms, and meant to be applied line-by-line – which can find a Hessian-vector product $\mathbf{Hv}$ with the same cost complexity as the gradient $\mathbf{g}$. In Section 7.3, this dissertation will demonstrate that a combination of the discrete adjoint method with a single-derivative instance of NMD can find the same product, for very little human effort.[6] The same will be shown for the combination of CD with the adjoint method, which has never been demonstrated before and hence constitutes another important corollary contribution. However, implementing these combinations is in fact so straightforward that they would not constitute an adequate demonstration for this chapter. Consider instead the equally-overlooked (if not more-so) vector-Hessian-vector product – i.e., for the problem given in Equation 1.3, $\mathbf{g}^T\mathbf{Hd}$ and $\mathbf{d}^T\mathbf{Hd}$. These can be found by computing the directional derivatives[7] of $f(\mathbf{x} + r_1\mathbf{d} + r_2\mathbf{g})\,|_{r_1=r_2=0}$ with respect to r_1 and r_2, without any modification of the adjoint method. (Note that products of two derivatives cannot be found directly with directional evaluations; here, $\mathbf{g}$ is pre-computed.)

As the focus of this dissertation is limited to derivatives across the matrix routines (the dark gray steps in Figure 1.1), it is assumed a standard AD technique is applied to the user code at each of the other steps, with the real NMD work happening at the data hand-offs. Although any number of derivatives are possible with these tools,

[5] This can be intuitively understood by drawing a parallel with linear solvers, which find $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ but never build the full matrix inverse $\mathbf{A}^{-1}$.

[6] Further, should the full $\mathbf{H}$ truly be needed, this hybrid approach can easily build it column by column, with the same $\mathcal{O}(N)$ complexity as the much more difficult to implement second-order adjoint methods.

[7] Note that such non-adjoint directional values are not unique to NMD and could be found with AD for purely user-written, non-matrix code (although this fact, too, is often overlooked in engineering literature, e.g., [11,12]). Instead, NMD's role is to carry the derivative information through linear algebra operations. Directional differentiation is chosen as a demonstration problem to emphasize that even one or two second derivatives can be of great use, and thus motivate methods like NMD that can quickly make such quantities available for any matrix problem. However, note that the most common discrete adjoint method is itself a matrix problem – typically, without NMD, $\mathbf{Hd}$ can only be found if a scalar-only method is available to find the gradient.

for this theoretical example it is taken as a given that the user code is altered such that every scalar variable (i.e., the input variables and any temporary non-matrix variables created within `Afunc`, `Bfunc`, `Cfunc`, `bfunc` or `ffunc`) is represented with an operator-overloaded data type capable of tracking two first derivatives (that is, $\partial/\partial r_1$ and $\partial/\partial r_2$), one univariate second derivative ($\partial^2/\partial r_1^2$) and one mixed second derivative ($\partial^2/\partial r_1 \partial r_2$). Any AD type that supports sparse Hessian calculation (in this case, treated as a two-variable matrix regardless of the actual N) would be sufficient for this task; if nothing else, one could even nest a first-derivative type within itself to find derivatives of derivatives. Operator-overloaded types, however, cannot be used inside most matrix math routines.

For such a simple example with a small number of matrix operations and only a few derivatives, it is of course possible to apply manual derivative formulas (e.g., [20]) every step of the way; in the case of the second derivatives, these could be applied twice. However, for larger codes it would be preferable to automate the process as much as possible, ideally without altering each intermediate operation. Assuming the code is real-valued and complex variants of the matrix routines are available, two runs of the complex-step method could find the first derivatives without intermediate additions to the code; CD, however, cannot find second derivatives (at least not without an adjoint hybrid, which is saved for Section 7.3). This is where the NMD method comes into play.

1.4 An Example of Nilpotent Matrix Differentiation

Both NMD and many forms of AD analytically (but not computationally) represent operator-overloaded calculations by adding non-real variables to the real-valued program input $\mathbf{x}$ – for example, to find derivatives with respect to x_j and x_k, one could compute $f(\mathbf{x} + \mathbf{D}\hat{\mathbf{e}})$, where the vector $\hat{\mathbf{e}}$ represents a set of N non-real basis elements and $D_{jj} = D_{kk} = 1$ but the rest of the matrix $\mathbf{D}$ is zero. In the case of this specific example, though, the AD-augmented run of the user program is treated analytically as $f(\mathbf{x}_i + \mathbf{d}_i\varepsilon_1 + \mathbf{g}_i\varepsilon_2)$, where the basis elements ε_1 and ε_2 are commutative under multiplication and have the following nilpotent properties:

$$\varepsilon_1{}^3 = 0 \quad \text{yet} \quad \varepsilon_1 \neq 0 \quad \text{and} \quad \varepsilon_1^2 \neq 0 \tag{1.4a}$$

$$\varepsilon_2{}^2 = 0 \quad \text{yet} \quad \varepsilon_2 \neq 0 \tag{1.4b}$$

$$\varepsilon_1{}^2\varepsilon_2 = 0 \quad \text{yet} \quad \varepsilon_1\varepsilon_2 \neq 0 \tag{1.4c}$$

Given this algebra, any (arbitrary) scalar z from the original program can be replaced with:

$$\hat{z} = z + z_1\varepsilon_1 + z_2\varepsilon_2 + z_{11}\varepsilon_1{}^2 + z_{12}\varepsilon_1\varepsilon_2 \tag{1.5}$$

As will be explained in Chapter 5, when the program is properly initialized, the real-valued constants z_1, z_2, z_{11}, and z_{12} can in fact be interpreted as

$$z_1 = {}^{\partial z}\!/_{\partial r_1} \qquad z_2 = {}^{\partial z}\!/_{\partial r_2} \tag{1.6}$$

$$z_{11} = \frac{1}{2}{}^{\partial^2 z}\!/_{\partial r_1{}^2} \qquad z_{12} = {}^{\partial^2 z}\!/_{\partial r_1 \partial r_2} \tag{1.7}$$

As will further be explained in Chapter 6, any z can alternatively be replaced with:

$$\check{\mathbf{Z}} = \mathcal{DM}(z) = z\mathbf{I} + \left({}^{\partial z}\!/_{\partial r_1}\right)\mathbf{E}^{(1)} + \left({}^{\partial z}\!/_{\partial r_2}\right)\mathbf{E}^{(2)} + \frac{1}{2}\left({}^{\partial^2 z}\!/_{\partial r_1{}^2}\right)\mathbf{E}^{(1)2} + \left({}^{\partial^2 z}\!/_{\partial r_1 \partial r_2}\right)\mathbf{E}^{(1)}\mathbf{E}^{(2)} \tag{1.8}$$

where $\mathbf{I}$ is the 5×5 identity matrix and $\mathbf{E}^{(1)}$ and $\mathbf{E}^{(2)}$ are nilpotent matrices constructed specifically (and automatically) for this problem:

$$\mathbf{E}^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{E}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.9}$$

This defines the operator:

$$\mathcal{DM}(z) = \begin{bmatrix} z & 0 & \partial z/\partial r_1 & \partial z/\partial r_2 & \partial^2 z/\partial r_1 \partial r_2 \\ 0 & z & 0 & 2\left(\partial z/\partial r_1\right) & \partial^2 z/\partial {r_1}^2 \\ 0 & 0 & z & 0 & \partial z/\partial r_2 \\ 0 & 0 & 0 & z & \partial z/\partial r_1 \\ 0 & 0 & 0 & 0 & z \end{bmatrix} \tag{1.10}$$

The pattern defined by a matrix operator like Equation 1.10 will herein be referred to as a "stencil," as it must be applied identically to every entry in every matrix; the derivatives at a given position in the stencil must always be taken with respect to the same program input. When building this representation, the derivatives would be supplied by the aforementioned standard AD technique. Application of the operator to an entire matrix is performed blockwise; for example, if the matrix $\mathbf{A}$ in the demonstration problem from Equation 1.1 were a trivial 2×2 matrix, it would be redefined as:

$$\check{\mathbf{A}} := \mathcal{DM}(\mathbf{A}) \tag{1.11}$$

$$= \mathcal{DM}\left(\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \right) \tag{1.12}$$

$$= \begin{bmatrix} A_{11} & 0 & \partial A_{11}/\partial r_1 & \partial A_{11}/\partial r_2 & \partial^2 A_{11}/\partial r_1 \partial r_2 & A_{12} & 0 & \partial A_{12}/\partial r_1 & \partial A_{12}/\partial r_2 & \partial^2 A_{12}/\partial r_1 \partial r_2 \\ 0 & A_{11} & 0 & 2\left(\partial A_{11}/\partial r_1\right) & \partial^2 A_{11}/\partial {r_1}^2 & 0 & A_{12} & 0 & 2\left(\partial A_{12}/\partial r_1\right) & \partial^2 A_{12}/\partial {r_1}^2 \\ 0 & 0 & A_{11} & 0 & \partial A_{11}/\partial r_2 & 0 & 0 & A_{12} & 0 & \partial A_{12}/\partial r_2 \\ 0 & 0 & 0 & A_{11} & \partial A_{11}/\partial r_1 & 0 & 0 & 0 & A_{12} & \partial A_{12}/\partial r_1 \\ 0 & 0 & 0 & 0 & A_{11} & 0 & 0 & 0 & 0 & A_{12} \\ A_{21} & 0 & \partial A_{21}/\partial r_1 & \partial A_{21}/\partial r_2 & \partial^2 A_{21}/\partial r_1 \partial r_2 & A_{22} & 0 & \partial A_{22}/\partial r_1 & \partial A_{22}/\partial r_2 & \partial^2 A_{22}/\partial r_1 \partial r_2 \\ 0 & A_{21} & 0 & 2\left(\partial A_{21}/\partial r_1\right) & \partial^2 A_{21}/\partial {r_1}^2 & 0 & A_{22} & 0 & 2\left(\partial A_{22}/\partial r_1\right) & \partial^2 A_{22}/\partial {r_1}^2 \\ 0 & 0 & A_{21} & 0 & \partial A_{21}/\partial r_2 & 0 & 0 & A_{22} & 0 & \partial A_{22}/\partial r_2 \\ 0 & 0 & 0 & A_{21} & \partial A_{21}/\partial r_1 & 0 & 0 & 0 & A_{22} & \partial A_{22}/\partial r_1 \\ 0 & 0 & 0 & 0 & A_{21} & 0 & 0 & 0 & 0 & A_{22} \end{bmatrix} \tag{1.13}$$

and $\check{\mathbf{B}}$ and $\check{\mathbf{C}}$ would be redefined in an identical manner. In the case of the vector $\check{\mathbf{b}}$, as the rightmost column of the stencil contains a copy of every derivative and as matrix multiplication is performed column-wise for the right multiplicand, a single-column structure can be preserved by dropping the other columns:

$$\check{\mathbf{b}} := \mathcal{DV}(\mathbf{b}) = \mathcal{DV}\left(\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) \tag{1.14}$$

$$= \left[\ {}^{\partial^2 b_1}/_{\partial r_1 \partial r_2} \quad {}^{\partial^2 b_1}/_{\partial r_1{}^2} \quad {}^{\partial b_1}/_{\partial r_2} \quad {}^{\partial b_1}/_{\partial r_1} \quad b_1 \quad {}^{\partial^2 b_2}/_{\partial r_1 \partial r_2} \quad {}^{\partial^2 b_2}/_{\partial r_1{}^2} \quad {}^{\partial b_2}/_{\partial r_2} \quad {}^{\partial b_2}/_{\partial r_1} \quad b_2\ \right]^T \tag{1.15}$$

Some readers may object that the expanded matrix appears much larger than the original; however, note that while the number of non-zero entries in the matrix has grown by a factor of 12, the number of computed values (reals plus derivatives) has likewise grown by a factor of 5, for a per-value growth factor of 2.4. Compare this to the product and chain rules for ordinary scalar derivatives, which also require a disproportionate number of additional calculations to differentiate individual operations.

Importantly, because the pattern shown in Equations 1.13 and 1.15 repeats itself after matrix addition and multiplication, $\check{\mathbf{M}}^{(1)}$ and $\check{\mathbf{D}}$ will automatically compute the derivatives ${}^{\partial M^{(1)}_{ij}}/_{\partial \mathbf{r}}$ and ${}^{\partial D_{ij}}/_{\partial \mathbf{r}}$, keeping them in the same stencil structure without any work on the part of the user; likewise, the result $\mathbf{u}$ will carry ${}^{\partial u_i}/_{\partial \mathbf{r}}$, which can be passed back to the ordinary AD types for any remaining scalar calculations. This is a small example problem with only a few operations, but in general the derivative calculation will propagate through arbitrarily many matrix operations, e.g., for coupled systems updated in a loop. With certain exceptions – e.g., the transpose operation, which (as will be discussed in Section 6.2) only needs a small alteration – the work done by NMD is limited to matrix creation and analysis, thus satisfying the goal of an automated process.

As for the work done during matrix creation, assuming the user defines the non-zero values of his or her matrices using row/column/value $\{i, j, v\}$ triplets, standard AD will provide operator-overloaded variants of v that normally could not be assembled into a matrix; here, NMD can simply provide a single function to map from an original set of one or more triplets to an extended set. For example, in MATLAB, this is a one-line addition to the matrix creation routine:

```
[Ai2_vec, Aj2_vec, Aval2_vec] = AD2Stencil(Ai1, Aj1, Aval1_AD);
```

Here, the RHS inputs constitute an AD-overloaded triplet[8] $\{i, j, A_{ij}\}$ of the sparse matrix $\mathbf{A}$ and the LHS outputs are real-valued vectors (a set of triplets) that define the stencil $\mathcal{DM}\left(A_{ij}\right)$. Likewise, for analysis of the results, NMD can provide a function that takes an index i corresponding to a scalar in the original vector $\mathbf{u}$, examines the stencil-expanded variant $\breve{\mathbf{u}}$, and returns an AD type:

```
uval1_AD = Stencil2AD(u_NMD, ui1);
```

Once this second hand-off is complete, the overloaded type can propagate all derivatives to the final program output where they can be analyzed. With the operator-overloaded AD and NMD both fully implemented, the program can be represented schematically as shown in Figure 1.2.

After the program has been modified in these ways, it can recover each of the derivatives from Equation 1.8; for the example problem, the directional second derivatives can then be interpreted as the requested products:

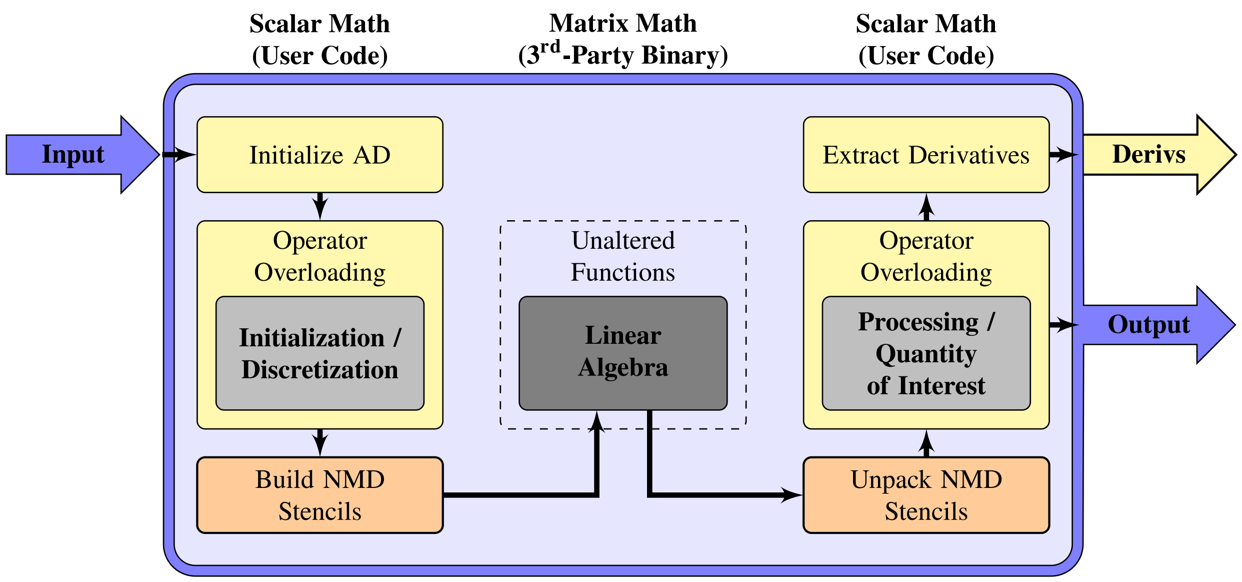
$$\partial^2 f / \partial {r_1}^2 = \mathbf{d}^T \mathbf{H} \mathbf{d} \qquad\qquad \partial^2 f / \partial r_1 \partial r_2 = \mathbf{g}^T \mathbf{H} \mathbf{d} \tag{1.16}$$

1.5 Dissertation Goals and Organization

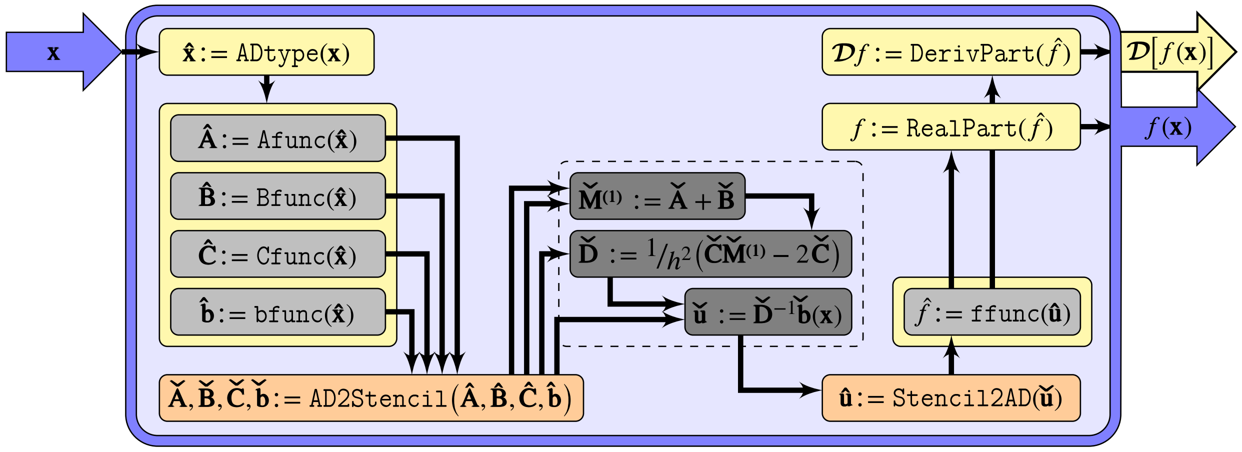
Before closing this summary chapter, recall that this particular example problem was chosen simply to motivate the use of automatic higher-order derivatives for matrix codes. This dissertation is concerned with the topic of matrix differentiation in general, not optimization in particular – as will be highlighted in the following chapter, automatic differentiation is not limited to optimization problems. As such, in this document analysis is limited to the derivation of NMD (as well as hybrids thereof); while numerical experiments are performed in Chapter 8 on a stochastic diffusion problem, this is mostly for the sake of benchmarking and as proof-of-concept. During the abstract discussions to follow, however, it is important to know that many concrete uses like the above demonstration are waiting in the wings.

The remainder of this document is organized as follows: in Part I, Chapter 2 will further motivate matrix differ-

[8] In fact, although this will not always be possible in general, in MATLAB it is possible to overload entire vectors as AD types (even if full matrix operations are too slow); as such, the above line of code can redefine every triplet – and thus the entire matrix $\mathbf{A}$ – with a single call.



(a)



(b)

Figure 1.2: AD-augmented schematic of (a) a generic program and (b) the example specified by Equation 1.1: User source code (light gray) is modified to use operator-overloaded AD data types (yellow); unaltered linear algebra routines (dark gray) operate on real-valued matrices defined by NMD (orange).

entiation by discussing a wider range of engineering applications that require second or higher derivatives; Chapter 3 will detail existing standard differentiation approaches (of both the automatic and manual variety); and Chapter 4 will expand the topic to include special algebras (i.e., non-real number systems) capable of differentiation. In Part II, Chapter 5 will derive new "hypercomplex" algebra types built with scalar nilpotent elements, capable of any-order differentiation; Chapter 6 will provide several examples of matrix structures that are isomorphic to particular scalar algebras, introduce the concept of stencils for differentiation, then provide an algorithm capable of finding stencils to represent arbitrary hypercomplex systems; and Chapter 7 will examine several hybrid methods, including the nilpotent and complex-step adjoint methods and a (non-adjoint) NMD-CD combination. Finally, in Part III Chapter 8 will describe the stochastic diffusion problem and investigate method accuracy and cost, and Chapter 9 will draw conclusions.

Chapter 2

Motivation: Numerical Uses of High-Order Derivatives

Many developing fields of numerical computing [11,13–17,21–40] require or are otherwise improved by access to high-order partial derivatives of computer codes – that is, those of higher order than the first derivatives found in the gradient or Jacobian; these methods concern the Hessian or higher tensors, sparse subsets thereof, or in some cases [19,41,42] implicit evaluations of derivative matrices projected onto a subspace. What follows is a non-comprehensive overview of several broad fields and specific algorithms that utilize these quantities. The discussion is limited to the usage of the derivatives in general and ignores how they are provided – existing methods that actually supply them will be presented in Chapter 3 and Chapter 4. Throughout this chapter, it will be taken as a given that an algorithm such as NMD, which can make derivatives of certain programs – namely, those with a heavy reliance on matrix operations – easier to obtain, will find a natural niche among the many different approaches to differentiating code.

2.1 Optimization

Most real-valued, single-metric optimization algorithms can be expressed as the search for an exact or approximate solution to the minimization problem

$$\operatorname{argmin}_{\mathbf{x}} \quad f(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^{N_x}, f \in \mathbb{R} \tag{2.1}$$

where f is a cost function to be minimized or the negative of a metric to be maximized. A great many methods attempt to find this solution by finding the point where the gradient is zero:

$$\nabla f(\mathbf{x}) = 0 \tag{2.2}$$

A number of algorithms accelerate this search by utilizing second derivative information to predict changes in ∇f. This section attempts to provide a brief, noninclusive overview of several such methods.

2.1.1 Newton's Method

Newton's method is a powerful, if costly, iterative optimization scheme which requires both first- and second-order derivatives. At each step in the algorithm, a quadratic interpolant is used to estimate the location of the minimum. Using a Taylor expansion to build the interpolant, the one-dimensional method update step can be derived as:

$$f(x_i + \Delta x) \approx \hat{f}(\Delta x) = f(x_i) + f'(x_i)\Delta x + f''(x_i)\frac{\Delta x^2}{2} \tag{2.3}$$

$$\hat{f}'(\Delta x) = 0 \tag{2.4}$$

$$\therefore \quad f'(x_i) + f''(x_i)\Delta x = 0 \tag{2.5}$$

$$\therefore \quad \Delta x = -f'(x_i)/f''(x_i) \tag{2.6}$$

Moving to many variables, the interpolant requires both the gradient and the Hessian matrix:

$$f(\mathbf{x}_i + \mathbf{\Delta x}) \approx \hat{f}(\mathbf{\Delta x}) = f(\mathbf{x}_i) + \nabla f(\mathbf{x}_i)^T \mathbf{\Delta x} + \frac{1}{2}\mathbf{\Delta x}^T \mathbf{H} \mathbf{\Delta x} \tag{2.7}$$

$$\nabla \hat{f}(\mathbf{\Delta x}) = 0 \tag{2.8}$$

$$\therefore \quad \nabla f(\mathbf{x}_i) + \mathbf{H}\mathbf{\Delta x} = 0 \tag{2.9}$$

$$\therefore \quad \mathbf{\Delta x} = -\mathbf{H}^{-1}\nabla f(\mathbf{x}_i) \tag{2.10}$$

where $\mathbf{H} \equiv \nabla^2 f(\mathbf{x}) \in \mathbb{R}^{N_x \times N_x}$ is the Hessian. The full method thus proceeds as shown in Algorithm 2.1.

A number of methods exist for efficiently finding ∇f, as will be discussed in Chapter 3. Unfortunately, the

Algorithm 2.1 Newton's Method

```
1. Given:          $\mathbf{x}_0 \in \mathbb{R}^{N_x}$                 // Initial estimate
2. Repeat:                                    // Enter Newton loop
                   a. $\mathbf{H}_i := \nabla^2 f(\mathbf{x_i})$           // Compute each entry of Hessian matrix
                   b. $\mathbf{\Delta x}_i := -\mathbf{H}_i^{-1} \nabla f(\mathbf{x}_i)$   // Solve linear system to compute Newton step
                   c. $\mathbf{x}_{i+1} := \mathbf{x}_i + \mathbf{\Delta x}_i$        // Increment approximate solution
   Until Converged                            // e.g., exit if $\mathbf{\Delta x}_i$ is very small
3. Output:         $\text{argmin}_x \quad f(\mathbf{x})$            // Converged optimum of $f$
```

cost of building the Hessian in Step 2a can be prohibitive (e.g., [26]), as second-order differentiation algorithms typically have a time complexity that is $\mathcal{O}({N_x}^2)$ more expensive than the original problem; for this reason, there are almost always more efficient optimization algorithms available. However, some application-specific second-order adjoint methods (e.g., [23, 25]; see Section 3.1.3 for more details) have been shown to find $\mathbf{H}$ in $\mathcal{O}(N_x)$, and in those applications Newton's method has been competitive. Alternatively, a broad category of "Quasi-Newton" methods exist for building a (relatively) inexpensive, continuously-updating approximation of the Hessian. (As this work concerns exact derivatives, such approximations will not be considered.) For problems with large N_x, though, inverting or even storing $\mathbf{H}$ can be too expensive. In such cases, one of the following methods is often preferred.

2.1.2 Linear and Nonlinear Conjugate Gradient Methods

As the names imply, both linear (CG, [43]) and (most) nonlinear conjugate gradient (NCG, [12]) methods are constructed using only a gradient (albeit merely an implicit one for linear CG), without higher-order information. They are described here for two reasons: first, linear CG is a central component of the "Hessian-free" method described in Section 2.1.3 and can be considered a subset of NCG; second and more importantly, second derivative information has repeatedly (e.g., [13, 15, 27]) been suggested for use in NCG, even though it has rarely been directly available.

The linear conjugate gradient method (more commonly referred to simply as *the* conjugate gradient method) is used to solve systems of linear equations, and can be derived by re-framing the problem as one of minimizing a quadratic function. However, it is not itself truly considered an optimization algorithm, nor does it require derivatives, and so only a brief summary is needed. In this work, all the reader needs to know is that (1) CG is a tool for solving (potentially very large) systems $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ when $\mathbf{A}$ is symmetric; (2) it does so by iterating through multiple search directions $\mathbf{d}_i$ to update an approximation $\mathbf{x}_i$ of the solution, until the residual $\mathbf{Ax}_i - \mathbf{b}$ is sufficiently small; and (3) during the iteration, it never technically needs the full $\mathbf{A}$ – only the result of the matrix-vector product $\mathbf{A}\mathbf{d}_i$, which is a vector that might be provided by a user-written function. As mentioned in Section 1.3, similar matrix-vector products (albeit, in every other case, *Hessian*-vector products) appear repeatedly throughout this work, in different contexts; in many circumstances, computing the product might have a small fraction the cost of building the entire matrix. This observation will become significant both for NCG and in Section 2.1.3, among many other places.

The nonlinear variant of CG is also derived in terms of minimization of a quadratic function; here, the framework is used more literally, to optimize some problem of the form given by Equation 2.1. Just like Newton's method, NCG assumes the problem is locally quadratic; it does not, however, always require second-order information (with notable exceptions, below). Very briefly, the method iterates through a search via the steps laid out in Algorithm 2.2.

Algorithm 2.2 Nonlinear Conjugate Gradient Optimization

```
1. Given:          $\mathbf{x}_0 \in \mathbb{R}^{N_x}$                          // Initial estimate
2. Initialize:     $\mathbf{d}_0 := \nabla f(\mathbf{x})$                      // Initial search direction
3. Repeat:                                               // Enter NCG loop
                   a. $\alpha_i := \mathrm{argmin}_\alpha \quad f(\mathbf{x}_i + \alpha\mathbf{d}_i)$    // Inner loop: line search for minimal value of $f$ along $\mathbf{d}_i$
                   b. $\mathbf{x}_{i+1} := \mathbf{x}_i + \alpha_i\mathbf{d}_i$               // Increment approximate solution
                   c. $\mathbf{d}_{i+1} := \nabla f(\mathbf{x}_{i+1}) + \beta_i\mathbf{d}_i$        // Increment search direction
   Until Converged                                       // e.g., exit if $\mathbf{\Delta x}$ is very small
4. Output:         $\mathrm{argmin}_x \quad f(\mathbf{x})$                     // Converged optimum of $f$
```

Here, $\mathbf{x}_i$ is the estimated solution at iteration i, $\mathbf{d}_i$ is the search direction, α_i is the scalar result of a one-dimensional line search, and β_i is the scalar NCG update parameter. Different line search methods for α and different formulas for β define different variants of the NCG method; a comprehensive survey can be found in [12].

Of particular interest for this work, in 1993 Pearlmutter [19] suggested (albeit in a context other than NCG) the line-search update step

$$\alpha_k = -\frac{\nabla f(\mathbf{x}_k)^T \mathbf{d}_i}{\mathbf{d}_i^T \mathbf{H}_k \mathbf{d}_i} \tag{2.11}$$

which is simply a one-dimensional Newton method in the direction of $\mathbf{d}_i$ – this equation is applied repeatedly in an inner loop until an ideal α_i is found. As for the direction update, in 1967 Daniel [13] proposed

$$\beta_i = \frac{\nabla f(\mathbf{x}_{i+1})^T \mathbf{H}_i \mathbf{d}_i}{\mathbf{d}_i^T \mathbf{H}_i \mathbf{d}_i} \tag{2.12}$$

and in 2008 Andrei [14] proposed the variant

$$\beta_i = \frac{\nabla f(\mathbf{x}_{i+1})^T \mathbf{H}_i \mathbf{d}_i - \nabla f(\mathbf{x}_{i+1})^T \mathbf{d}_i}{\mathbf{d}_i^T \mathbf{H}_i \mathbf{d}_i} \tag{2.13}$$

These options are often overlooked, due to the assumed cost of building $\mathbf{H}$; indeed, [12] explicitly chose not to discuss Daniel's approach because it "requires evaluation of the Hessian". As was shown in Section 1.3, though, these formulas do not technically require the Hessian matrix itself; they only require the Hessian-vector product $\mathbf{H}_i\mathbf{d}_i$ or the vector-Hessian-vector products $\mathbf{g}^T\mathbf{Hd}$ and $\mathbf{d}^T\mathbf{Hd}$, where $\mathbf{g} = \nabla f(\mathbf{x})$ is a separately-evaluated instance of the gradient.

Unfortunately, in the absence of NMD, calculating these products for large engineering applications can be difficult. (For his part, in [15] Andrei was forced to use a limited-accuracy approximation.) If they could be found efficiently and accurately, however, they could be extremely useful for NCG; indeed, the terms have broad utility in many methods, including the ones presented in the next section. Importantly, an existing approach capable of finding this quantity (albeit for a limited category of scalar codes, typically outside of engineering) does in fact exist, and will be discussed in Section 3.1.4; further, several entirely novel approaches will be derived and presented in Chapter 7.

2.1.3 Truncated-Newton and Hessian-Free Optimization

In cases where the main cost of the Newton method lies in the inversion (rather than the construction) of the Hessian matrix, a truncated-Newton method (TN, [28]) can be of great use. These methods make use of any iterative linear algebra solver (i.e., any solver that works by converging toward a solution, rather than solving directly – the linear conjugate gradient method is a prime example) to only *approximately* solve for the Newton update direction $\mathbf{\Delta x}$ via a reduced or "truncated" number of iterations. In other words, rather than computing the exact minimum of the interpolant from Equation 2.7, a truncated-Newton method merely comes close enough to generate an effective update direction. Step 2b in Algorithm 2.1 therefore turns into an inner loop, which is terminated upon reaching some (very relaxed) convergence criteria of its own.

So-called "Hessian-free" optimization methods (e.g., [16,17]) take TN one step further by choosing linear CG as the iterative method and providing $\mathbf{H}\boldsymbol{\Delta}\mathbf{x}$ directly, without building $\mathbf{H}$; they are thereby even more effective for problems with very large N_x where $\mathbf{H}$ is costly to build. This approach is shown in Algorithm 2.3. The now-familiar caveat still applies: existing methods that can find $\mathbf{H}\boldsymbol{\Delta}\mathbf{x}$ efficiently and exactly[1] are application-specific and more common outside of engineering (e.g., the cited Hessian-free example [16] concerns the deep learning of neural networks). The new NMD methods presented in Part II could conceivably make Hessian-free or other truncated-Newton methods more attractive to engineers.

Algorithm 2.3 Hessian-Free Optimization

```
1. Given:          $\mathbf{x}_0 \in \mathbb{R}^{N_x}$                          // Initial estimate
                   $\delta_{CG} \in \mathbb{R}$                                // CG convergence parameter
2. Repeat:                                                         // Enter Newton loop
                   a. Define: $\mathbf{h}(\mathbf{v}) \equiv \mathbf{H}(\mathbf{x}_i)\mathbf{v}$                 // Implicit matrix-vector product
                   b. Conjugate Gradient: $\mathbf{h}(\mathbf{v}) \approx -\nabla f(\mathbf{x}_i)$    // Inner loop: truncated solve of $\mathbf{Hv} \approx -\nabla f$
                         i. Increment CG Estimate $\mathbf{v}_j$
                         ii. Exit if: $\left|\mathbf{h}(\mathbf{v}_j) + \nabla f(\mathbf{x}_i)\right| < \delta_{CG}$
                   c. $\boldsymbol{\Delta}\mathbf{x}_i := \mathbf{v}$                              // Approximate Newton step
                   d. $\mathbf{x}_{i+1} := \mathbf{x}_i + \boldsymbol{\Delta}\mathbf{x}_i$                      // Increment approximate solution
  Until Converged                                                  // e.g., exit if $\boldsymbol{\Delta}\mathbf{x}_i$ is very small
3. Output:         $\mathrm{argmin}_x \quad f(\mathbf{x})$                        // Converged optimum of $f$
```

In cases where $\mathbf{H}$ might not be positive-definite, some algorithms simply add a parameterized constant along the diagonal, which can be done implicitly as $(\mathbf{H} + k\mathbf{I})\,\boldsymbol{\Delta}\mathbf{x} = \mathbf{H}\boldsymbol{\Delta}\mathbf{x} + k\boldsymbol{\Delta}\mathbf{x}$; others supply a different "curvature" matrix – for example, [29] used a modified variant of the Gauss-Newton (GN) method, a nonlinear least-squares algorithm. In standard GN, an approximation to $\mathbf{H}$ is built as $\mathbf{G} = \mathbf{J}^T\mathbf{J} \approx \mathbf{H}$ (where $\mathbf{J}$ is the Jacobian matrix), thereby bypassing any need to find actual second-order information; however, this approximation is only effective when very close to convergence or when nonlinear effects are minor. In the modified variant of GN, a program is divided into those subsections that are known to be well-behaved and those that are not. A variant of $\mathbf{G}$ is then built by chaining together a combination of Jacobians and Hessians – for example, if $f(\mathbf{x}) = g_1(g_2(\mathbf{x}))$ but the Hessian $\mathbf{H}^{(2)}$ of g_2 might not be

[1] Approximations are available, but as will be shown in Section 3.1.1, they are known to result in dramatic errors.

positive-definite, then a curvature matrix could be defined as $\mathbf{G} = \mathbf{J}^{(2)T}\mathbf{H}^{(1)}\mathbf{J}^{(2)}$, which is presumably better than using no nonlinear information at all. Note that these matrices can be built by applying standard differentiation methods (manual or automatic) to each program subsection in turn; with some effort, these methods can also calculate $\mathbf{G}\boldsymbol{\Delta}\mathbf{x}$ cheaply for a more reliable Hessian-free method. Again, NMD might be used in these circumstances to supply such derivatives across matrix routines.

2.2 Uncertainty Quantification and Sensitivity Analysis

As the name implies, uncertainty quantification (UQ, [44]) can be defined as the study, measurement, and prediction of uncertainty in any given process. It includes the propagation of uncertainties from uncertain inputs to uncertain outputs – e.g., using (approximations of) the probability distribution of program input(s) to estimate the distribution of the output(s) – and the inverse problem, where the uncertainties in real-world measurements of some system are compared to the outputs of a program modeling that system, in order to calibrate or quantify the input parameters of the program. UQ is often mentioned in the same breath as sensitivity analysis (SA, [45, 46]), which concerns the relative importance of each input, or the pairwise interaction (second-order) effects of inputs; it frequently involves the calculation of first- or second-order [32, 33] sensitivity *derivatives* – the name of which is simply an application-specific labeling of the same output-with-respect-to-input derivatives this dissertation concerns itself with. General use of UQ employs a very wide array of tools, including both SA and some of the surrogate modeling techniques to be discussed in Section 2.3; as will be seen there, many surrogates benefit from second-order derivatives. There are also uses of differentiation very specific to UQ; as one example of obvious relevance, for certain problems the inverse Hessian of a least-squares measure can yield a covariance matrix. [30, 31]

More broadly, one of the chief challenges faced in many UQ applications is often described as "the curse of dimensionality", wherein either (1) the number of input variables needed by a program or (2) the number of necessary sample points (and thus, the number of runs of the program) grows dramatically and nonlinearly with the complexity of the problem under consideration. Among other reasons, this is because arbitrary random variables and random processes can have a very high intrinsic number of dimensions – that is, many more than their physical range and domain would otherwise suggest. Once decomposed into orthogonal basis functions, it may take thousands of such functions (e.g., [47]) to capture all significant behavior – and each function must be weighted by a coefficient, either (1) provided by a very large input vector[2] or (2) calculated by comparison with an even larger database of function outputs and/or real-world measurements. In even more extreme cases, the most difficult inverse problems require one to use scattered and uncertain measurements to infer, not just model input parameters, but the entire internal state of a complex system. For example, in [31] oceanographers attempt to use satellite measurements of the ocean surface to infer the state of the deep sea (and the uncertainty thereof), the behavior of which is complex enough to require hundreds of millions of variables in a discretized PDE mesh.

This curse of dimensionality makes reverse-mode tools like the adjoint method (Section 3.1.2) very attractive, as they can produce derivatives with respect to each of N_x different inputs with a cost that remains independent of N_x. When users desire higher-order information for UQ applications, other reverse-mode methods will likely be similarly attractive. Two existing second-order reverse approaches will be discussed in Section 3.1.3 and Section 3.1.4, and again, this work will contribute several new options in Section 7.3, including an adjoint approach for derivatives higher than second order.

2.3 Surrogate Modeling

As mentioned repeatedly throughout this text, engineering simulations can be very expensive – and yet, a number of applications (including both optimization and uncertainty quantification) can require a great many output samples from these simulations. In many circumstances, it is more practical to provide cheap approximations of the output than it is to evaluate the true simulation each time. In these cases, one can build a low-fidelity "surrogate" model (SM) which attempts to inexpensively map the same inputs to similar outputs as the high-fidelity or "truth" model. This surrogate can then be evaluated many times, at a small fraction the cost of the high-fidelity model. One of the simplest surrogate models is a quadratic approximation – such as the Taylor-series interpolant from Equation 2.7, which explicitly called for the Hessian – but there are far more sophisticated options. There are more types than can practically be mentioned here; an incomplete list includes response surface methodology [48], polynomial chaos expansion [47], and Kriging and

[2] The demonstration problem detailed and implemented in Chapter 8 will provide one such example of UQ, by finding derivatives of an approximated random process (a two-dimensional field) with respect to the scalar random-variable coefficients (the program inputs) needed by a Karhunen-Loève expansion.

co-Kriging [22,49],[3] as well as many tools from the field of machine learning, such as support vector machines [50], genetic programming [51], and neural networks [52]. Another type known as multifidelity modeling is often described under the same umbrella (e.g., [11,53]) as surrogate modeling, although it could arguably be called distinct in that it does not fit a generic model to the high-fidelity code; instead, a lower-fidelity model is built via simplification of the full model. This can be done via the simple expedient of coarsening a discretized mesh or relaxing internal solution convergence criteria, or via more sophisticated reduced-order modeling (ROM) techniques that strip models down to only those states that are quantifiably important – for example, one of the most popular ROM tools is the proper orthogonal decomposition (POD) method, [54] which examines a large number of state-vectors sampled over time, then identifies (via principle component analysis) the subspace in which the process varies the most. Regardless of their inner workings, though, multifidelity models can generally be used in the same contexts as canonical surrogate types and described with much the same language.

Surrogates can be classified (e.g., in [55]) by their order, or degree of fit with the truth model at the point of construction (or at other high-fidelity points the model must fit): if, at some high-fidelity observation $f(\mathbf{x}_0)$, only the SM's scalar output $\hat{f}(\mathbf{x}_0)$ matches up, then the surrogate is called "zero-order"; if the gradient also matches (i.e., $\nabla\hat{f}(\mathbf{x}_0) = \nabla f(\mathbf{x}_0)$), it is called "first-order"; and if the Hessian matches (i.e., $\nabla^2\hat{f}(\mathbf{x}_0) = \nabla^2 f(\mathbf{x}_0)$), it is called "second-order". Methods calling for a second-order surrogate (e.g., [11, 53, 56]) imply a need to either build an effective approximation of $\mathbf{H}$ or else directly collect at least some second-order derivative information from the high-fidelity model. Further, even in some cases that do not require derivative agreement, first [47] and second [22,49] derivatives have been shown to improve SM properties, simply by providing additional information for the model to fit.

Possible uses of surrogate models (of any order) include Monte Carlo modeling for uncertainty quantification, wherein a SM is run a great many times with random-variable inputs for the sake of studying the output probability distribution. In this case, the SM need not even be zero-order (e.g., it might smooth the data such that it doesn't exactly fit some points), but the better the approximation of the surrogate, the more accurate the distribution. Another major use is surrogate-based optimization (SBO, [53]), wherein a high-fidelity problem $f(\mathbf{x})$ is optimized by iteratively

[3] In [49] a Hessian-vector product was used to supply further covariance data (beyond the original outputs and the gradient) to a co-Kriging model, but this was only in addition to the data from a whole Hessian; the multiplication appears to have been explicit, computed after the whole Hessian was found.

(1) building a local surrogate $\hat{f}_i(\cdot)$ from high-fidelity samples at or near the current location $\mathbf{x}_i$; (2) finding the input $\mathbf{x}_{i+1}$ that optimizes the low-fidelity problem $\hat{f}_i(\mathbf{x}_{i+1})$, which might require many evaluations of the surrogate within some inner-loop optimization algorithm; and (3) updating or re-building the surrogate $\hat{f}_{i+1}(\cdot)$ from new high-fidelity evaluations. This process is shown in Algorithm 2.4; it can also be made more sophisticated by restricting step (2) to stay within a "trust region" (e.g., [55,56]) where the surrogate is expected to provide a good approximation. Many SBO and trust-region algorithms are built on the assumption that $\hat{f}$ is (at least) first-order, in which case they are proven to converge to the high-fidelity optimum. However, first-order SMs effectively limit one to first-order (gradient-only) optimization algorithms [11]; to get a benefit from a second-order approach like Newton's method or Hessian-enhanced NCG, the curvature of the surrogate must match that of the truth model. In the case of multifidelity modeling, where the low-fidelity models are not even guaranteed to be zero-order, methods exist to overlay one or more "correction" [11] surrogates to make the combined model match; this requires collecting derivative information[4] from both the high- and low-fidelity models.

Algorithm 2.4 Second-Order Surrogate-Based Optimization

```
1. Given:          $\mathbf{x}_0 \in \mathbb{R}^{N_x}$                       // Initial estimate
2. Repeat:                                             // Outer (high fidelity) loop
                   a. Define $\hat{f}_i$ such that:                  // Surrogate model of $f(\mathbf{x})$ near $\mathbf{x}_i$
                          $\hat{f}_i(\mathbf{v}) \approx f(\mathbf{x}_i + \mathbf{v})$                  // Zero-order fit
                          $\nabla_\mathbf{v}\hat{f}_i(\mathbf{v}) \approx \nabla_\mathbf{v} f(\mathbf{x}_i + \mathbf{v})$          // First-order fit
                          $\nabla^2_\mathbf{v}\hat{f}_i(\mathbf{v}) \approx \nabla^2_\mathbf{v} f(\mathbf{x}_i + \mathbf{v})$        // Second-order fit
                   b. $\Delta\mathbf{x}_i := \mathrm{argmin}_\mathbf{v} \quad \hat{f}(\mathbf{v})$        // Inner (low fidelity) loop: optimizer of choice
                   c. $\mathbf{x}_{i+1} := \mathbf{x}_i + \Delta\mathbf{x}_i$                  // Increment approximate solution
   Until Converged                                     // e.g., if $\Delta\mathbf{x}_i$ is very small
3. Output:         $\mathrm{argmin}_x \quad f(\mathbf{x})$                   // Converged optimum of $f$
```

2.4 A Selection of Higher-Order Applications

The preceding sections concerned themselves with second-order (Hessian) derivatives, which the simplest in-

[4] Note that the cited example again incorrectly assumes the need to build the full Hessian – those authors' second-order corrections could be reframed to only require Hessian-vector or vector-Hessian-vector products.

stances of NMD can easily provide. However, NMD can be used to arbitrarily high order. Uses for third and higher derivatives are comparatively rare, but hardly unheard of; some of the obvious cases are those where derivative requirements stack – for example, optimization on a embedded manifold (e.g., a curved, constrained subspace of the search space, imbued with a "geodesic", or nonlinear redefinition of a straight line) requires first or second derivatives with respect to the manifold curvature, on top of the first or second derivatives of the objective function being optimized. [57–59] Beyond stacking, two of the broadest uses of any-order derivatives are Householder's methods and generic Taylor (power) series.

Householder's methods [37] are essentially higher-order generalizations of Newton's method, albeit used to find roots – rather than optima – of functions; under this framework, Newton's method is the first-order Householder method (utilizing only first derivatives) to find where a function is zero. (In other words, Newton optimization finds the root of the derivative function.) Householder methods can be generalized to any order of derivative; each successive order also increases the method's order of convergence. Unfortunately, the cost of finding higher derivatives often outpaces the speedup from faster convergence – however, the second-order variant, known as the Halley method, has been shown to be competitive for certain problems. [38]

Much more practical is the use of arbitrary Taylor series, which can be used to approximate difficult nonlinear functions to any desired degree of accuracy. For example, in astrodynamics [39,40], satellite guidance can be informed by fifth- or higher-order series approximating a gravity potential, and in the study of particle accelerators [36], beam dynamics can be described by power series of arbitrarily high order. Taylor series are also a useful tool for numerical bifurcation analysis [34, 35, 60], itself a very broad field with applications in turbulence and chaos theory.

Chapter 3

Standard Differentiation Approaches

As shown in the previous chapter, many engineering applications benefit from the use of second- or higher-order derivatives. Of course, many methods for numerical differentiation exist, of varying precision, ease of use, and computational cost. Beyond this, differentiation approaches can generally be divided into two categories: those that work in "forward mode" and those that work in "reverse mode". Forward-mode methods begin by modifying either the program inputs or the way those inputs are treated by the program, then run to completion before examining the modified output. Reverse-mode methods begin with the pre-computed, undifferentiated output; find the derivative of this value, not with respect to the program input but to the penultimiate (second-to-last) operation(s) performed by the program; then proceed backward, computing the chain rule in reverse for every operation performed. This approach can be very efficient for problems with many inputs (e.g., single-metric optimization over many parameters), as the cost is typically a function only of the number of outputs. The forward mode, for its part, is dependent only on the number of inputs, and can be very efficient in cases with many outputs (e.g., derivatives of an entire converged state vector).

Ultimately, between precision, ease, cost, and forward vs reverse mode, the choice of ideal method is problem-dependent and a matter of user preference. To give a broad summary of the approaches available, this chapter will begin by discussing the most widely-used forward-mode non-AD technique, finite differencing, and the most popular reverse-mode technique in engineering, the manually-implemented adjoint method, in both its first- and second-order forms. After presenting the oft-mentioned technique for computing $\mathbf{Hv}$, discussion will move on to automatic differentiation itself and the computation of first- and higher-order derivatives for scalars and matrices. Finally, some differentiation approaches differ in the underlying theory used to derive and implement them; discussion of these non-standard methods will be saved for Chapter 4.

3.1 Leading Non-Automatic Tools

In the broad ecosystem of competing differentiation methods for engineering applications, AD tools (including NMD) fill the niche that demands precise values with minimal programming effort. Other niches, of course, have different demands. Two of the most common demands are (1) very simple approximations and (2) maximally efficient gradients; these are usually addressed by the finite difference and adjoint methods, respectively. Another niche is the computation of derivative matrix products, which can be addressed by some AD tools or by intrusive implementation. (It must be mentioned yet again: this can be more difficult for matrix codes.) All three approaches are addressed below. It should be noted that finite differencing, which is ignorant of the internal workings of a program, is already compatible with matrix subroutines; it is, however, very limited in precision. The discrete adjoint method, for its part, is entirely defined by matrix operations – its main limitation lies in its implementation difficulty.

3.1.1 Finite Differences: Easy, Approximate, Forward-Mode Differentiation

The most commonly used method for finding a numerical derivative is the obvious approach: mimic the definition of the derivative by running two simulations with very slightly different inputs and computing the slope between the outputs:

$$\begin{aligned}\frac{df}{dx} &\equiv \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \\ &\approx \left.\frac{f(x+h)-f(x)}{h}\right|_{h\ll 1}\end{aligned} \tag{3.1}$$

This is the simplest type of what are more broadly known as finite difference (FD) [61] methods. Many engineers are probably more familiar with finite differences as a means of discretizing a continuous variable over a mesh; while this process involves manipulation of the same underlying FD equations, it must be made clear that this is not the type of method considered within this work. Instead, herein FD methods are only considered in the context of computing derivatives of entire functions or subfunctions with respect to their inputs.

While FD is very simple to implement, it suffers from accuracy limitations. One would expect that the closer the data points – the smaller the step size h between them – the more accurate the derivative computation will be.

However, because computers are limited to a certain number of bits for each number, when the two function outputs are almost identical subtractive cancellation error (a loss of significant figures, also known as rounding error) begins to occur and gets progressively worse as the step size decreases; this behavior is shown in Figure 3.1. Practical use of FD often involves careful work to decide on a step-size; worse, for many problems, there is no globally optimal value of h applicable to each derivative direction or every parameter setting.

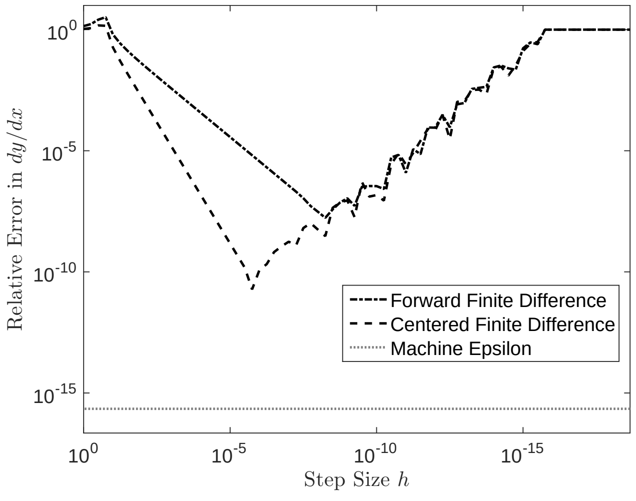
Of course, more sophisticated FD methods exist than such simple rise-over-run calculations. [64,65] In general, the label "finite differencing" is used to describe any class of method that compares function values at multiple points in order to infer an approximate derivative of that function. More specifically, FD methods work by assigning a weight to each function output and taking the sum. (For example, in Equation 3.1, the weight for $f(x+h)$ would be $1/h$ and the weight for $f(x)$ would be $-1/h$.) This general framework can be used to find approximate derivatives of any order, for any number of inputs, from any number of function evaluations (above a certain minimum number required for a given derivative) at any location, with any order of approximation accuracy. However, no matter the formula,[1] FD can never be truly exact – subtractive cancellation is always a factor. Further, for the derivatives-of-functions usage considered here, in the case of expensive engineering simulations most users try to avoid using more than a few function evaluations, instead limiting themselves to very simple formulas. The most common formula remains the so-called forward difference method of Equation 3.1, which is accurate to $\mathcal{O}(h)$ (at least in the regime of h unaffected by cancellation error); the next most popular formula is the central difference method:

$$\frac{df}{dx} \approx \left. \frac{f(x+h) - f(x-h)}{2h} \right|_{h \ll 1} \tag{3.2}$$

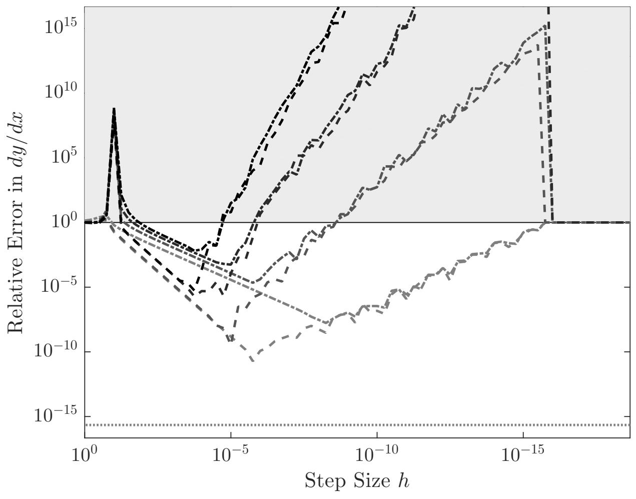
which is accurate to $\mathcal{O}(h^2)$. (See Figure 3.1(a).) For cases with multiple inputs (e.g., $f(\mathbf{x})$ for some vector of inputs $\mathbf{x}$), the forward difference method requires fewer overall function evaluations – only a single offset evaluation in each direction, as opposed to two evaluations for central differences.

In the case of higher derivatives, it is very common to inductively apply a first-order formula multiple times. For example, for a centered univariate second derivative, simply take the slope between $f'(x + h/2)$ and $f'(x - h/2)$,

[1] With the arguable exception of FD in the complex plane, which will be discussed in Section 4.1.



(a)



(b)

Figure 3.1:
(a) Finite differencing error in the first derivative. Following the traditions set by [62] and [63], the test function is $y(x) = e^x/\sqrt{\sin^3 x + \cos^3 x}$ evaluated at $x = 3\pi/4 - 0.2$ and h is presented logarithmically decreasing from left to right, such that expected "better" values are on the right. Note that forward and centered FD initially follow $\mathcal{O}(h)$ and $\mathcal{O}(h^2)$ decreases in error, respectively, until dominated by $\mathcal{O}(h)$ subtractive cancellation error. The minimal-error value of h is problem-dependent, and in real-world applications no comparison to a known value will be available; as such, FD error will likely be significant.
(b) FD error in first through fourth derivatives of the same test function. Higher derivatives use darker lines; the shaded area indicates relative error greater than 100%. Note that the higher the order of derivative, the greater the $\mathcal{O}$ growth rate in subtractive error and the smaller (i.e., the harder to identify) the minimal-error domain of h. Further note the early spike in error due to the strongly nonlinear nature of the test function – attempts to avoid subtractive error with large step sizes can still lead to wildly incorrect results.

each of which are in turn approximated by their own central differences:

$$\frac{d^2 f}{dx^2} = \lim_{h\to 0} \frac{\left(\frac{f(x+h)-f(x)}{h}\right) - \left(\frac{f(x)-f(x-h)}{h}\right)}{h} \tag{3.3}$$

$$= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}\left(h^2\right) \tag{3.4}$$

Unfortunately, whether univariate or multivariate, higher orders of derivatives suffer from higher error and are more sensitive to step size – see Figure 3.1(b). Finally, as a forward-mode method, FD requires additional function evaluations for each individual derivative in a gradient or Hessian; for problems with many inputs, this can be prohibitively costly. An alternative differentiation approach known as the adjoint method addresses these problems, finding exact derivatives for many inputs.

3.1.2 The Discrete Adjoint Method: Intrusive, Exact, Reverse-Mode Differentiation

In terms of usage, when evaluating competing strengths and weaknesses the adjoint method might be thought of as an exact opposite of FD: it is a reverse-mode approach that requires problem-specific derivation and intrusive manual implementation, yet yields exact values and can compute an entire gradient with a run-time cost comparable to the underlying function itself. Just as with FD, though, the method describes a broad approach, not a single algorithm. That approach can generally be described as a type of blockwise reverse-mode differentiation, applied to programs that utilize vector quantities as intermediate variables; rather than operate on each atomic operation, the discrete version of the approach divides a program into more manageable vector-valued subroutines and utilizes key Jacobian[2] matrices.

This subsection, while not exhaustive, will examine one of the most common generalizations of an end-to-end (input-to-output) adjoint approach. In this generalization, the underlying (forward) problem takes a vector of input parameters $\mathbf{x} \in \mathbb{R}^{N_x}$; somehow solves for or converges to a state vector $\mathbf{u}(\mathbf{x}) \in \mathbb{R}^{N_u}$ (where typically $N_u \gg N_x$) according to some constraint or residual function $\mathbf{r}(\mathbf{u}, \mathbf{x}) \in \mathbb{R}^{N_u}$, which is assumed be be zero for any converged $\mathbf{u}$; then computes a scalar quantity of interest $f(\mathbf{u}, \mathbf{x}) \in \mathbb{R}$ as output. (Vector-valued outputs are less common, but can still be informed by the following procedure.) In this process, the solution of $\mathbf{r} = \mathbf{0}$ for $\mathbf{u}$ is assumed to constitute the

[2] As will be discussed in Section 3.1.3, there are also (comparatively rare) second-order adjoint approaches that additionally utilize the Hessian.

overwhelming majority of the work, and thus is assumed to be the most difficult subsection to differentiate in terms of both human labor and computational run-time. As a simple example from aerospace design optimization, $\mathbf{x}$ might be at set of airfoil design variables to be optimized, $\mathbf{u}$ might be a set of steady-state flow variables (pressure, velocities, etc) at many points in a discretized mesh, $\mathbf{r}$ might be the residuals of both the discretized flow PDEs and boundary conditions, and f might simply be the lift generated by a given design – easily computed by examining the solved-for pressure at each mesh point.

Other variants of the adjoint method exist to find the gradients of intermediate quantities within programs (e.g., [66]) or of functions integrated across time ([67]); many applications break $\mathbf{u}$ or $\mathbf{r}$ into multiple quantities – e.g., [22, 68] track flow and geometry variables in distinct state vectors, while [22, 67] have different functions for flow residuals, grid deformation residuals, initial conditions, or boundary value constraints. Finally, while "discrete" methods (including the one to follow) operate on the existing computational (already-discretized) quantities from the forward problem, "continuous" methods begin with the underlying equations describing the physical system of interest.

In all cases, the goal is to find some gradient – in this case, $\mathbf{g} \equiv \nabla_x f = {}^{\mathrm{d}f}/_{\mathrm{d}\mathbf{x}}$, where ${}^{\mathrm{d}(\cdot)}/_{\mathrm{d}\mathbf{x}}$ represents the total derivative. Note that, unlike some other fields discussed in Chapter 2, in this context $\mathbf{g}$ is by convention a row vector, as is any other derivative of a scalar with respect to a vector. The quantities $\mathbf{x}$, $\mathbf{u}$, and $\mathbf{r}$ are column vectors; derivatives of these with respect to scalars remain column vectors, while derivatives of columns with respect to vectors yield Jacobian matrices indexed by the respective (row, column) indices.

For this problem formulation, derivation of the adjoint method begins by expanding the total derivative:

$$\mathbf{g} = \frac{\mathrm{d}f(\mathbf{u}(\mathbf{x}),\mathbf{x})}{\mathrm{d}\mathbf{x}} = \frac{\partial f}{\partial \mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{x}} + \frac{\partial f}{\partial \mathbf{x}} \tag{3.5}$$

where ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ is the Jacobian of $\mathbf{u}$ with respect to $\mathbf{x}$. Note that, as the bulk of the user program is assumed to go into finding $\mathbf{u}(\mathbf{x})$, this matrix may be difficult or impossible to find directly. Further note that, as $\mathbf{u}$ is presumably a converged PDE solution where each u_i is correlated with other $u_{j\neq i}$ (not to mention other quantities such as the geometry of the computational mesh, etc.), it is impossible to change any single u_i in isolation – therefore, $\mathbf{u}(\mathbf{x})$ is more properly described as $\mathbf{u}(\mathbf{y}(\mathbf{x}))$ for some unknown $\mathbf{y}(\mathbf{x})$, such that the partial derivative Jacobian ${}^{\partial\mathbf{u}}/_{\partial\mathbf{x}} = 0$. This technicality can safely be ignored here, but will play a brief role in the second-order derivation in Section 3.1.3.

In the next step of the derivation, because **u** is assumed to be converged, **r** will always be zero – that is, any change in **x** implies a corresponding change in **u** which satisfies

$$\mathbf{r}(\mathbf{u}(\mathbf{x}),\mathbf{x}) = 0 \tag{3.6}$$

Because this quantity never changes, the total derivative is also zero:

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}\mathbf{x}} = \frac{\partial \mathbf{r}}{\partial \mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{x}} + \frac{\partial \mathbf{r}}{\partial \mathbf{x}} = \mathbf{0} \tag{3.7}$$

Thus, by imposing the convergence criteria, one can *indirectly* extract an expression for $^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{x}} = -\frac{\partial \mathbf{r}}{\partial \mathbf{u}}^{-1}\frac{\partial \mathbf{r}}{\partial \mathbf{x}} \tag{3.8}$$

Note that, as $^{\partial\mathbf{r}}/_{\partial\mathbf{x}}$ is a matrix and not a vector, actually computing $^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ would require N_x system solutions. However, it is known from Equation 3.5 that the full $^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ is not required – only the projection of that matrix onto a row vector. Inserting Equation 3.8 into 3.5 yields:

$$\mathbf{g} = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}} = \left(-\frac{\partial f}{\partial \mathbf{u}}\frac{\partial \mathbf{r}}{\partial \mathbf{u}}^{-1}\right)\frac{\partial \mathbf{r}}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{x}} \tag{3.9}$$

where the term in parenthesis has been singled out for special consideration. This term is a row vector times a matrix, which yields another row vector; under a transpose, it becomes

$$\left(-\frac{\partial f}{\partial \mathbf{u}}\frac{\partial \mathbf{r}}{\partial \mathbf{u}}^{-1}\right)^{T} = -\left(\frac{\partial \mathbf{r}}{\partial \mathbf{u}}^{T}\right)^{-1}\frac{\partial f}{\partial \mathbf{u}}^{T} \tag{3.10}$$

which is simply the (column-vector) solution to a linear system of equations. Denote this solution as a vector $\boldsymbol{\lambda}$ of "*adjoint variables*",[3] defined by what is known as the *adjoint equation*:

$$\frac{\partial \mathbf{r}}{\partial \mathbf{u}}^{T} \boldsymbol{\lambda} = -\frac{\partial f}{\partial \mathbf{u}}^{T} \tag{3.11}$$

Substituting $\boldsymbol{\lambda}$ into Equation 3.9 yields:

$$\mathbf{g} = \boldsymbol{\lambda}^{T} \frac{\partial \mathbf{r}}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{x}} \tag{3.12}$$

In practice, then, the adjoint method for finding the gradient across the entire program consists of three steps: first, building (1) the intermediate gradients $\partial f/\partial \mathbf{x} \in \mathbb{R}^{N_x}$ and $\partial f/\partial \mathbf{u} \in \mathbb{R}^{N_u}$ and (2) the Jacobians $\partial \mathbf{r}/\partial \mathbf{u} \in \mathbb{R}^{N_u \times N_u}$ and $\partial \mathbf{r}/\partial \mathbf{x} \in \mathbb{R}^{N_u \times N_x}$; second, solving the adjoint equation 3.11 for $\boldsymbol{\lambda}$; and third, inserting $\boldsymbol{\lambda}$ into Equation 3.12.

Conceptually, the adjoint variables can be thought of as a linearized relationship between the output and the residual function, just as the original solution state vector represents a relationship between the residual and the inputs. Likewise, the matrix $\partial \mathbf{r}/\partial \mathbf{u}$, which – as will be shown below – is often already found by the original forward problem, can be thought of as representing the process by which the program finds $\mathbf{u}$; the matrix $\partial \mathbf{r}/\partial \mathbf{x}$ as representing how the input parameters influence that process; the vector $\partial f/\partial \mathbf{u}$ as representing how the output quantity of interest depends on the state vector; and the vector $\partial f/\partial \mathbf{x}$ as representing any dependency of f on $\mathbf{x}$ which bypasses $\mathbf{u}$ – e.g., attempts by the user to penalize (negatively weight) extreme values in $\mathbf{x}$. Note that by finding $\boldsymbol{\lambda}$, the adjoint method has bypassed the difficultly of either solving for or differentiating $\mathrm{d}\mathbf{u}/\mathrm{d}\mathbf{x}$, which would have represented the complicated dependence of the solution state on the input vector. Building the other Jacobians, in contrast, is far more tractable.

Take as simple example a program that solves the canonical $\mathbf{A}(\mathbf{x})\,\mathbf{u} = \mathbf{b}(\mathbf{x})$ and assume the quantity of interest

[3] Note that so-called adjoint variables, from which the adjoint method draws its name, are not actually specific to this method; instead, they generally describe intermediate variables in any reverse-mode differentiation method.

is the Euclidean norm of **u**:

$$\mathbf{u} = \mathbf{A}(\mathbf{x})^{-1}\mathbf{b}(\mathbf{x}) \tag{3.13}$$

$$f(\mathbf{x}) = f(\mathbf{u}) = \sqrt{\mathbf{u}^T\mathbf{u}} \tag{3.14}$$

The residual of the solution **u** is:

$$\mathbf{r}(\mathbf{u},\mathbf{x}) = \mathbf{A}(\mathbf{x})\,\mathbf{u} - \mathbf{b}(\mathbf{x}) \tag{3.15}$$

The derivative of this quantity with respect to **u** is thus

$$\frac{\partial \mathbf{r}}{\partial \mathbf{u}} = \mathbf{A}(\mathbf{x}) \tag{3.16}$$

which was already built by the forward problem. The derivative with respect to **x** is

$$\frac{\partial \mathbf{r}}{\partial \mathbf{x}} = \frac{\partial\big(\mathbf{A}(\mathbf{x})\,\mathbf{u}\big)}{\partial \mathbf{x}} - \frac{\partial \mathbf{b}}{\partial \mathbf{x}} \tag{3.17}$$

which can be defined elementwise as

$$\begin{aligned}\left[\frac{\partial \mathbf{r}}{\partial \mathbf{x}}\right]_{i,j} = \frac{\partial r_i}{\partial x_j} &= \frac{\partial\left(\sum_k A_{i,k}(\mathbf{x})\,u_k\right)}{\partial x_j} - \frac{\partial b_i}{\partial x_j} \\ &= \sum_k \frac{\partial A_{i,k}}{\partial x_j}u_k - \frac{\partial b_i}{\partial x_j}\end{aligned} \tag{3.18}$$

The two intermediate gradients of f are

$$\frac{\partial f}{\partial \mathbf{u}} = \frac{\mathbf{u}}{\sqrt{\mathbf{u}^T\mathbf{u}}} = \frac{\mathbf{u}}{f(\mathbf{u})} \tag{3.19}$$

which is trivial to compute, and

$$\frac{\partial f}{\partial \mathbf{x}} = 0 \tag{3.20}$$

because f does not depend directly on $\mathbf{x}$.

In this example and in general, $\partial \mathbf{r} / \partial \mathbf{x}$ can be built in multiple ways, all of which require some form of differentiation – potentially, in fact, by one of the traditional tools of automatic differentiation that will be discussed in Section 3.2. More often, the user must manually differentiate the equations, either elementwise via Equation 3.18 or through some more efficient, problem-specific approach.

3.1.3 Second-Order Adjoints

Second-order (Hessian) adjoint methods cannot, unfortunately, run in constant ($\mathcal{O}(1)$) time. To understand why, think of a first-order method as running the original problem in reverse, producing one output for every original input. Now conceptualize a second-order approach as applying a second analytical adjoint methodology to the results of the first – that is, as a method that considers the gradient to be N_x different outputs of a new function, then finds the gradient of each of these outputs to build the Hessian matrix. Such a process necessarily has complexity $\mathcal{O}(N_x)$, as reverse-mode methods are linear in the number of outputs. This is still vastly superior to any forward-mode method, which would have complexity $\mathcal{O}({N_x}^2)$

More broadly than this adjoint-of-an-adjoint approach, discussions of second-order methods in the literature often (e.g., [25,68,69]) present four different approaches at once, derived by different combinations and orderings of (1) the direct differentiation of either the forward equations or some intermediate adjoint equation and (2) the introduction of new adjoint variables. What follows is an independent derivation of what has uniformly been found to be the most efficient approach: the direct second-order differentiation of the forward equations, followed afterward by the introduction of an adjoint vector; methods such as this are often simply presented in isolation (e.g., [22, 70]) without discussion of the other approaches. Note that the three omitted methods require at least the same amount of derivation and implementation work, if not more; the method presented here is not more analytically advanced, *per se* – it is simply more cost-effective.

Before beginning the derivation, note that it necessarily includes rank-three tensors (the derivatives of vectors with respect to two other vectors). As such, it becomes convenient to adopt an index notation to differentiate individual scalars at a time, using the indices $(i,\ j)$ to index into $\mathbf{x}$ and $(m,\ n)$ to index into $\mathbf{u}$ – that is, $i, j \in \{1, \dots, N_x\}$ and $m, n \in \{1, \dots, N_u\}$. When practical, the resulting equations will be reformulated in ways that take as much advantage of matrix arithmetic as possible.

The derivation begins with the same total derivative operator used in Equation 3.5:

$$\frac{\mathrm{d}(\cdot)}{\mathrm{d}\mathbf{x}} = \frac{\partial(\cdot)}{\partial\mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{x}} + \frac{\partial(\cdot)}{\partial\mathbf{x}} \tag{3.21}$$

$$\frac{\mathrm{d}(\cdot)}{\mathrm{d}x_i} = \sum_m \left(\frac{\partial(\cdot)}{\partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\right) + \frac{\partial(\cdot)}{\partial x_i} \tag{3.22}$$

$$\frac{\mathrm{d}}{\mathrm{d}x_i}\left[f(\mathbf{u}(\mathbf{x}), \mathbf{x})\right] = \sum_m \left(\frac{\partial f}{\partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\right) + \frac{\partial f}{\partial x_i} \tag{3.23}$$

Applying this operator a second time yields:

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x_i\mathrm{d}x_j} = \frac{\mathrm{d}}{\mathrm{d}x_j}\left[\sum_m \left(\frac{\partial f}{\partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\right) + \frac{\partial f}{\partial x_i}\right] \tag{3.24}$$

$$= \sum_m \left[\left(\frac{\mathrm{d}}{\mathrm{d}x_j}\left[\frac{\partial f}{\partial u_m}\right]\right)\frac{\mathrm{d}u_m}{\mathrm{d}x_i} + \frac{\partial f}{\partial u_m}\frac{\mathrm{d}^2 u_m}{\mathrm{d}x_i\mathrm{d}x_j}\right] + \frac{\mathrm{d}}{\mathrm{d}x_j}\left[\frac{\partial f}{\partial x_i}\right] \tag{3.25}$$

$$= \sum_m \left[\left(\frac{\partial}{\partial u_m}\left[\frac{\mathrm{d}f}{\mathrm{d}x_j}\right]\right)\frac{\mathrm{d}u_m}{\mathrm{d}x_i} + \frac{\partial f}{\partial u_m}\frac{\mathrm{d}^2 u_m}{\mathrm{d}x_i\mathrm{d}x_j}\right] + \frac{\partial}{\partial x_i}\left[\frac{\mathrm{d}f}{\mathrm{d}x_j}\right] \tag{3.26}$$

where the two instances of ${}^{\mathrm{d}f}/_{\mathrm{d}x_j}$ are entries of the original gradient, which can be expanded to yield:

$$\begin{aligned}\frac{\mathrm{d}^2 f}{\mathrm{d}x_i\mathrm{d}x_j} &= \sum_m \left[\left(\frac{\partial}{\partial u_m}\left[\sum_n \left(\frac{\partial f}{\partial u_n}\frac{\mathrm{d}u_n}{\mathrm{d}x_j}\right) + \frac{\partial f}{\partial x_j}\right]\right)\frac{\mathrm{d}u_m}{\mathrm{d}x_i} + \frac{\partial f}{\partial u_m}\frac{\mathrm{d}^2 u_m}{\mathrm{d}x_i\mathrm{d}x_j}\right] \\ &\quad + \frac{\partial}{\partial x_i}\left[\sum_n \left(\frac{\partial f}{\partial u_n}\frac{\mathrm{d}u_n}{\mathrm{d}x_j}\right) + \frac{\partial f}{\partial x_j}\right]\end{aligned} \tag{3.27}$$

$$\begin{aligned}&= \sum_m \sum_n \left(\frac{\partial^2 f}{\partial u_m \partial u_n}\frac{\mathrm{d}u_n}{\mathrm{d}x_j} + \frac{\partial f}{\partial u_n}\frac{\mathrm{d}}{\mathrm{d}x_j}\left[\cancelto{0}{\frac{\partial u_n}{\partial u_m}}\right]\right)\frac{\mathrm{d}u_m}{\mathrm{d}x_i} \\ &\quad + \sum_m \frac{\partial^2 f}{\partial x_j \partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_i} + \frac{\partial f}{\partial u_m}\frac{\mathrm{d}^2 u_m}{\mathrm{d}x_i\mathrm{d}x_j}\end{aligned} \tag{3.28}$$

$$
\begin{aligned}
&+ \sum_n \frac{\partial^2 f}{\partial x_i \partial u_n}\frac{\mathrm{d}u_n}{\mathrm{d}x_j} + \frac{\partial f}{\partial u_n}\frac{\mathrm{d}}{\mathrm{d}x_j}\left[\cancelto{0}{\frac{\partial u_n}{\partial x_i}}\right] \\
&+ \frac{\partial^2 f}{\partial x_i \partial x_j}
\end{aligned}
$$

The first crossed-out is term is zero because individual elements of **u** are not direct functions of each other; the second is zero because (as mentioned above) $\mathbf{u}(\mathbf{x})$ is in actuality $\mathbf{u}(\mathbf{y}(\mathbf{x}))$ for some unknown $\mathbf{y}$, and hence, all partial derivatives with respect to $\mathbf{x}$ are zero; only the total derivative is non-zero. The remaining equation simplifies to:

$$
\begin{aligned}
\frac{\mathrm{d}^2 f}{\mathrm{d}x_i \mathrm{d}x_j} &= \sum_m \sum_n \frac{\partial^2 f}{\partial u_m \partial u_n}\frac{\mathrm{d}u_n}{\mathrm{d}x_j}\frac{\mathrm{d}u_m}{\mathrm{d}x_i} \\
&+ \sum_m \frac{\partial^2 f}{\partial x_j \partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_i} + \frac{\partial^2 f}{\partial x_i \partial u_m}\frac{\mathrm{d}u_m}{\mathrm{d}x_j} + \frac{\partial f}{\partial u_m}\frac{\mathrm{d}^2 u_m}{\mathrm{d}x_i \mathrm{d}x_j} \\
&+ \frac{\partial^2 f}{\partial x_i \partial x_j}
\end{aligned} \tag{3.29}
$$

This can be expressed in somewhat more compact form using matrices, but to avoid the rank-three tensor ${}^{\mathrm{d}\mathbf{u}^2}/_{\mathrm{d}\mathbf{x}^2}$, derivatives must still be taken one at a time:

$$
\frac{\mathrm{d}^2 f}{\mathrm{d}x_i \mathrm{d}x_j} = \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i}^T \frac{\partial^2 f}{\partial \mathbf{u}^2}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 f}{\partial \mathbf{u}\partial x_j}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2 f}{\partial \mathbf{u}\partial x_i}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial f}{\partial \mathbf{u}}\frac{\mathrm{d}^2 \mathbf{u}}{\mathrm{d}x_i \mathrm{d}x_j} + \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{3.30}
$$

Here, ${}^{\partial^2 f}/_{\partial \mathbf{u}^2}$ is the partial-derivative Hessian of f with respect to $\mathbf{u}$, while ${}^{\partial^2 f}/_{\partial \mathbf{u}\partial x_i}$ can be considered the derivative of ${}^{\partial f}/_{\partial \mathbf{u}}$ with respect to x_i and ${}^{\partial^2 f}/_{\partial x_i \partial x_j}$ is a single entry from the partial-derivative Hessian of of f with respect to $\mathbf{x}$ (which, again, is only non-zero when f depends directly on $\mathbf{x}$ in a way independent of $\mathbf{u}$). While the calculation of these quantities is laborious to implement, it is computationally tractable.

Next, ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}x_i}$ is the i^{th} column of ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ – note the repeated use of entries from this matrix. In the first-order case, computing ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ (via Equation 3.8) was considered too costly, as the complexity was $\mathcal{O}(N_x)$; however, as discussed, this is the same complexity one would already expect for a second-order adjoint method. Moreover, while the first-order instance of the matrix was projected onto a vector, in this instance every column is required – use of the entire matrix is unavoidable.

Once these first derivatives ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ are found from Equation 3.8, all that remains is to find the second derivatives ${}^{\mathrm{d}^2\mathbf{u}}/_{\mathrm{d}x_i\mathrm{d}x_j}$; here, just as in the first-order case, a substitution can be found by exploiting the fact that $\mathbf{r}$ is constant for converged $\mathbf{u}$, with zero first and second total derivative. Note here that, as ${}^{\partial^2\mathbf{r}}/_{\partial\mathbf{u}^2}$ is a rank-three tensor, a summation of matrix "slices" is required, but the second total derivative otherwise has the same form as that for f:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}x_i\mathrm{d}x_j} = \sum_{m=1}^{N_u}\left(\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial u_m}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j}\right) + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_j}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_i}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial\mathbf{r}}{\partial\mathbf{u}}\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}x_i\mathrm{d}x_j} + \frac{\partial^2\mathbf{r}}{\partial x_i\partial x_j} = 0 \tag{3.31}$$

$$\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}x_i\mathrm{d}x_j} = -\left(\frac{\partial\mathbf{r}}{\partial\mathbf{u}}\right)^{-1}\left(\sum_{m=1}^{N_u}\left(\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial u_m}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j}\right) + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_j}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_i}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2\mathbf{r}}{\partial x_i\partial x_j}\right) \tag{3.32}$$

Inserting this into Equation 3.30:

$$\begin{aligned}\frac{\mathrm{d}^2 f}{\mathrm{d}x_i\mathrm{d}x_j} =& \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i}^T\frac{\partial^2 f}{\partial\mathbf{u}^2}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 f}{\partial\mathbf{u}\partial x_j}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2 f}{\partial\mathbf{u}\partial x_i}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 f}{\partial x_i\partial x_j}\\ &- \frac{\partial f}{\partial\mathbf{u}}\left(\frac{\partial\mathbf{r}}{\partial\mathbf{u}}\right)^{-1}\left(\sum_{m=1}^{N_u}\left(\frac{\mathrm{d}u_m}{\mathrm{d}x_i}\frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial u_m}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j}\right) + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_j}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2\mathbf{r}}{\partial\mathbf{u}\partial x_i}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2\mathbf{r}}{\partial x_i\partial x_j}\right)\end{aligned} \tag{3.33}$$

A close examination reveals that the terms in parenthesis are premultiplied by the the same expression from Equations 3.9 - 3.11 used to build the first-order adjoint variables:

$$-\frac{\partial f}{\partial\mathbf{u}}\left(\frac{\partial\mathbf{r}}{\partial\mathbf{u}}\right)^{-1} = \left[-\left(\frac{\partial\mathbf{r}}{\partial\mathbf{u}}^T\right)^{-1}\frac{\partial f}{\partial\mathbf{u}}^T\right]^T = \boldsymbol{\lambda}^T \tag{3.34}$$

Thus, this most common type of Hessian adjoint method begins with N_x linear solves of the system

$$\frac{\partial\mathbf{r}}{\partial\mathbf{u}}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{x}} = -\frac{\partial\mathbf{r}}{\partial\mathbf{x}} \tag{3.35}$$

in order to compute each column of ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$, followed by a single solve of the first-order adjoint equation

$$\frac{\partial\mathbf{r}}{\partial\mathbf{u}}^T\boldsymbol{\lambda} = -\frac{\partial f}{\partial\mathbf{u}}^T \tag{3.36}$$

for a total cost proportional to $N_x + 1$ runs of the original problem. After the solves, the final Hessian can be computed by inserting ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ and $\boldsymbol{\lambda}$ into

$$\begin{aligned}\frac{\mathrm{d}^2 f}{\mathrm{d}x_i \mathrm{d}x_j} =& \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i}^T \frac{\partial^2 f}{\partial \mathbf{u}^2} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 f}{\partial \mathbf{u} \partial x_j} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2 f}{\partial \mathbf{u} \partial x_i} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 f}{\partial x_i \partial x_j} \\ & - \boldsymbol{\lambda}^T \left(\sum_{m=1}^{N_u} \left(\frac{\mathrm{d}u_m}{\mathrm{d}x_i} \frac{\partial^2 \mathbf{r}}{\partial \mathbf{u} \partial u_m} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} \right) + \frac{\partial^2 \mathbf{r}}{\partial \mathbf{u} \partial x_j} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_i} + \frac{\partial^2 \mathbf{r}}{\partial \mathbf{u} \partial x_i} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}x_j} + \frac{\partial^2 \mathbf{r}}{\partial x_i \partial x_j} \right)\end{aligned} \tag{3.37}$$

While the first-order adjoint method presented earlier only required the user (either manually or via an application of automatic differentiation to the relevant program sections) to provide two row vectors (${}^{\partial f}/_{\partial \mathbf{x}} \in \mathbb{R}^{N_x}$ and ${}^{\partial f}/_{\partial \mathbf{u}} \in \mathbb{R}^{N_u}$) and two matrices (${}^{\partial \mathbf{r}}/_{\partial \mathbf{u}} \in \mathbb{R}^{N_u \times N_u}$ and ${}^{\partial \mathbf{r}}/_{\partial \mathbf{x}} \in \mathbb{R}^{N_u \times N_x}$), this second-order method also requires three additional matrices (${}^{\partial^2 f}/_{\partial \mathbf{x}^2} \in \mathbb{R}^{N_x \times N_x}$, ${}^{\partial^2 f}/_{\partial \mathbf{u}^2} \in \mathbb{R}^{N_u \times N_u}$, and ${}^{\partial^2 f}/_{\partial \mathbf{u} \partial \mathbf{x}} \in \mathbb{R}^{N_u \times N_x}$) and three rank-three tensors (${}^{\partial^2 \mathbf{r}}/_{\partial \mathbf{x}^2} \in \mathbb{R}^{N_u \times N_x \times N_x}$, ${}^{\partial^2 \mathbf{r}}/_{\partial \mathbf{u}^2} \in \mathbb{R}^{N_u \times N_u \times N_u}$, and ${}^{\partial^2 \mathbf{r}}/_{\partial \mathbf{u} \partial \mathbf{x}} \in \mathbb{R}^{N_u \times N_u \times N_x}$). In many cases, a number of these terms will be zero (e.g., if some subsection of the program has been linearized), but this cannot be guaranteed in general. When they must be computed, the tensors might easily be too large to fit in memory, implying the need for a user-written function to build temporary subsets on the fly (e.g., the matrix slice ${}^{\partial^2 \mathbf{r}}/_{\partial \mathbf{u} \partial u_m}$ from the summation over m in Equation 3.37) even if such computations are redundant (e.g., repeated identically for all i, j in 3.37).

Beyond the burden of providing these quantities, as presented here (and in the broader literature) this approach suffers from poor performance when finding subsets of the Hessian. For a single scalar entry $H_{i,j}(\mathbf{x}) = {}^{\mathrm{d}^2 f}/_{\mathrm{d}x_i \mathrm{d}x_j}$, the method requires two columns of ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ and thus a total of three system solves, counting the adjoint; worse, at times a single entry may call for entire $N_u \times N_u$ residual derivative matrices or even a $N_u \times N_u \times N_u$ tensor. For a column of the Hessian, all of ${}^{\mathrm{d}\mathbf{u}}/_{\mathrm{d}\mathbf{x}}$ is required, and thus each of the $N_x + 1$ solves that would be required for the full Hessian.

While it is theoretically possible to re-derive an alternate approach that utilizes directional derivatives for efficient calculation of individual columns (or Hessian-vector products, which will be described in the next section), this author has yet to see such an approach.[4] Presenting a novel derivation is beyond the scope of this work, especially as the goal herein is to present easy-to-implement alternatives.

[4] Hessian-vector methods do exist for scalar AD; the discussion here is referring to matrix problems.

3.1.4 Directional Derivatives and Implicit Products

In 1993, Pearlmutter [19] presented an intrusive differential operator, intended to be paired with neural network backpropagation, that is capable of exactly finding the product $\mathbf{Hv}$ of the Hessian and a vector without ever building the full matrix. As already stated in Chapter 1, one way for engineers to understand the appeal of this simplification is to compare it to the matrix system solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Even though this expression includes $\mathbf{A}^{-1}$, such an inverse is usually too expensive to compute and is almost never found in its entirety. Instead, most solver algorithms can directly find the matrix-vector product without ever explicitly building the inverse.

Importantly, unlike the second-order adjoint method presented above, each use of this operator has the same order of complexity as the algorithm (e.g., for Pearlmutter, backprogagation) used to find the gradient. Also note that this capability includes as a subset the computation of a single column of $\mathbf{H}$, via the simple expedient of setting $\mathbf{v}$ to a unit vector in a cardinal direction.

Pearlmutter derived his operator using a Taylor series expansion similar to Equation 2.7, expressing the gradient near a known point $\mathbf{x}$ as

$$\nabla f(\mathbf{x} + \mathbf{\Delta x}) = \nabla f(\mathbf{x}) + \mathbf{H\Delta x} + \mathcal{O}(\|\mathbf{\Delta x}\|^2) \tag{3.38}$$

and solved for the Hessian-vector product $\mathbf{H\Delta x}$:

$$\mathbf{H\Delta x} = \nabla f(\mathbf{x} + \mathbf{\Delta x}) - \nabla f(\mathbf{x}) + \mathcal{O}(\|\mathbf{\Delta x}\|^2) \tag{3.39}$$

Setting $\mathbf{\Delta x} = h\mathbf{v}$ and solving for $\mathbf{Hv}$ then yields

$$\mathbf{Hv} = \frac{\nabla f(\mathbf{x} + h\mathbf{v}) - \nabla f(\mathbf{x})}{h} + \mathcal{O}(h) \tag{3.40}$$

which is a simple finite difference formula between two evaluations of the gradient. (This is the same approximation used by Andrei in [15].) To calculate precise values, Pearlmutter then defined the operator

$$\mathcal{R}_{\mathbf{v}}\left\{\nabla f(\mathbf{x})\right\} \equiv \frac{\partial}{\partial r}\left(\nabla f(\mathbf{x}_i + r\mathbf{v})\right)\Big|_{r=0} = \mathbf{H}\mathbf{v} \tag{3.41}$$

$$\mathcal{R}_{\mathbf{v}}\left\{g(\mathbf{y})\right\} \equiv \frac{\partial}{\partial r}\left(g(\mathbf{y}_i + r\mathbf{v})\right)\Big|_{r=0} \tag{3.42}$$

which in its general form (Equation 3.42) can be manually applied line-by-line to every instruction in existing back-propagation codes, or to other cases where intrusive implementation is feasible. For example, to apply the operator to a single scalar multiplication within a code:

$$\mathcal{R}_{\mathbf{v}}\left\{a(\mathbf{x})b(\mathbf{x})\right\} = \mathcal{R}_{\mathbf{v}}\left\{a(\mathbf{x})\right\}b(\mathbf{x}) + a(\mathbf{x})\mathcal{R}_{\mathbf{v}}\left\{b(\mathbf{x})\right\} \tag{3.43}$$

He then suggested multiple applications, including both the line-search update step from Equation 2.11 and a pairing with linear conjugate gradients to find the Newton update step, which of course forms the core of Hessian-free optimization from Section 2.1.3. (Note that these suggestions were phrased in terms of machine learning, rather than optimization in general.)

Since then, many AD tools have taken on the ability to reformulate problems in the same form as in Equation 3.41, extending the option for implicit (cheap) **Hv** to many other scalar codes. In fact, this could be considered an extension of a pre-existing AD capability (e.g., [42]) to implicitly find directional scalar first derivatives (gradient-vector products), directional gradients (Jacobian-vector products), or even Jacobians of multidimensional input subspaces (Jacobian-matrix products). Note that these approaches are unfortunately inapplicable to the adjoint method, as it is built upon an inversion of the purely internal Jacobian $^{\partial\mathbf{r}}/_{\partial\mathbf{u}}$, which is totally independent of the number of inputs or outputs.

3.2 Standard Automatic Differentiation

It is true that when high-precision derivatives are called for, at least some modification of a user's source code becomes necessary. Often, however, users do not wish to derive and implement these calculations manually, as in the preceding sections – in many practical applications, the implementation time would outweigh the usage time. This is the need served by *automatic differentiation* (AD) tools, of which there are simply too many to give a truly comprehensive survey here.[5] Traditionally, these tools use either automated source code modification (e.g., [1–3, 7]) or a new operator-overload data type (e.g., [4–6]), both of which approaches can come in forward- and reverse-mode (e.g., [72]) variants. These methods are, admittedly, sometimes less efficient than derivatives found via manual coding, but are instead designed with an eye toward fast (or, at worst, less-slow) implementation. Forward-mode variants, as expected, favor decreased implementation time at the cost of increased run time, while reverse-mode methods gain more efficient calculation by requiring more human effort (yet remain easier than manual reverse calculation).

AD packages are always built under the assumption that they will have access to a user's raw source code or the derivatives of user sub-functions; as a result, in most cases they are not designed with matrix operations in mind.[6] However, despite the matrix-focused nature of this work, a brief overview of standard AD approaches is helpful for the simple reason that matrix operations are almost never used in isolation. If derivatives are to be propagated across these operations, a traditional (scalar) AD tool must bridge the differentiation gap between program initialization and matrix function call and between matrix result and program output. This section will focus solely on first-order, single-derivative (or, in the case of reverse mode, single-gradient) methods; once this foundation is complete, the jump to higher-order multivariate derivatives in Section 3.3 will be relatively simple.

The concept of using one program to automatically differentiate another dates back to at least Beda *et al*'s work in 1959 ([73], as cited in [72]). In the early decades of the field, most of the work focused on the forward mode of calculation; in many cases, the process was symbolic, yielding at program's end a (typically extremly long and expensive) algebraic expression for the derivative. Numeric methods that simply applied the chain rule appear to have come into prominence in the 1980s [72], along with the introduction of the reverse mode of automatic computation [72, 74].

Most AD theory uses a simplified view of underlying programs, expressing them as a linear sequence of many commands; such a stylized program is shown in the pseudocode of Algorithm 3.1, along with detailed comments. It

[5] The AD community portal www.autodiff.org lists 30 tools for C/C++, 15 for Fortran77, and 6 for MATLAB. [71]

[6] Several exceptions to this rule will be discussed in Section 3.4; in brief, they amount to matrix packages built with AD in mind, rather than the other way around.

is perhaps most intuitive to demonstrate the various AD approaches by showing their modifications to this program – hence the color-coding to indicate the original source code, which will be contrasted with new or modified AD code.

Algorithm 3.1 Pseudocode model for programs to which typical AD is applied. Blue shading indicates external interface for the function or program; gray shading indicates open-source user code.

```
   Define:    $\mathbb{S}$                                   // Set of scalar data types         e.g., float, complex<double>
              $g_i(\cdot) : \mathbb{S}^{N_i} \to \mathbb{S}$   // Command at step $i$              e.g., a1=x1*x2; a2=exp(x3);
1. Given:     $\mathbf{x} \in \mathbb{S}^{N_x}$              // Input parameters                 e.g., f(double xvec[5], float x6);
2. Execute:   $\mathbf{y^{(1)}} \leftarrow \mathbf{x}$                          // Initialize set of internal program variables
              for $i = 1 \ldots N_C$:                                          // Program expressed as a sequence of commands
                  $a_i \leftarrow g_i(\mathbf{z^{(i)}}) : \mathbf{z^{(i)}} \subseteq \mathbf{y^{(i)}}$   // Function of some subset of previous results
                  $\mathbf{y^{(i+1)}} \leftarrow \{\mathbf{y^{(i)}}, a_i\}$      // Add result to internal set
3. Output:    $f(\mathbf{x}) \leftarrow a_{N_C} \in \mathbb{S}$   // Quantity of Interest
```

3.2.1 Forward-Mode AD

The simplest mode of AD computes, in sequence, the derivative of each intermediate variable a_i with respect to one or more inputs (i.e., independent differentiation variables x_j) via repeated application of the chain rule. For each desired derivative ${}^{\mathrm{d}(\cdot)}/_{\mathrm{d}x_j}$, a set of new variables are introduced – in the single-derivative case, one new $\dot{a}_i$ for each a_i in the original program. The first such variables are the derivatives of the differentiated inputs with respect to themselves – i.e., $\dot{x}_j = {}^{\mathrm{d}x_j}/_{\mathrm{d}x_j} = 1$. In the cases of undifferentiated inputs, the paired initial derivative variables are set to zero – i.e., no inputs are assumed to be functions of each other, so that $\dot{x}_k = {}^{\mathrm{d}x_k}/_{\mathrm{d}x_j} = 0$ for $k \neq j$. The chain rule applied to the first user operation amounts to:

$$\dot{a}_1 = \frac{\mathrm{d}}{\mathrm{d}x_j} g_1(x_j) \tag{3.44}$$

$$= \frac{\partial g_1}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}x_j}\bigg|_{y=x_j} \tag{3.45}$$

$$= \frac{\partial g_1}{\partial x_j} \dot{x}_j \tag{3.46}$$

The calculation is repeated for all following operations:

$$\dot{a}_{i+1} = \frac{\mathrm{d}}{\mathrm{d}x_j} g_{i+1}(g_i(\cdot)) \tag{3.47}$$

$$= \frac{\partial g_{i+1}}{\partial g_i} \frac{\mathrm{d}g_i}{\mathrm{d}x_j} \tag{3.48}$$

$$= \frac{\partial g_{i+1}}{\partial g_i} \dot{a}_i \tag{3.49}$$

When implemented explicitly, this algorithm takes the form shown in Algorithm 3.2, which also allows for multiple inputs to each intermediate operation g_i. Note that explicit implementation is exactly the approach taken by source-code modification tools; yellow highlighting in the pseudocode indicates new commands automatically inserted by an AD tool.

Algorithm 3.2 Pseudocode for forward-mode automatic differentiation via the source-modification approach. Blue shading indicates external interface for the function or program; gray shading indicates open-source user code; yellow shading indicates new code generated by AD. Every line of user code is followed by one or more lines of generated AD code.

```
   Define:   $\mathbb{S}$                                   // Set of scalar data types      e.g., float, complex<double>
             $g_i(\cdot) : \mathbb{S}^{N_i} \to \mathbb{S}$   // Command at step $i$          e.g., a1=x1*x2; a2=exp(x3);
1. Given:    $\mathbf{x} \in \mathbb{S}^{N_x}$              // Input parameters              e.g., f(double xvec[5], float x6);
             $j \in 1 \ldots N_x$                            // Index of target derivative    i.e., find ${}^{\mathrm{d}f}/_{\mathrm{d}x_j}$
2. Execute:  $\mathbf{y}^{(1)} \leftarrow \mathbf{x}$                    // Initialize set of internal program variables
             $\dot{\mathbf{y}}^{(1)} \leftarrow \mathbf{0}$              // Initialize set of derivatives w/r/t $x_j$
             $\dot{y}_j^{(1)} \leftarrow 1$                              // Derivative of $x_j$ w/r/t to itself is 1
             for $i = 1 \ldots N_C$:                                     // User program expressed as a sequence of commands
                 $a_i \leftarrow g_i(\mathbf{z}^{(i)}) : \mathbf{z}^{(i)} \subseteq \mathbf{y}^{(i)}$   // Function of some subset of previous results
                 $\dot{a}_i \leftarrow \sum_k (\dot{z}_k^{(i)})({}^{\partial g_i}/_{\partial z_k^{(i)}})$   // Apply chain rule (total derivative) to user code
                 $\mathbf{y}^{(i+1)} \leftarrow \{\mathbf{y}^{(i)}, a_i\}$              // Add result to internal set
                 $\dot{\mathbf{y}}^{(i+1)} \leftarrow \{\dot{\mathbf{y}}^{(i)}, \dot{a}_i\}$   // Add derivative of result to internal set
3. Output:   $f(\mathbf{x}) \leftarrow a_{N_C} \in \mathbb{S}$                 // Quantity of Interest
             ${}^{\mathrm{d}f}/_{\mathrm{d}x_j} \leftarrow \dot{a}_{N_C} \in \mathbb{S}$   // Derivative of QoI
```

3.2.2 Reverse-Mode AD

A derivation[7] of the reverse mode can begin with the observation that, for the sequence of operations shown in Algorithm 3.1, the whole function $f(x_j)$ can be expressed as a function of any intermediate set of internal variables $\mathbf{y^{(i)}}(x_j)$:

$$f(x_j) = f\left(\mathbf{y^{(i)}}(x_j)\right) \tag{3.50}$$

$$\mathbf{y^{(i)}} = [a_1, a_2, \dots, a_i] \tag{3.51}$$

$$a_n = g_n(\mathbf{y^{(n)}}) \tag{3.52}$$

Consider the chain rule for another intermediate variable a_k which came before step i:

$$\frac{\partial f}{\partial a_k} = \sum_{m=k}^{i-1} \frac{\partial f}{\partial a_m} \frac{\partial g_m}{\partial a_k} \tag{3.53}$$

Define this derivative as an *adjoint variable* $\bar{a}_k$. This yields the inductive relation

$$\bar{a}_k = \sum_{m=k}^{i-1} \bar{a}_m \frac{\partial g_m}{\partial a_k} \tag{3.54}$$

In contrast to the forward-mode variables $\dot{a}_k$, which represented the cumulative impact of all preceding calculations on the derivative a_k, the adjoint variable $\bar{a}_k$ represents the cumulative impact of a_k on *upcoming* calculations. More than that, if the future adjoint variables $\bar{a}_m$ are known, $\bar{a}_k$ can be expressed in terms that do not rely on the output f directly. Noting that the derivative of the final atomic operation $f = a_{N_C}$ with respect to itself is one, it can be seen that one can simply set the final adjoint $\bar{a}_{N_C} = 1$ and calculate every other adjoint variable in reverse order. After every operation stretching back to g_1 has been differentiated, one will have obtained the adjoints of every input variable – i.e., the entire gradient of $f(\mathbf{x})$. Pseudocode for this process is shown in Algorithm 3.3.

[7] This derivation generally follows the logic and notation from [72].

The reverse mode is quite elegant, in that it can compute an arbitrarily-large gradient with only one new adjoint operation per original (forward-problem) operation – indeed, as shown in [72], even a very simple automatic implementation is guaranteed to have a cost less than five times that of the original program. However, the approach suffers from a major drawback: the derivatives ${}^{\partial g_m}/_{\partial a_k}$ in Equation 3.54 are functions of the forward variables $\mathbf{y^{(m)}}$ – in other words, every single intermediate variable from the forward problem must be saved after its initial calculation. While the stylized program in Algorithm 3.1 was represented as storing the entire vector $\mathbf{y^{(N_C)}}$, this was only shown for the sake of analytic simplicity; for programs of significant size, it can quickly become impractical. Nevertheless, reverse differentiation can still be put to important use in engineering applications – e.g., by differentiating smaller sub-functions to build the intermediate Jacobians and gradients needed by the (matrix) adjoint method discussed in Section 3.1.2 and Section 3.1.3.

Algorithm 3.3 Pseudocode for reverse-mode automatic differentiation. Light blue shading indicates external interface for the function or program; gray shading indicates open-source user code; yellow shading indicates new code generated by AD, which generates the entire gradient at the cost of storing all forward variables (see text).

```
   Define:     $\mathbb{S}$                          // Set of scalar data types         e.g., float, complex<double>
               $g_i(\cdot) : \mathbb{S}^{N_i} \to \mathbb{S}$        // Command at step $i$                e.g., a1=x1*x2; a2=exp(x3);
1. Given:      $\mathbf{x} \in \mathbb{S}^{N_x}$                   // Input parameters                 e.g., f(double xvec[5], float x6);
2. Execute:    $\mathbf{y^{(1)}} \leftarrow \mathbf{x}$                          // Initialize set of internal program variables
               for $i = 1 \dots N_C$:                    // Program expressed as a sequence of commands
                   $a_i \leftarrow g_i(\mathbf{z^{(i)}}) : \mathbf{z^{(i)}} \subseteq \mathbf{y^{(i)}}$   // Function of some subset of previous results
                   $\mathbf{y^{(i+1)}} \leftarrow \{\mathbf{y^{(i)}}, a_i\}$          // Add result to internal set
3. Output:     $f(\mathbf{x}) \leftarrow a_{N_C} \in \mathbb{S}$              // Quantity of Interest
4. Store:      $\mathbf{y^{(N_C)}} \in \mathbb{S}^{N_C+N_x}$             // All internal variables must be retained
5. Reverse:    $\bar{\mathbf{y}} \leftarrow \mathbf{0} \in \mathbb{S}^{N_C+N_x}$                 // Initalize adjoints of all variables
               $\bar{y}_{\mathrm{end}} \leftarrow 1$                            // Adjoint of output is 1
               for $i = N_C, \dots 1$:                   // Each user command examined in reverse
                   $\forall z_k^{(i)} \in \mathbf{z^{(i)}}$ where $z_k^{(i)} = y_m^{(i)}$     // For each forward variable $z \in \mathbf{y}$ used as input to $g_i$
                       $\bar{y}_m \leftarrow \bar{y}_m + \bar{y}_i({}^{\partial g_i(\mathbf{z^{(i)}})}/_{\partial z_k^{(i)}})$  // Piece-wise summation of chain rule
6. Output:     $\nabla f \leftarrow [\bar{y}_1, \dots, \bar{y}_{N_x}]$          // Gradient of $f(\mathbf{x})$
```

3.2.3 Automatic Source Modification vs Operator Overloading

Two broad approaches exist to implement either the forward- or reverse-mode methodology – namely, source modification and operator overloading. In the first approach, another program is used to process user source code and generate a new program with additional code; in the second, a new data type is introduced to replace real-valued variables in the user program.

The two methods have different advantages. Source modification can be more efficient at run-time because the generating program can take a "long view" of many operations at once, searching for possible shortcuts, whereas operator overloading must work naively on an element-by-element basis – after all, real-world programs do not actually have the daisy-chain sequential dependency of Algorithm 3.1; instead, the dependency of one function upon another is better represented as a directed acyclic graph (DAG), which opens up an entire field of optimization approaches rooted in graph theory. This long-view feature can also make for more parallelizable code. Finally, older programming languages such as Fortran77 do not even support new data types. The source-modification approach generally follows the explicit forward or reverse algorithms already shown in Algorithm 3.2.

Operator-overloaded types, in contrast, lean even harder into the "automatic" part of AD, trading some efficiency for even easier implementation. All or most differentiation is encapsulated away from the user's sight; every time the original code performs an operation, the overloaded variables used by the operation compute the relevant derivative formula, the results of which are stored alongside the operation's output variable(s). This encapsulation also makes AD easier to use during code development – although source modification is automatic, if the user wishes to change his or her underlying code, the AD tool must be run again after every change. In the forward mode, operator overloading can be represented as shown in Algorithm 3.4.

The overloading approach can also implement reverse-mode differentiation, in a way almost identical to Algorithm 3.3. In this mode, the forward sweep of the program still uses overloaded types; however, rather than calculate derivatives, the overloaded variables $\hat{a}_i$ save the real values a_i to hard-disk along with a label to identify each operation $g_i(\cdot)$. The collected set of program data, known as a "tape", is then used to run the reverse problem.

In this work, it is generally assumed (in pseudocode and elsewhere) that scalar user code is differentiated with a forward-mode overloaded type, as it has a similar "set it and forget it" philosophy to NMD. However, there is no reason that source-modification or reverse tools cannot be used to differentiate non-matrix sections of a program.

Algorithm 3.4 Pseudocode for forward-mode automatic differentiation via operator overloading. Light blue shading indicates external interface for the function or program; gray shading indicates open-source user code; yellow shading indicates new data types or new code generated by AD. By defining and initializing new datatypes, the user source is left relatively intact.

```
Define:     $\mathbb{S}$                                  // Set of scalar datatypes                  e.g., float, std::complex<double>
            $\mathbb{A} \equiv \mathbb{S} \times \mathbb{S}$                     // Set of AD data types                     e.g., ADOLC::adouble [75]
            $\hat{v} \in \mathbb{A}$                               // Overloaded instance of variable $v$       i.e., $\hat{v} \equiv \{v, \dot{v}\}$
            $g(\cdot) : \mathbb{S}^{N_i} \to \mathbb{S}$                   // Original command                         e.g., a1=x1*x2; a2=exp(x3);
            $\hat{g}(\cdot) : \mathbb{A}^{N_i} \to \mathbb{A}$                   // Command overloaded w/ chain rule         i.e., $\hat{g}(\mathbf{v}) \equiv \{g(\mathbf{v}), \sum_k (\dot{z}_k^{(i)})(\partial g_i / \partial z_k^{(i)})\}$;
1. Given:   $\mathbf{x} \in \mathbb{S}^{N_x}$                            // Input parameters                         e.g., f(double xvec[5], float x6);
            $j \in 1...N_x$                          // Index of target derivative               i.e., find $\mathrm{d}f / \mathrm{d}x_j$
2. Execute: $\hat{\mathbf{y}}^{(1)} \leftarrow [\mathbf{x}, \mathbf{0}]$                           // Initialize overloaded internal program variables
            $\hat{y}_j^{(1)} \leftarrow [x_j, 1]$                          // Derivative of $x_j$ w/r/t to itself is 1
            for $i = 1 \ldots N_C$:                              // User program expressed as a sequence of commands
                $\hat{a}_i \leftarrow \hat{g}_i(\hat{\mathbf{z}}^{(i)}) : \hat{\mathbf{z}}^{(i)} \subseteq \hat{\mathbf{y}}^{(i)}$          // Function of some subset of previous results
                $\hat{\mathbf{y}}^{(i+1)} \leftarrow \{\hat{\mathbf{y}}^{(i)}, \hat{a}_i\}$                // Add result to internal set
            $\hat{f} \leftarrow \hat{a}_{N_C} \in \mathbb{A}$                       // Overloaded Quantity of Interest
3. Output:  $f(\mathbf{x}) \leftarrow f_0 \in \mathbb{S}$                  // Base (Undifferentiated) QoI
            $\mathrm{d}f / \mathrm{d}x_j \leftarrow f_1 \in \mathbb{S}$                // Derivative of QoI
```

3.3 Multivariate and High-Order AD

The preceding section introduced the basic form of standard AD tools using only a single new variable (i.e., a forward derivative or adjoint) per original variable, allowing for the calculation of first derivatives. This work, however, is focused on finding the Hessian and higher derivative tensors. These, too, can be found by existing AD tools (in non-matrix cases) via various modifications discussed below. These include vectorization, to find additional derivatives at once, and simplistic nesting of AD tools, to find higher derivatives. Another, more advanced method for arbitrary-order differentiation that is more efficient than simple nesting will be saved for Section 4.5, as it is built around a special class of algebras.

3.3.1 Vectorization

The examples above were limited to one derivative at a time, thereby forcing users to run AD once for each input variable (or, in the case of reverse mode, once for each output). It is perhaps intuitive to avoid this inconvenience by adding more AD variables, computing multiple derivatives at once; instead of a single variable ($\dot{a}$ or $\bar{a}$), the modified program would use an entire vector ($\dot{\mathbf{a}} = \nabla a$ or $\bar{\mathbf{a}} = \partial \mathbf{f} / \partial a$) of variables at each point in the code. In many cases, however, the cost savings of this vectorization approach may not be worth the back-end implementation effort or the extra memory burden – usually, even in the limit of very many derivatives at once, one can save at most 50% of the time that would otherwise be spent going one at a time. There are exceptions to this rule: programs burdened with a large amount of computation that is not differentiated. For example, if a program must spend a significant amount of time loading constant, input-independent data from disk, it may make sense to avoid doing this more than once. A plot of relative cost savings for various fractions of such fixed overhead can be seen in Figure 3.2. Note that an AD-enabled program with this problem could almost certainly be improved simply by running the differentiation in a loop across only those subroutines that require it. Sometimes, though, vectorization might be the path of least resistance, especially when there is enough memory to spare.

Beyond the occasional opportunity for cost savings, there is one other situation where multiple simultaneous derivatives are called for: the calculation of mixed higher-order derivatives, which requires access to lower-order factors. This is especially evident in the case of inductive AD – the simplest (and most simplistic) higher-order methodology.

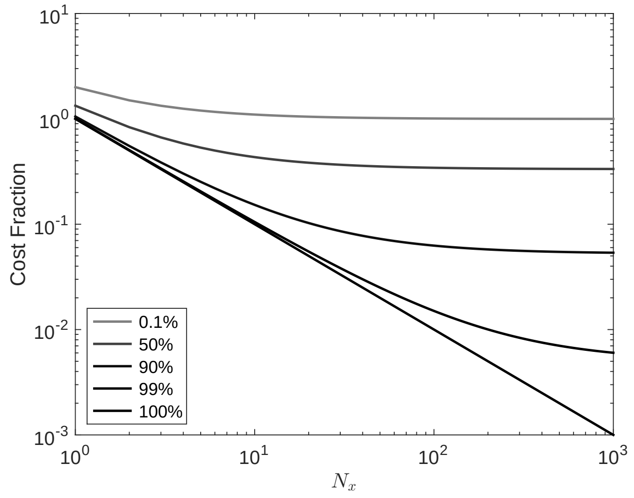


Figure 3.2: Vectorization Savings vs Fixed Overhead: Cost fraction of a single vectorized run vs many single-derivative runs, plotted logarithmically against number of vectorized inputs. If differentiated code is a relatively small portion of the total program cost, vectorization can avoid redundant computation of the fixed overhead. The legend shows relative cost of overhead compared to total program cost; in the limit of 100% overhead, a vectorized run has a cost $1/N_x$ that of an equivalent number of singleton runs.

3.3.2 Naive Induction

Inductive AD is the intuitive approach to taking second- and higher-order derivatives: apply AD once, then apply it again. This can take the form of applying a source-modification tool to an already-modified code, or of replacing the real numbers in the back-end of an overloaded type with yet another overloaded type. This approach, unfortunately, suffers from several significant inefficiencies.

Mathematically, for a univariate second derivative of a variable y, the most literal form of induction can be expressed in operator-overloaded notation by introducing a second-order AD variable $\hat{\hat{y}}$, built by augmenting a first-order variable $\hat{y}$:

$$\dot{y} \equiv \frac{\mathrm{d}}{\mathrm{d}x_j}\left[y\right] \tag{3.55}$$

$$\hat{y} \equiv y + \dot{y} \tag{3.56}$$

$$\begin{aligned} \dot{\hat{y}} &\equiv \frac{\mathrm{d}}{\mathrm{d}x_j}\left[\hat{y}\right] \qquad (3.57) \\ &= \dot{y} + \ddot{y} \end{aligned}$$

$$\hat{\hat{y}} \equiv \hat{y} + \dot{\hat{y}} \tag{3.58}$$

$$= (y + \dot{y}) + (\dot{y} + \ddot{y}) \tag{3.59}$$

In both the overloaded and source-modification approach, two copies of the first derivative of y are carried. This is mirrored in the computation of values; for example, the inductive approach does not account for Leibniz's rule (i.e., the product rule for higher orders) – instead, it repeatedly calculates every permutation of a derivative term. For the second univariate derivative of $y(x)z(x)$, it would compute $(yz'' + y'z') + (y'z' + y''z)$, rather than saving a step and finding $yz'' + 2y'z' + y''z$. This also occurs in the multivariate (vectorized) case, where the entire first-order vector is calculated twice. In the case of higher tensors, the redundancy is given by binomial coefficients – e.g., for third derivatives, there will be three repeated calculations (and three copies) of the gradient and three of the Hessian. Almost as bad, naive induction does not account for symmetry of the Hessian: rather than limiting itself to the $N(1 + N)/2$ unique second derivatives, it redundantly computes all N^2 entries. The redundancy again worsens for higher tensors, which have more symmetries. These redundancies can be removed easily; however, this presumes the creators of the AD method anticipate a need for higher derivatives. In many cases, they do not.

In one of the first methods designed with higher orders in mind, [76] a tree structure was used to store the derivatives of derivatives, thereby allowing for sparse tensors, but much of the other computational redundancy remained. Further, simply in terms of computer architecture, a tree structure can become very inefficient for tensors in many variables – it is far better to keep sequential indices in similarly sequential memory, leading to faster cache operation. Due to inefficiencies such as these, later work moved from inductive implementation to inductive *derivation*; some methods (e.g., [77]) carefully calculated Leibniz's rule (or, for other operations, the relevant time-saving coefficients) and utilized elaborate indexing schemes for data stored in contiguous memory. More recent advances (e.g., [78]) abandon induction entirely in favor of the algebra of Taylor polynomials, as will be discussed in Section 4.5.

It is also worth mentioning that, if all one needs is the Hessian, a number of sophisticated methods exist for automatically finding a sparsity pattern (e.g., [79]) and, based on that pattern, finding the most efficient subspace (e.g., [80]) to find non-zero entries via a matrix of smaller dimension – an approach that is effectively a higher-dimensional case of the directional derivatives described in Section 3.1.4. These tools are arguably distinct from differentiation itself, as they amount to pre-processing before true AD begins, and will not be further considered in this work.

3.4 Matrix Operations and AD

So far within this chapter, only the non-automatic (finite difference and adjoint) methods have been capable of finding derivatives of matrix functions. Unfortunately, many (if not most) large-scale engineering simulations or optimizations rely heavily on such functions, typically supplied via third-party numerical libraries. Such libraries typically have a great many source files (e.g., LAPACK 3.6.0 has 5,830 files [81]), in which case even allowing new data types – let alone full source modification – becomes laborious to the point of impracticality. Further, the best library binaries (e.g., many implementations of the BLAS specification [82–84]) are often optimized by hand for individual machine architectures; even if the (often proprietary) source is available, re-compiling from scratch might negate this advantage.

It is therefore fair to claim that in general, if any user program written in a low-level language calls a third-party matrix library, external AD tools are no longer a valid option for scalar-by-scalar differentiation; for practical differentiation applications, any method that can use linear algebra routines as-is would be vastly preferable. However, in this claim the terms “low-level language” and “external” are carrying special weight.

Before addressing these qualifications, note that formulas certainly do exist for derivatives of whole matrices – see for example [85] or [20]. Take, as two important examples, matrix multiplication and matrix-vector system solution. If, given a matrix $\mathbf{A}$ from the original problem, one defines a derivative matrix $\dot{\mathbf{A}}$ such that $\dot{A}_{i,j} = {}^{\mathrm{d}A_{i,j}}/_{\mathrm{d}x}$ for every entry (i, j), then the matrix product rule is derived as:

$$\mathbf{C} = \mathbf{AB} \tag{3.60}$$

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \tag{3.61}$$

$$\dot{C}_{i,j} = \sum_k A_{i,k} \dot{B}_{k,j} + \dot{A}_{i,k} B_{k,j} \tag{3.62}$$

$$\dot{\mathbf{C}} = \dot{\mathbf{A}}\mathbf{B} + \mathbf{A}\dot{\mathbf{B}} \tag{3.63}$$

which has the same form as the scalar product rule. System solution, however, has a form different from the quotient rule. It is derived by beginning with a standalone inverse:

$$\mathbf{C} = \mathbf{A}^{-1} \tag{3.64}$$

$$\mathbf{AC} = \mathbf{I} \tag{3.65}$$

$$\dot{\mathbf{A}}\mathbf{C} + \mathbf{A}\dot{\mathbf{C}} = \mathbf{0} \tag{3.66}$$

$$\dot{\mathbf{C}} = -\mathbf{A}^{-1}\dot{\mathbf{A}}\mathbf{C} \tag{3.67}$$

$$= -\mathbf{A}^{-1}\dot{\mathbf{A}}\mathbf{A}^{-1} \tag{3.68}$$

A system solution is then the product of the inverse with a vector:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{3.69}$$

$$\dot{\mathbf{x}} = \left(-\mathbf{A}^{-1}\dot{\mathbf{A}}\mathbf{A}^{-1}\right)\mathbf{b} + \mathbf{A}^{-1}\dot{\mathbf{b}} \tag{3.70}$$

$$= \mathbf{A}^{-1}\left(\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x}\right) \tag{3.71}$$

In other words, to find a single first derivative of a linear solve, one first solves the original problem, then computes $\dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x}$ and solves a second system for $\dot{\mathbf{x}}$.

Similar formulas exist for many other matrix operations; if one wishes to manually compute matrix derivatives, this is the way to go about it. Further, as discussed in the same sources above, adjoint variants of matrix operations exist, allowing for reverse-mode computation if enough memory exists to store every intermediate matrix. Considering that these formulas exist, and returning to the earlier qualifying terms "low-level language" and "external", it becomes very important to acknowledge that in a *high-level* language such as MATLAB, which treats matrices as encapsulated objects (rather than a collection of individually-defined memory pointers and other parameters, such as in BLAS/LAPACK), standard AD approaches can and have been implemented – e.g., for MATLAB, the most popular tool is ADiMat ([86, 87]), which comes with some limited higher-order functionality.

Likewise, while "external" AD implementation for low-level libraries is cumbersome or impossible, certain numerical computation packages are, in fact, *internally* designed by their authors with AD at least partly in mind. For example, the Trilinos Project [88] from Sandia National Labs provides the Sacado [89] AD package, capable of differentiating Trilinos-specific matrices. In the unique case of the Eigen [90] matrix library (written in C++ with heavy use of modern templating features), users can actually supply custom data-types – including overloaded AD variables – with which to build matrices and vectors. Eigen is open-source and designed to be recompiled on a program-by-program basis; as such, it can even take special steps at compile time to optimize low-level operations (e.g., memory addressing to improve cache performance, etc) around a given type. To the author's knowledge, this makes Eigen the single exception to the general rule claimed above: it is the only practical and even halfway efficient way to use standard AD tools to differentiate matrices element-by-element, rather than with whole-matrix formulas.

There is one obvious drawback to automatic differentiation via any of these tools: *AD users must design their* *entire program around a specific library and in a specific language, from the ground up*. Looking ahead to Part II, NMD is designed to fill the gaps between these cases, providing a method that can work in any context and any language.

Before beginning derivation of the new method, however, it is necessary to lay some groundwork by reviewing several differentiation approaches that do *not* use derivative formulas directly; instead, they utilize algebras other than that of the real numbers. The first of these approaches, the complex-step method, is even compatible with forward-mode first derivatives of matrix functions; the others all help put the upcoming derivation into context.

Chapter 4

Special Algebras for Differentiation

All the methods presented in Chapter 3 utilized simple freshman calculus to find derivatives. There are, however, other approaches – non-real algebras which can find derivatives either as accidental side-effects or through clever design. One of the most important reasons to review these methods, as far as this work is concerned, is the fact that any of the non-real numbers considered here can be represented as matrices of real numbers held in specific patterns; these representations will be saved for Chapter 6. For now, it is necessary to see how these systems work for the differentiation of normal scalar code – and, in the case of the first method, for the differentiation of matrices.

4.1 Complex-Step Differentiation (CD)

As discussed in the previous chapter, automatic derivatives of matrix functions cannot (usually) be found by back-end operator overloading or source modification. There is, however, one potential exception for real-valued problems: many libraries already come standard with function variants utilizing the arithmetic of complex numbers. In many cases, a switch from real to complex data types can in fact serve as a form of AD.

4.1.1 History and Derivation

Any discussion of derivatives involving complex numbers would be incomplete without a mention of Cauchy's integral formula, which states that any univariate derivative of a complex point can be defined in terms of a contour integral enclosing that point on the complex plane:

$$f^{(n)}(z_0) = \frac{n!}{2\pi i} \oint \frac{f(z)}{(z - z_0)^{n+1}} dz \tag{4.1}$$

Indeed, the first use of complex numbers for numerical differentiation began with this equation, in the 1967 work of Lyness and Moler [62] who introduced a number of (quite complicated)[1] integration formulas based on the trapezoid rule. Like the Cauchy equation itself, these methods were all designed with derivatives of arbitrary order in mind, and as the trapezoid rule does not require the subtraction of nearly-equal numbers, these formulas were capable of finding derivatives to very high precision – albeit only in the limit of many function evaluations. For example, in their example problem they found a fifth derivative of a real-valued function to nine significant figures via 59 complex evaluations, and to ten significant figures via 1097 evaluations.

In 1998, Squire and Trapp [92] revisited the methods of Lyness and Moler and presented a vastly-simplified, single-evaluation variant limited to first derivatives:

$$\frac{d}{dx}\big(f(x)\big) \approx \frac{1}{h}\mathrm{Im}\big(f(x + ih)\big) \tag{4.2}$$

where the approximation holds in the limit of small h. This formula has since been termed the *complex-step derivative approximation*, or CD. The accuracy of the formula was proven with a simple Taylor series expansion:

$$\begin{aligned}
f(x + ih) &= f(x) + \frac{hi}{1!}\frac{df(x)}{dx} + \frac{h^2 i^2}{2!}\frac{d^2 f(x)}{dx^2} + \frac{h^3 i^3}{3!}\frac{d^3 f(x)}{dx^3} + \dots &(4.3)\\
&= f(x) + hi\frac{df(x)}{dx} - \frac{h^2}{2}\frac{d^2 f(x)}{dx^2} - \frac{h^3 i}{6}\frac{d^3 f(x)}{dx^3} + \dots &(4.4)\\
\mathrm{Im}\big(f(x + ih)\big) &= h\frac{df(x)}{dx} - \frac{h^3}{6}\frac{d^3 f(x)}{dx^3} + \dots &(4.5)\\
\frac{df(x)}{dx} &= \frac{1}{h}\mathrm{Im}\big(f(x + ih)\big) + \mathcal{O}\big(h^2\big) &(4.6)
\end{aligned}$$

Note that CD requires only a single function evaluation and can avoid subtractive cancellation error entirely;

[1] In a 2013 blog post [91] for MathWorks, Moler himself stated "[...] the actual algorithms are pretty exotic. [...] If you asked me to describe them in any detail today, I wouldn't be able to."

therefore, for very small h (e.g., 10^{-20} or smaller) it can yield results that are accurate to within a few bits of machine precision – for double-precision floating-point arithmetic, this can translate to 14 or 15 significant figures.

The CD equation (4.2) can be interpreted in several ways. The first is as a two-point trapezoidal integration of the parameterized Cauchy formula with $z_0 = x$ and $z = x + ihe^{2\pi it}$:

$$f'(x) = \frac{1}{2\pi i}\int_0^1 \frac{f\left(x+ihe^{2\pi it}\right)}{\left(ihe^{2\pi it}\right)^2}\left(-2\pi h\right)e^{2\pi it}dt \tag{4.7}$$

$$= \frac{1}{hi}\int_0^1 \frac{f\left(x+ihe^{2\pi it}\right)}{e^{2\pi it}}dt \tag{4.8}$$

If f is analytic, the direction of the derivative is irrelevant and one can set $f'(x) = {}^{\partial f}/_{\partial x}$. Then, for $f(z) = u+vi$,

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial x}i = \frac{1}{hi}\int_0^1 \frac{f\left(x+ihe^{2\pi it}\right)}{e^{2\pi it}}dt \tag{4.9}$$

$$\frac{\partial u}{\partial x}i - \frac{\partial v}{\partial x} = \frac{1}{h}\int_0^1 \frac{f\left(x+ihe^{2\pi it}\right)}{e^{2\pi it}}dt \tag{4.10}$$

$$\frac{\partial u}{\partial x} = \frac{1}{h}\mathrm{Im}\left(\int_0^1 \frac{f\left(x+ihe^{2\pi it}\right)}{e^{2\pi it}}dt\right) \tag{4.11}$$

Note that a two-point trapezoid rule is simply an average of the endpoints – yet evaluating this integral only at $t = 0$ and $t = 1$ sends $e^{2\pi it}$ to one, causing the points to coincide. Thus, the supposed "trapezoid integration" only requires a single evaluation and takes the same form as Equation 4.2. This is of course a very imprecise approximation of the integral, but in this particular case it becomes precise in the limit of a tight contour (corresponding to a small h).

Another interpretation of Equation 4.6 is as the definition of the derivative for a complex analytic function. As discussed in [63], if $f(z)$ is analytic, $z = x + yi$ and $f(z) = u + vi$, then the Cauchy-Riemann equations state:

$$i\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \tag{4.12}$$

$$\therefore \qquad \frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{4.13}$$

$$\text{and} \qquad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x} \tag{4.14}$$

These relationships can be visualized as in Figure 4.1 using overlaid contour plots of u and v. Note there that the real- and imaginary-valued gradients (and hence contours) are perpendicular to each other; further note that the magnitudes of the gradients are identical at any given point, which can be seen by identical contour spacing at highest resolution.

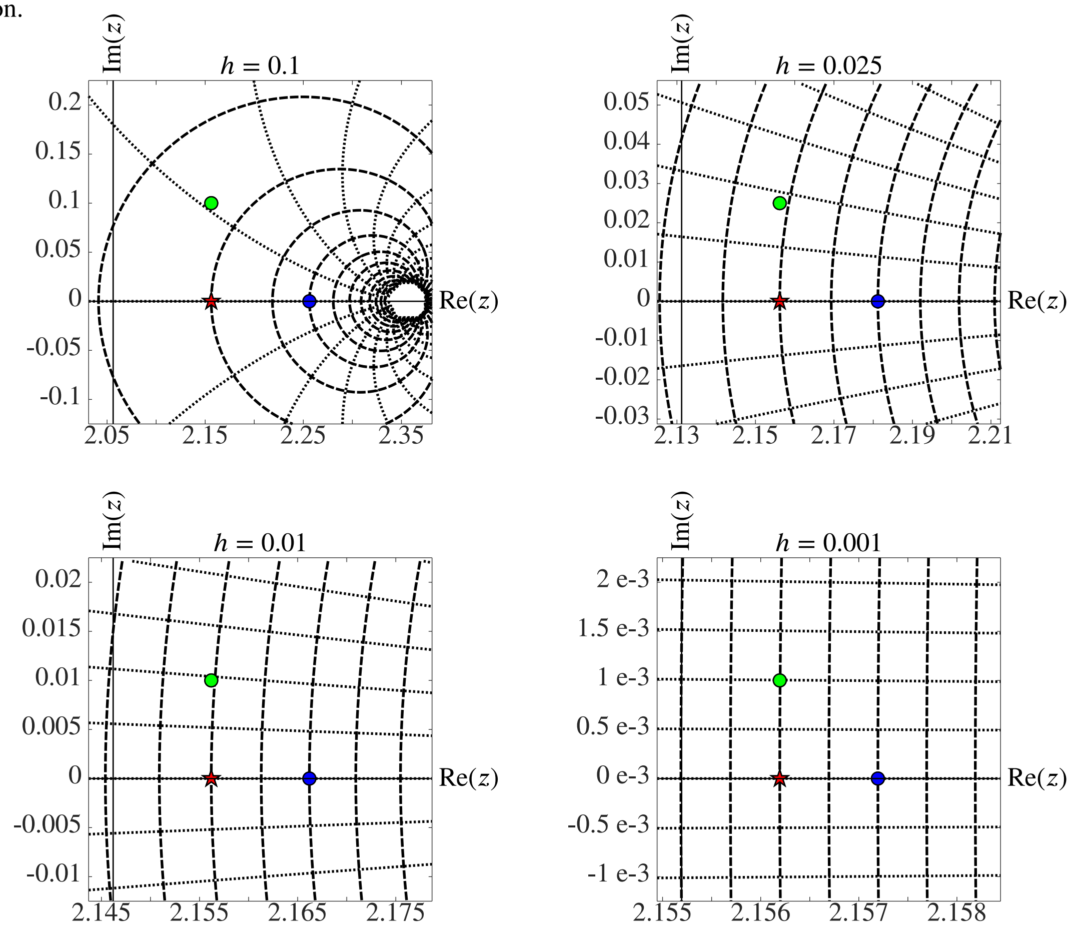


Figure 4.1: Finite differencing in the complex plane: Contour plots of the complex function $f(z) = {}^{e^z}\!/\!\sqrt{\sin^3(z) + \cos^3(z)}$, with dashes representing lines of constant real value and dots representing lines of constant imaginary value; the interval between contours is uniform and identical for both real and imaginary measures. Each subfigure represents a closer zoom on the point $z_0 = {}^{3\pi}\!/_4 - 0.2$ (red star) and a correspondingly smaller step-size h; the blue dot represents a real-valued FD evaluation $f(z_0 + h)$ and the green dot represents a complex-valued CD evaluation $f(z_0 + ih)$. Per the Cauchy-Riemann relationships (Equation 4.12), the contours are perpendicular; for small h, they are almost equally-spaced, representing identical slopes.

The figure also demonstrates the differences between the FD and CD sampling schemes. To see how the complex-step evaluation $f(x+ih)$ can be treated similarly to the finite-difference evaluation $f(x+h)$, examine the original real-valued derivative ${}^{\partial u}/_{\partial x}$ from Equation 4.13 and substitute in the definition of the derivative in the imaginary direction for the RHS:

$$\begin{aligned}\frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} &(4.15)\\ &= \frac{\partial}{\partial y}\Big(\mathrm{Im}\big(f\,(x+iy)\big)\Big) &(4.16)\\ &= \lim_{h\to 0}\frac{\mathrm{Im}\big(f\,(x+i(y+h))\big)-\mathrm{Im}\big(f\,(x+iy)\big)}{h} &(4.17)\end{aligned}$$

As f was originally a function of real variables only, $y = 0$:

$$\frac{\partial u}{\partial x} = \lim_{h\to 0}\frac{\mathrm{Im}\big(f\,(x+ih)\big)-\mathrm{Im}\big(f\,(x)\big)}{h} \tag{4.18}$$

Equation 4.18 has a structure with an obvious parallel to the real-valued definition of the derivative. Indeed, the last subplot of Figure 4.1 shows that a very small step in the imaginary direction produces the same change in the imaginary output as a similar step in the real direction would in the real output. The standard forward finite-difference method uses the analogous real-valued equation to build an approximation with a small h; the complex-step method, however, can make one more simplification beyond Equation 4.18 by noting that the original f was purely real-valued, and thus $\mathrm{Im}\big(f(x)\big) = 0$:

$$\frac{\partial u}{\partial x} = \lim_{h\to 0}\frac{\mathrm{Im}\big(f\,(x+ih)\big)}{h} \tag{4.19}$$

which directly yields the approximation from Equation 4.2.

4.1.2 Practical Usage

With CD, one is now free to choose an arbitrarily small step size, and can achieve results five to ten orders of magnitude more accurate than even centered FD; this behavior is shown in Figure 4.2. There are minor implementation issues that must be addressed when transforming a code to use the method – namely, there are a few functions that are

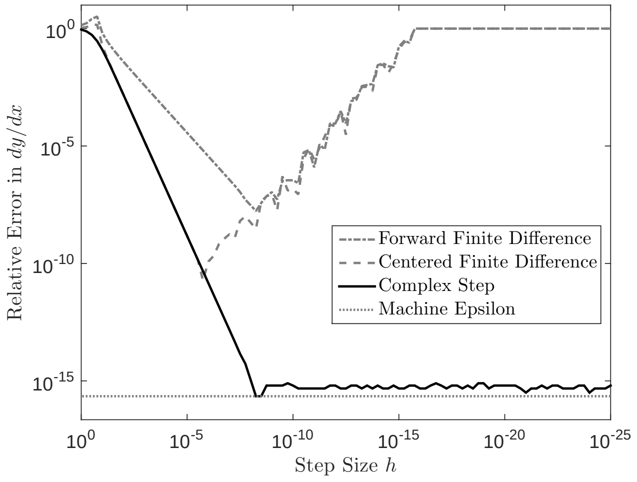


Figure 4.2: Complex-step error in the first derivative: Just as in Figure 3.1 (adopting the same conventions as [62] and [63]), the test function is $y(x) = {}^{e^x}\!/\!\sqrt{\sin^3 x + \cos^3 x}$ evaluated at $x = {}^{3\pi}\!/\!_4 - 0.2$ and h is presented logarithmically decreasing from left to right, such that expected "better" values are on the right. The complex-step method exhibits no cancellation error; it is limited only by machine epsilon – which, for the double-precision arithmetic used here, comes to $2^{-52} \approx 2.22 \times 10^{-16}$.

not complex-analytic (e.g., absolute value), and a few that are undefined off the real line (e.g., comparison operators like less-than). These issues were overcome by Martins, Sturdza and Alonso [63], who in 2003 introduced[2] the method to the aerospace field and presented implementations for multiple languages including C++ and FORTRAN, along with scripts to automate the change of types; all problematic functions were simply redefined to operate only on the real part of the number. In the case of C++, they provided a custom "wrapper" overloaded data type, which added the needed functionality to the default complex type; for FORTRAN, they provided a module that overloaded all intrinsic complex functions. They then applied CD to several large aerospace problems written in FORTRAN, including an analysis of supersonic natural laminar flow and an aero-structural solver for multidisciplinary optimization of a wing.

It is important to note that use of the C++ wrapper forces users to implement a custom type throughout their (scalar) code – in which case, the same limitations of ordinary overloaded AD apply; as ordinary AD is not an approximation and can be more efficient, it may remain more attractive. In the FORTRAN case, in fact, the provided module actually changed all intrinsic functions (even those that were already analytic) to use literal derivative formulas rather than complex arithmetic, effectively transforming complex numbers into standard AD variables. Only operators

[2] More precisely, Martins developed the modified implementations as a corollary to his 2002 PhD dissertation [93], advised by Alonso, where he used CD to verify the coupled aero-structural adjoint method that was the primary thrust of his work. However, their joint publication with Sturdza [63] (which did not discuss his adjoint method) is generally treated as the seminal CD work, alongside Squire and Trapp [92].

(addition, multiplication, etc), which cannot be redefined in FORTRAN, were left unchanged. This still provided an advantage for that language, in which a change from intrinsic to custom types is more complicated than in C++.

Martins et al [63] did not discuss the application of CD to matrix functions in their work, beyond noting that any conjugate transpose operation in MATLAB must be replaced by a real-valued transpose; as they did not address the issue for any other language, this mention may have merely been for the sake of reshaping sets of variables stored as vectors. The method does, however, work for many matrix routines.

If $f(\mathbf{x})$ truly represents an arbitrary program with complex-analytic behavior, one might expect any internal use of matrix math to be irrelevant as far as CD is concerned – intuitively, the method can be applied to any such matrix operation as long as each scalar entry of the matrix result is a complex-analytic function of the scalar entries of the matrix argument(s). This intuition was more rigorously investigated by Al-Mohy and Higham [94], who analyzed the complex-step approximation for derivatives of matrix-input functions, i.e.,

$$f'(\mathbf{A}) = \left.\frac{\operatorname{Im}\left(f\left(\mathbf{A} + ih\dot{\mathbf{A}}\right)\right)}{h}\right|_{h\ll 1} \tag{4.20}$$

Their analysis pointed to a few flaws: first, many individual matrix operations are *not* complex-analytic – for example, any algorithm that utilizes a complex conjugate. Second, CD only works because it avoids subtractive cancellation errors – but certain functions that internally already use complex math (e.g., trigonometric functions that utilize a difference of complex exponentials), when fed a very small input, might introduce such errors.

However, because matrix addition, multiplication, and inversion are analytic and do not suffer from these flaws, any matrix function represented by a power series is amenable to CD, as is any rational (polynomial times an inverse polynomial) representation; this includes the very common approach of repeated multiplication by the same matrix. This still gives CD a broad applicability to matrix math.

4.2 Complex Variants

While complex-step differentiation is practical and elegant, it is limited by the fact that it cannot find higher-order derivatives. This shortcoming was addressed in at least three competing follow-on methods, each describing a different algebra and each published between 2011 and 2013. Two of them, utilizing “multicomplex” and “quasi-complex” numbers, introduced additional imaginary elements and are covered below; the third, using “hyper-dual” numbers, switched to nilpotent elements and is covered in Section 4.4. All three, however, were preceded by a more efficient method utilizing truncated Taylor polynomials, described in Section 4.5; despite this, it is important to discuss each, as they will give analytic insight into the more effective scalar and matrix algebras derived in Part II.

4.2.1 Multicomplex Variables

In 2012, Lantoine, Russell, and Dargent [39] introduced an arbitrary-order differentiation method based on the existing algebra of multicomplex numbers [95, 96], denoted in set notation as $\mathbb{C}_{(q)}$. A number $z \in \mathbb{C}_{(q)}$ is inductively defined as:

$$z = z_0 + z_q i_q \tag{4.21}$$

$$z_0, z_q \in \mathbb{C}_{(q-1)} \tag{4.22}$$

$$i_q^2 = -1 \tag{4.23}$$

$$\forall j, k : i_j i_k = i_k i_j \tag{4.24}$$

$$\mathbb{C}_{(1)} \equiv \mathbb{C} \tag{4.25}$$

$$\mathbb{C}_{(0)} \equiv \mathbb{R} \tag{4.26}$$

Given an element $z \in \mathbb{C}_{(q)}$ (termed *a multicomplex number of degree q*), the scalar values z_0 and z_q are not real numbers, but elements of $\mathbb{C}_{(q-1)}$. Unrolling one level of induction, z can thus be expressed as:

$$z = (z_{0,0} + z_{0,q-1} i_{q-1}) + (z_{0,q} + z_{q,q-1} i_{q-1}) i_q \tag{4.27}$$

Here, the subscripts are used to indicate which imaginary elements will be multiplied with a given coefficient – for instance, $z_{q,q-1}$ is multiplied by both i_{q-1} and i_q but $z_{0,0}$ has not been multiplied with anything.

Lantoine et al demonstrated that, just as in CD, very small steps in imaginary directions could be used to approximate derivatives to high accuracy. Omitting their derivation for brevity and using a second order variable as an example:

$$z \in \mathbb{C}_{(2)} \tag{4.28}$$

$$\therefore \qquad z = z_0 + z_{0,1} i_1 + z_{0,2} i_2 + z_{2,1} i_1 i_2 \tag{4.29}$$

Their multicomplex differentiation method (MCX) proceeded by setting the real coefficient of singleton imaginary dimensions to the small step-size h. Using the functions Im_1, Im_2, and Im_{12} to access $z_{0,1}$, $z_{0,2}$, and $z_{2,1}$, respectively, the formulas for all first- and second-order derivatives of a two-input function are:

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{\mathrm{Im}_1\left(f(x + hi_1 + hi_2, y))\right)}{h} \approx \frac{\mathrm{Im}_2\left(f(x + hi_1 + hi_2, y))\right)}{h} \tag{4.30}$$

$$\frac{\partial f(x,y)}{\partial y} \approx \frac{\mathrm{Im}_1\left(f(x, y + hi_1 + hi_2))\right)}{h} \approx \frac{\mathrm{Im}_2\left(f(x, y + hi_1 + hi_2))\right)}{h} \tag{4.31}$$

$$\frac{\partial^2 f(x,y)}{\partial x^2} \approx \frac{\mathrm{Im}_{12}\left(f(x + hi_1 + hi_2, y))\right)}{h^2} \tag{4.32}$$

$$\frac{\partial^2 f(x,y)}{\partial y^2} \approx \frac{\mathrm{Im}_{12}\left(f(x, y + hi_1 + hi_2))\right)}{h^2} \tag{4.33}$$

$$\frac{\partial^2 f(x,y)}{\partial x \partial y} \approx \frac{\mathrm{Im}_{12}\left(f(x + hi_1, y + hi_2))\right)}{h^2} \tag{4.34}$$

Note that, in terms of which coefficients track which derivatives, this method corresponds exactly with the most naive type of AD induction described in Section 3.3.2 – the MCX variable from Equation 4.29 is essentially used as a two-input variant of the second-order univariate derivative in Equation 3.59. During any computation of high-order derivative tensors (including the Hessian), this approach will redundantly compute many lower-order values; for example, when finding ${}^{\partial^2 f}/_{\partial x^2}$, ${}^{\partial f}/_{\partial x}$ will be contained in both $\mathrm{Im}_1(f)$ and $\mathrm{Im}_2(f)$.

In general, for all q^{th}-order derivatives of n inputs, MCX needs one evaluation for each unique entry in the highest-order tensor, and each evaluation needs to utilize variables with 2^q real coefficients. The cost relative to the original program (not counting the difference between complex arithmetic and simpler derivative formulas) is thus:

$$\binom{n+q-1}{q}2^q = \frac{(n+q-1)!}{q!(n-1)!}2^q \tag{4.35}$$

Lantonine et al emphasized ease of implementation as a primary reason to consider MCX, as the inductive definition can easily be echoed with recursive data structures and repeated application of ordinary complex operator logic. However, just as with ordinary CD, use of the method requires an overloaded type – and unlike CD, there are no situations (such as FORTRAN operators or complex matrix[3] libraries) where MCX is more practical to use than any other overloaded AD type. Further, even when it comes to use of imaginary elements, another high-order tool can be more efficient both in terms of speed and memory.

4.2.2 Quasi-Complex Gradients

Also in 2012, this author (working with Ryan Starkey) independently developed an approach called the method of quasi-complex gradients (QCG) [97]. In contrast to the already-existing algebra of multicomplex numbers, the method defined an entirely new algebra of what were termed quasi-complex numbers. Denote the set of n-variable, q^{th}-order quasi-complex numbers by $\mathbb{C}_{(n,q)}$. These numbers were constructed by beginning with a first-order definition for numbers from $\mathbb{C}_{(n,1)}$:

$$a = a_0 + \sum_{j=1}^{n} a_j i_{(j,1)} \tag{4.36}$$

$$\forall j : a_j \in \mathbb{R} \tag{4.37}$$

$$\forall j : i_{(j,1)}^2 = -1 \tag{4.38}$$

Such first-order variables were shown to be capable of computing many CD-type first derivatives at once – hence the name quasi-complex "gradient". Higher-order numbers $a \in \mathbb{C}_{(n,q)}$ were inductively defined by replacing each real value within lower-order numbers with other quasi-complex numbers, in turn built around a generalization

[3] Lantonine et al did describe a possible matrix representation of multicomplex numbers, but appear not to have used it in their reported experiments. This representation will be discussed in Section 6.3.1.

of imaginary numbers $\mathbb{I}_{(q)}$:

$$a = a_0 + \sum_{j=1}^{n} a_j i_{(j,q)} \tag{4.39}$$

$$\forall j \, : \, a_j \in \mathbb{C}_{(n,q-1)} \tag{4.40}$$

$$\forall j, p, k, r \, : \, i_{(j,p)} i_{(k,r)} = i_{(k,r)} i_{(j,p)} \begin{cases} \in \mathbb{I}_{(p+r)} & (p+r) \leq q \\ = -1 & (p+r) > q \end{cases} \tag{4.41}$$

In other words, where multicomplex numbers are an inductive nesting of standard complex numbers, quasi-complex numbers were designed to be inductive *with vectorized imaginary components*. This allowed for the calculation of entire derivative tensors at once (rather than one entry at a time, as with MCX), which avoided much of the redundant computation of lower-order terms. Further, because the test-cases chosen for publication proved (inadvertently) to have significant overhead, vectorization also provided very significant cost-savings, just as seen in Figure 3.2.

More importantly, the above algebraic definition was only used for derivation, not implementation – many of the pitfalls of naive induction mentioned in Section 3.3.2 were recognized, and significant effort was made to avoid them by writing a special program to automatically generate code for overloaded QCG data-types of any desired order. This generating program only needed to be run once for a given order of derivative, as the resulting variable declaration was portable to any user program. The program was designed to be aware of variable aliases (i.e., redundant copies) – the output code only tracked unique data, stored in a single contiguously-indexed vector for the sake of cache operation. The generating algorithm symbolically built analytic QCG representations of the needed operators (multiplication, division, etc.) and functions (exp, etc.), then recursively replaced the necessary symbols with additional nested QCG representations, until the desired depth of recursion (i.e., order of derivative) had been reached; it then converted the symbolic representation to C++ code, using the appropriate object member variables and indices in place of the analytic symbols. Where possible, it used the process of symbolic expansion to recognize variables worth pre-computing before entering loops. For example, the standard complex multiplication and division operators both utilize the real component of the number to compute an imaginary result; when applied to the expansion of a q^{th}-order derivative term, this "quasi-real" value translated into the value of a $(q-1)^{\text{th}}$-order combination of components, which was computed before entering the q^{th} loop.

Simply by recognizing and avoiding redundant data, the cost of QCG relative to the original program was proportional only to the total number of derivatives to be calculated across all tensors, i.e.,

$$\sum_{j=1}^{q} \binom{n+j-1}{j} = \sum_{j=1}^{q} \frac{(n+j-1)!}{j!(n-1)!} \tag{4.42}$$

which translated to a savings of nearly 2^q over MCX.

Unfortunately, the generating program was still limited by the algebraic definition of quasi-complex numbers, which made it unaware of simplifying derivative formulas like Leibniz's rule; in a way, it was thus similar to the tree structure [76] mentioned in Section 3.3.2, albeit with a much more efficient memory scheme – but unlike that approach, QCG used imaginary arithmetic, which comes with a cost discussed in the next section. Further considering that, just as with MCX, changing a code to use QCG was no easier than with an overloaded AD type (assuming proper use and understanding of the AD tool in question, which can admittedly be nontrivial), other approaches can still be more attractive.

4.2.3 Shortcomings of Imaginary Elements

The use of small steps in imaginary directions – whether one at a time, as with CD, or many at a time, as with MCX and QCG – can compute derivative approximations with the same accuracy shown in Figure 4.2. However, the use of complex arithmetic is more expensive than directly using derivative formulas. One can better understand the difference by directly comparing individual complex operations to the corresponding formulas used by standard automatic differentiation. Consider three simple examples: multiplication, division, and the cosine function. If used within a larger code that computes some $f(x)$, the goal then is to compute $g_\star(a, b \mid x) = ab$, $g_\div(c, d \mid x) = {}^{c}/_{d}$, and $g_{\cos}(\theta \mid x) = \cos(\theta)$ along with the corresponding derivatives $\dot{g}_\star = {}^{\mathrm{d}g_\star}/_{\mathrm{d}x}$, $\dot{g}_\div = {}^{\mathrm{d}g_\div}/_{\mathrm{d}x}$, and $\dot{g}_{\cos} = {}^{\mathrm{d}g_{\cos}}/_{\mathrm{d}x}$, found from the intermediate variables a, b, c, d, and θ already computed by the program. From Equation 4.2, these variables will carry approximate derivatives of themselves with respect to x in the imaginary component and can be rewritten as $\hat{a} \approx a + ih\dot{a}$, etc.

In the case of multiplication,

$$\hat{g}_{\star}(\hat{a},\hat{b}) = (a + ih\dot{a})(b + ih\dot{b}) \tag{4.43}$$

$$= (ab - h^2\dot{a}\dot{b}) + i(h\dot{a}b + ha\dot{b}) \tag{4.44}$$

Now introduce the approximation implicitly assumed by Equation 4.2:

$$h \gg h^2 \approx 0 \tag{4.45}$$

Under this approximation, the $h^2\dot{a}\dot{b}$ term is entirely extraneous – more than a quarter of the cost of complex multiplication is wasted. The high accuracy of the method holds because Equation 4.43 can be simplified to:

$$\mathrm{Re}\left(\hat{g}_{\star}\right) \approx ab \tag{4.46}$$

$$\mathrm{Im}\left(\hat{g}_{\star}\right) = h(\dot{a}b + a\dot{b}) \tag{4.47}$$

The imaginary component contains the product rule – in other words, the same derivative structure of $\hat{a}$ and $\hat{b}$ has been passed down. Next, in the case of division,

$$\hat{g}_{\div}(\hat{c},\hat{d}) = \frac{c + ih\dot{c}}{d + ih\dot{d}} \tag{4.48}$$

$$= \frac{cd + h^2\dot{c}\dot{d}}{d^2 + h^2\dot{d}^2} + i\frac{h\dot{c}d - hc\dot{d}}{d^2 + h^2\dot{d}^2} \tag{4.49}$$

Again using Equation 4.45, this can be simplified to:

$$\mathrm{Re}\left(\hat{g}_{\div}\right) \approx \frac{cd}{d^2} = \frac{c}{d} \tag{4.50}$$

$$\mathrm{Im}\left(\hat{g}_{\div}\right) \approx h\frac{\dot{c}d - c\dot{d}}{d^2} \tag{4.51}$$

which is of course the true result and the derivative quotient rule, respectively. However, the complex division required 13 multiplications, three additions or subtractions, and two divisions. Compare this to five multiplications, one subtraction, and two divisions for the quotient rule – in this case, CD is several times more expensive than traditional AD.

Finally, in the case of the cosine function, the trigonometric identity

$$\cos(p \pm q) \equiv \cos(p)\cos(q) \mp \sin(p)\sin(q) \tag{4.52}$$

can be used to expand the definition of $g_{\cos}$:

$$\hat{g}_{\cos}(\hat{\theta}) = \cos(\theta + ih\dot{\theta}) \tag{4.53}$$

$$= \cos(\theta)\cos(ih\dot{\theta}) - \sin(\theta)\sin(ih\dot{\theta}) \tag{4.54}$$

This can be recast using the hyperbolic cosine and sine functions

$$\sinh(p) = -i\sin(ip) \tag{4.55}$$

$$\cosh(q) = \cos(iq) \tag{4.56}$$

and simplified with the small-angle[4] approximations

$$\cosh(p)\,|_{p\approx 0} \approx 1 \tag{4.57}$$

$$\sinh(q)\,|_{q\approx 0} \approx q \tag{4.58}$$

to yield:

[4] Like the more common small-angle approximations for sin and cos, these are simply the first term of the Maclaurin/Taylor series expansions.

$$\hat{g}_{\cos}(\hat{\theta}) = \cos(\theta)\cosh(h\dot{\theta}) - \sin(\theta)\frac{\sinh(h\dot{\theta})}{-i} \tag{4.59}$$

$$\approx \cos(\theta) - ih\dot{\theta}\sin(\theta) \tag{4.60}$$

This results in:

$$\mathrm{Re}\left(\hat{g}_{\cos}\right) \approx \cos(\theta) \tag{4.61}$$

$$\mathrm{Im}\left(\hat{g}_{\cos}\right) \approx -h\dot{\theta}\sin(\theta) \tag{4.62}$$

which is again the true result as well as the chain rule applied to the derivative of the cosine function. However, along the way, the complex evaluation likely required at least twice as many trigonometric calculations (in the form of cosh and sinh) and two unnecessary multiplications.

While possibly pedantic, the preceding expansions and simplifications show that, in a certain sense, complex formulas *are* differentiation formulas – albeit with additional, costly calculations that are extraneous for $h \ll 1$. Fortunately, these extra calculations can be avoided by a switch from imaginary elements to another non-real type.

4.3 Dual Numbers

So-called *dual numbers* [98] are an extension of the real numbers, just as complex numbers are likewise an extension. (Note, these are unrelated to the concept of a dual *space*.) Whereas a complex number $z \in \mathbb{C}$ is defined as:

$$z = x + iy \tag{4.63}$$

$$x, y \in \mathbb{R} \tag{4.64}$$

$$i^2 = -1 \tag{4.65}$$

a dual number $a \in \mathbb{D}$[5] is defined as:

$$a = b + \varepsilon c \tag{4.66}$$

$$b, c \in \mathbb{R} \tag{4.67}$$

$$\varepsilon^2 = 0 \tag{4.68}$$

$$\varepsilon \neq 0 \tag{4.69}$$

In other words, where i is imaginary, ε is a *nilpotent* element. A crucial property of dual numbers is the fact that, like CD, the evaluation of a dual number input to a function can compute the derivative of that function [99] – this can be quickly demonstrated with a Taylor series expansion of $f(x + h\varepsilon)$ about x:

$$\begin{aligned} f(x + h\varepsilon) &= f(x) + \frac{h\varepsilon}{1!}\frac{\mathrm{d}f}{\mathrm{d}x} + \frac{h^2\varepsilon^2}{2!}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} + \frac{h^3\varepsilon\varepsilon^2}{3!}\frac{\mathrm{d}^3 f}{\mathrm{d}x^3} + \dots \\ &= f(x) + h\frac{\mathrm{d}f}{\mathrm{d}x}\varepsilon \end{aligned} \tag{4.70}$$

Note that, unlike the complex-step approximation, there are no h^2 or higher expressions – Equation 4.70 is a finite sequence with precisely two terms. Therefore, no matter the size of h, the nilpotent term holds the *exact* derivative. Indeed, just as with standard automatic differentiation, if h is set to 1 in the input, follow-on nilpotent terms are identically equal to the derivative with respect to this input, with no coefficient to divide out – setting a nilpotent term to unity is equivalent to the tautology "the derivative of this number with respect to itself is 1".

In this work, the nilpotent coefficient of any dual number shall be accessed via the operator Nil(·), just as the real and imaginary coefficients are accessed via the Re(·) and Im(·) operators, respectively. Using Equation 4.70 as an example:

[5] Note that dual numbers have not been used often enough in other contexts to have earned their own set notation. The use of $\mathbb{D}$ is adopted due to their frequency in this work.

$$\mathrm{Re}(f(x+h\varepsilon)) = f(x) \tag{4.71}$$

$$\mathrm{Nil}(f(x+h\varepsilon)) = h\frac{\mathrm{d}f}{\mathrm{d}x} \tag{4.72}$$

Another Taylor series expansion proves the chain rule holds:

$$g\big(f(x+h\varepsilon)\big) = g\left(f(x) + h\frac{\mathrm{d}f}{\mathrm{d}x}\varepsilon\right) \tag{4.73}$$

$$= g\big(f(x)\big) + \frac{h\varepsilon}{1!}\frac{\mathrm{d}f}{\mathrm{d}x}\frac{\mathrm{d}g}{\mathrm{d}f} + \frac{h^2\varepsilon^2}{2!}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\frac{\mathrm{d}g}{\mathrm{d}f} + \dots \tag{4.74}$$

$$= g\big(f(x)\big) + h\frac{\mathrm{d}f}{\mathrm{d}x}\frac{\mathrm{d}g}{\mathrm{d}f}\varepsilon \tag{4.75}$$

$$\mathrm{Re}\Big(g\big(f(x+h\varepsilon)\big)\Big) = g\big(f(x)\big) \tag{4.76}$$

$$\mathrm{Nil}\Big(g\big(f(x+h\varepsilon)\big)\Big) = h\frac{\mathrm{d}g}{\mathrm{d}x} \tag{4.77}$$

Note that both Taylor series from Equation 4.70 and Equation 4.73 are equally valid for complex (analytic) functions with real inputs, which means that – unlike CD – any code utilizing complex math would not break if differentiated by dual numbers. In other words, for $f(x) = u(x) + v(x)i$,

$$f(x+h\varepsilon) = f(x) + h\frac{\mathrm{d}f}{\mathrm{d}x}\varepsilon \tag{4.78}$$

$$= u(x) + v(x)i + h\left(\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{\mathrm{d}v}{\mathrm{d}x}i\right)\varepsilon \tag{4.79}$$

$$u(x+h\varepsilon) = u(x) + h\frac{\mathrm{d}u}{\mathrm{d}x}\varepsilon \tag{4.80}$$

$$v(x+h\varepsilon) = v(x) + h\frac{\mathrm{d}v}{\mathrm{d}x}\varepsilon \tag{4.81}$$

When it comes to implementation of scalar dual numbers (e.g., [100, 101]), the code for low-level nilpotent operations collapses to simple derivative formulas – and so, although derived via a special algebra, a dual number overloaded data-type is completely indistinguishable from a single-variable, first-order AD type.[6] It is still useful to frame differentiation in terms of dual numbers for the sake of algebraic manipulation and analysis – most importantly, such an analysis will lead to a more general differentiation approach in Section 5.3, along with a way to represent these numbers using only real values arranged in a matrix in Chapter 6. First, for the sake of completeness, it is necessary to visit two other algebras useful for AD.

4.4 Hyper-Dual Numbers

An extension of the duals termed "hyper-dual" numbers [26, 102–104] capable of second-order differentiation was introduced in 2011 by Fike and Alonso; the work ultimately became the subject of Fike's 2013 dissertation. [26][7] For the sake of simplicity and consistency with the algebraic framework described earlier in Section 4.2.1, hyper-duals are described herein with a parallel structure to multicomplex numbers: just as multicomplex numbers $\mathbb{C}_{(q)}$ can be thought of "complex numbers with complex components", hyper-duals are "dual numbers with dual components" – although unlike multicomplex numbers, hyper-duals are only nested once, rather than an arbitrary number of times. (Note, Fike and Alonso appear to have been unaware of Lantonine et al's [39] work[8] , and vice versa; Martins did, however, go on to cite both in a 2013 review [105].) For this reason, hyper-duals will be denoted herein as $\mathbb{D}_{(2)}$, the structure of which can be derived as:

$$a \in \mathbb{D}_{(2)} \tag{4.82}$$

$$a = a_0 + a_2\varepsilon_2 \tag{4.83}$$

$$a_0, a_2 \in \mathbb{D} \tag{4.84}$$

$$\varepsilon_2^2 = 0 \neq \varepsilon_2 \tag{4.85}$$

$$a_0 = a_{0,0} + a_{0,1}\varepsilon_1 \tag{4.86}$$

[6] Strictly speaking, there are operations on dual numbers that are undefined for reals, e.g., the conjugate $\overline{b + \varepsilon c} \equiv b - \varepsilon c$. However, these operations do not correspond to any differentiable functions and in practice are omitted from AD implementations.

[7] Note that Fike, like Martins (whose PhD thesis [93] broadened the use of the complex-step method), was also graduate student of Alonso; unlike Martins, hyper-dual numbers were the primary focus of Fike's dissertation, not a corollary.

[8] Just as this author was, at the time, unaware of their work.

$$a_2 = a_{2,0} + a_{2,1}\varepsilon_1 \tag{4.87}$$

$$a_{0,0}, a_{0,1}, a_{2,0}, a_{2,1} \in \mathbb{R} \tag{4.88}$$

$$\varepsilon_1^2 = 0 \neq \varepsilon_1 \tag{4.89}$$

$$\varepsilon_1\varepsilon_2 = \varepsilon_2\varepsilon_1 \neq 0 \tag{4.90}$$

As generalized induction is not involved, simpler subscripts can be used to identify the elements. The definition can thus be simplified to:

$$b \in \mathbb{D}_{(2)} \tag{4.91}$$

$$b = b_0 + b_1\varepsilon_1 + b_2\varepsilon_2 + b_{12}\varepsilon_1\varepsilon_2 \tag{4.92}$$

$$b_0, b_1, b_2, b_{12} \in \mathbb{R} \tag{4.93}$$

$$\varepsilon_1^2 = \varepsilon_2^2 = 0 \tag{4.94}$$

$$\varepsilon_1\varepsilon_2 = \varepsilon_2\varepsilon_1 \neq 0 \tag{4.95}$$

Just like MCX and QCG, when run through a formerly real-valued code hyper-dual numbers are capable of finding a mixed second derivative. Again using a Taylor series and the operators $\text{Nil}_1, \text{Nil}_2$ and Nil_{12} to access the coefficients of $\varepsilon_1, \varepsilon_2$ and ε_{12}, respectively:

$$f(x + h\varepsilon_1, y + h\varepsilon_2) = f(x) + h\frac{\partial f}{\partial x}\varepsilon_1 + h\frac{\partial f}{\partial y}\varepsilon_2 \tag{4.96}$$

$$+ \frac{\varepsilon_1^2 h^2}{2}\frac{\partial^2 f}{\partial x^2} + \frac{\varepsilon_1\varepsilon_2 h^2}{2}\frac{\partial^2 f}{\partial x\partial y} + \frac{\varepsilon_2\varepsilon_1 h^2}{2}\frac{\partial^2 f}{\partial x\partial y} + \frac{\varepsilon_2^2 h^2}{2}\frac{\partial^2 f}{\partial y^2} \tag{4.97}$$

$$= f(x) + h\frac{\partial f}{\partial x}\varepsilon_1 + h\frac{\partial f}{\partial y}\varepsilon_2 + h^2\frac{\partial^2 f}{\partial x\partial y}\varepsilon_1\varepsilon_2 \tag{4.98}$$

$$f(x) = \text{Re}\left[f(x + h\varepsilon_1, y + h\varepsilon_2)\right] \tag{4.99}$$

$$\frac{\partial f}{\partial x} = \text{Nil}_1\left[f(x + h\varepsilon_1, y + h\varepsilon_2)\right]/h \tag{4.100}$$

$$\frac{\partial f}{\partial y} = \text{Nil}_2\left[f(x + h\varepsilon_1, y + h\varepsilon_2)\right]/h \tag{4.101}$$

$$\frac{\partial^2 f}{\partial x\partial y} = \text{Nil}_{12}\left[f(x + h\varepsilon_1, y + h\varepsilon_2)\right]/h^2 \tag{4.102}$$

Note that Fike used a more involved derivation [102]: he started by looking for an extension of the complex numbers that could track two first derivatives and a second derivative, which at first led him to the set of quaternions; when these proved to suffer from the same subtractive cancellation error as finite differences, he began investigating different types of non-real elements, focusing especially on the commutivity property (Equation 4.90); he then ultimately chose nilpotent elements of degree two, recognizing that they led to zero truncation error in the derivative (i.e., higher-order terms in the Taylor series were zero). It appears that only then did the connection to canonical dual numbers become clear, leading to the name “hyper-dual”.

Fike and Alonso used the new numbers to find second derivatives for multiple aerospace applications, including unsteady Reynolds-averaged Navier Stokes computational fluid dynamics (URANS CFD) and inviscid transonic flow; the resulting information was used for Kriging (a type of surrogate modeling) and multi-objective optimization of a supersonic airfoil and a business jet. They also emphasized the potential use of hyper-duals to verify any new (hypothetical) second-order adjoint method, just as Martins [93] had done when developing his first-order coupled adjoint method.

Just like standard duals, the implementation [101] of hyper-duals collapses to literal differentiation formulas; as such, it should be compared to standard AD approaches. However, the new approach was not tested alongside any other second-order, operator-overloaded tool – the only methods considered appear to be the first-order[9] ADOL-C [75] and TAPENADE [7] packages, which use overloading and source modification, respectively. Just as with MCX and QCG, there is little to recommend hyper-duals over newer high-order, forward-mode overloaded types (e.g., [106–109]).

Despite this, hyper-duals are fascinating for their demonstration of high-order nilpotent arithmetic and its use in differentiation – through their use, Fike hinted at a powerful analytic approach to deriving other AD methods. The algebras derived in Chapter 5 owe a certain amount of inspiration to his method.

[9] ADOL-C is technically capable of higher orders, but must record a “tape” of every operation to hard-disk to do so; this is impractically expensive for the large aerospace applications discussed here. TAPENADE is likewise technically capable of second-order calculations, but only by applying it twice – a solution which suffers from all the drawbacks mentioned in Section 3.3.2. Both tools are quite old – TAPENADE dates from 2001, and ADOL-C from 1992 – and were not designed with modern, large-scale codes in mind.

4.5 Truncated Taylor Polynomials

In contrast to direct calculation of high-order derivatives, the tradition in AD for the past several decades (see, e.g., [110,111], cited in [112]) has been to use the algebra of truncated polynomials. Implicitly, these are Taylor series, but the underlying computations are the same for any polynomial.

To understand the concepts of both truncation and implicit Taylor behavior, consider two generic second-order univariate polynomials $y, z \in \mathbb{P}_{(1,\infty)}$, where $\mathbb{P}_{(1,\infty)}$ denotes the set of all single-variable polynomials of arbitrary order:[10]

$$y(t) = y_0 + y_1 t + y_2 t^2 \tag{4.103}$$

$$z(t) = z_0 + z_1 t + z_2 t^2 \tag{4.104}$$

When multiplied together, these yield:

$$\begin{aligned} f(t) = y(t)z(t) &= (y_0 + y_1 t + y_2 t^2)(z_0 + z_1 t + z_2 t^2) && (4.105) \\ &= y_0 z_0 + y_0 z_1 t + y_0 z_2 t^2 && (4.106) \\ &\quad + y_1 t z_0 + y_1 t z_1 t + y_1 t z_2 t^2 \\ &\quad + y_2 t^2 z_0 + y_2 t^2 z_1 t + y_2 t^2 z_2 t^2 \\ &= y_0 z_0 + (y_0 z_1 + y_1 z_0) t + (y_0 z_2 + y_1 z_1 + y_2 z_0) t^2 && (4.107) \\ &\quad + (y_1 z_2 + y_2 z_1) t^3 + y_2 z_2 t^4 \end{aligned}$$

When calculating *truncated* polynomials of degree 2, one simply neglects to compute the t^3 and t^4 terms; this truncation to finite length means that second-order truncated polynomials can be stored as overloaded variables in a program, each being internally represented by no more than three real values. For two such variables $\hat{y}, \hat{z} \in \mathbb{P}_{(1,2)}$, multiplication is defined as:

[10] Notation adapted from [108].

$$
\begin{aligned}
\hat{f}(t) &= \hat{y}(t)\hat{z}(t) \\
&= y_0 z_0 + (y_0 z_1 + y_1 z_0)t + (y_0 z_2 + y_1 z_1 + y_2 z_0)t^2
\end{aligned}
\tag{4.108}
$$

Polynomial calculations such as this are trivial to implement and can be done very efficiently. Note that – at least as far as their AD usage is concerned – elements such as $\hat{y}$ and $\hat{z}$ are not treated as functions, but as data; in the implementation of such an overloaded data-type, no powers of t ever have to be calculated, as the resulting coefficients will always be the same. Knowing this, re-imagine $\hat{y}$ and $\hat{z}$ as truncated Taylor series for functions of x with offset t:

$$
\hat{y}(x+t) = \hat{y}(x) + \frac{t}{1!}\frac{\partial \hat{y}(x)}{\partial x} + \frac{t^2}{2!}\frac{\partial^2 \hat{y}(x)}{\partial x^2} \tag{4.109}
$$

$$
\hat{z}(x+t) = \hat{z}(x) + \frac{t}{1!}\frac{\partial \hat{z}(x)}{\partial x} + \frac{t^2}{2!}\frac{\partial^2 \hat{z}(x)}{\partial x^2} \tag{4.110}
$$

$$
\begin{aligned}
\hat{f}(x+t) =& \hat{y}(x+t)\hat{z}(x+t) \\
=& \hat{y}(x)\hat{z}(x) \\
&+ \left(\hat{y}(x)\frac{\partial \hat{z}(x)}{\partial x} + \frac{\partial \hat{y}(x)}{\partial x}\hat{z}(x)\right)t \\
&+ \left(\frac{1}{2}\hat{y}(x)\frac{\partial^2 \hat{z}(x)}{\partial x^2} + \frac{\partial \hat{y}(x)}{\partial x}\frac{\partial \hat{z}(x)}{\partial x} + \frac{1}{2}\frac{\partial^2 \hat{y}(x)}{\partial x^2}\hat{z}(x)\right)t^2
\end{aligned}
\tag{4.111}
$$

$$
= \hat{f}(x) + \frac{\partial \hat{f}(x)}{\partial x}t + \frac{1}{2}\frac{\partial^2 \hat{f}(x)}{\partial x^2}t^2 \tag{4.112}
$$

The generic polynomial multiplication has computed the first- and second-order product rules and stored the result in Taylor form. A similar result holds for multivariate polynomials (i.e., those using multiple offsets t_j for each differentiated input x_j of the function). Note that summation of polynomials is trivial, and obviously preserves the Taylor structure; therefore, between addition and multiplication, it is now possible to compute arbitrary power series of elements in $\mathbb{P}_{(n,q)}$. That is, for any polynomial with n inputs and highest order q, the entire polynomial can itself be raised to any power and summed with other powers. This in turn allows computation of many elementary functions; the exponential of a polynomial [108] is a useful demonstration.

First, for the sake of more concise notation, re-designate elements of $\mathbb{P}_{(n,q)}$ as having the form $\hat{a} = a_0 + \dot{a}$, where

a_0 is the constant value and all other terms are gathered into $\dot{a}$; this implies that any function of $\dot{a}$ in isolation is in fact operating on a polynomial with zero constant term. Using this notation, the exponential function is simply:

$$e^{\hat{x}} = e^{x_0+\dot{x}} = e^{x_0}e^{\dot{x}} \tag{4.113}$$

Recall that e^x is defined as:

$$e^x \equiv \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{4.114}$$

Putting this into the second exponential in Equation 4.113 yields:

$$e^{\hat{x}} = e^{x_0}\left(\sum_{i=0}^{\infty} \frac{\dot{x}^i}{i!}\right) \tag{4.115}$$

However, examination of Equation 4.108 or Equation 4.111 will reveal that under multiplication, every first-order term is multiplied by a constant (zero-order) term from one of the multiplicands; likewise, every second-order term requires either two first-order terms or at least one constant term. Therefore, as $\dot{x}$ has no constant term, $\dot{x}^2$ will have no first-order coefficients and $\dot{x}^3$ will have no second-order coefficients. Generalizing to any degree of polynomial, $\dot{x}^{i+1}$ will have no i^{th}-order coefficients – and so, because $\dot{x}$ is truncated to the finite order q, the quantity $\dot{x}^{q+1}$ is necessarily zero. In other words, $\dot{x}$ is nilpotent, just like the unit elements used by dual and hyper-dual numbers.

Putting all this together, for any truncated Taylor polynomial $\hat{x} \in \mathbb{P}_{(n,q)}$, the exact exponential of $\hat{x}$ is computed by multiplying the original scalar exponential by a *finite* power series:

$$e^{\hat{x}} = e^{x_0}\left(\sum_{i=0}^{q} \frac{\dot{x}^i}{i!}\right) \tag{4.116}$$

Similar power-series formulas exist for other operations; the most common can be found in [42, 108]. Thus, if an overloaded program implements the algebra of $\mathbb{P}_{(n,q)}$, then any polynomial inputs structured as Taylor series will retain that structure as they propagate through the program. Derivatives can then be recovered from the output by dividing particular polynomial coefficients by their corresponding Taylor coefficients. By calculating all derivative terms via such simple power series of easy-to-compute polynomials, the Taylor-series approach to AD is significantly more efficient than using literal differentiation formulas, which require more complicated computations (e.g., binomial coefficients for Leibniz's rule).

Importantly, a multivariate Taylor approach similar to this univariate example can be coupled with NMD, as will be described in Part II and demonstrated in Part III. Before closing this chapter, though, it is necessary to note that still more efficient approaches exist for multivariate derivatives. [42, 78, 112] These methods actually limit themselves to *univariate* Taylor series – one for every entry in the highest-order tensor – and take advantage of the fact that such polynomials use much simpler memory addressing and can thus be computed more efficiently. (Note that NMD, being a matrix algorithm, cannot benefit from such a thing.) Each series is used to find derivatives along specially-selected directions in the domain space; the actual multivariate derivatives are then known functions of the final results. The precise details of such methods are beyond the scope of this paper, but the interested reader can find them in the cited literature.

Part II

Hypercomplex Algebras and Nilpotent Matrix Differentiation

Chapter 5

Generalized Differentiation via Nilpotent Elements

As already described in Section 4.3, dual numbers are an extension of the algebra of real numbers. Most precisely, they form a commutative two-dimensional associative algebra over the reals; more generally, using a historical terminology, they are a type of *hypercomplex* number [113, 114] – an umbrella category that also includes complex, multicomplex, quasi-complex and hyper-dual numbers. The standard duals, as defined, can find first-order derivatives with respect to a single input. It is possible, though, to find other derivatives by further extending the concept of dual numbers and their nilpotent elements to other types of hypercomplex numbers with more dimensions. This chapter introduces the concept by treating the new numbers as scalars; Chapter 6 will then present a matrix representation.

5.1 Hypercomplex Numbers

Hypercomplex numbers are defined [113] as elements of $(D + 1)$-dimensional distributive algebras over the real numbers, generically denoted herein[1] as $\mathbb{H}_{(D)}$; these algebras are not necessarily commutative, although non-commutative variants will not be considered in this chapter. Each number is defined by a set of real coefficients $\{x_0, x_1, \ldots, x_D\}$ for the basis elements $\{1, \hat{e}_1, \ldots, \hat{e}_D\}$, where each $\hat{e}_i$ is non-real and D is finite. Further, for such an algebra to be closed, products of elements must yield elements (real or otherwise) already in the set. A few common examples include:

[1] This is something of an abuse of set notation, as there are in fact many distinct types of hypercomplex algebras of the same dimension. Further, in many sources, the symbol $\mathbb{H}$ is instead used to represent the set of quaternions (which are themselves hypercomplex). It is appropriated here out of necessity.

- Standard complex numbers:

$$D = 1$$

$$\hat{e}_1 \equiv i \qquad\qquad i^2 = -1$$

- Standard dual numbers:

$$D = 1$$

$$\hat{e}_1 \equiv \varepsilon \qquad\qquad \varepsilon^2 = 0 \qquad\qquad \varepsilon \neq 0$$

- Hyper-dual numbers:

$$D = 3$$

$$\hat{e}_1 \equiv \varepsilon_1 \neq 0 \qquad\qquad \hat{e}_2 \equiv \varepsilon_2 \neq 0 \qquad\qquad \hat{e}_3 \equiv \varepsilon_{12} \neq 0$$

$$\varepsilon_1^2 = 0 \qquad\qquad \varepsilon_2^2 = 0 \qquad\qquad \varepsilon_{12}^2 = 0$$

$$\varepsilon_1 \varepsilon_2 = \varepsilon_2 \varepsilon_1 = \varepsilon_{12}$$

- Quaternions:

$$D = 3$$

$$\hat{e}_1 \equiv i \qquad\qquad \hat{e}_2 \equiv j \qquad\qquad \hat{e}_3 \equiv k$$

$$ij = k = -ji \qquad\qquad ki = j = -ik \qquad\qquad jk = i = -kj$$

$$i^2 = j^2 = k^2 = ijk = -1$$

Intuitively, one can apply the AD concept of vectorization to dual numbers, creating hypercomplex numbers with many nilpotent elements to find many first derivatives:

$$\hat{x} = x_0 + \sum_{i=1}^{D} x_i \varepsilon_i \tag{5.1}$$

$$\forall i : x_i \in \mathbb{R} \tag{5.2}$$

$$\forall i : \varepsilon_i^2 = 0 \tag{5.3}$$

$$\forall i : \varepsilon_i \neq 0 \tag{5.4}$$

$$\forall i, j : \varepsilon_i \varepsilon_j = 0 \tag{5.5}$$

Note that the last part of the definition (Equation 5.5) sets the nilpotent elements to be orthogonal to each other – without this, the dimensionality of the algebra would grow to 2^D, requiring a more sophisticated ruleset (discussed in the next section). A multivariate Taylor series once again demonstrates the recovery of derivatives:

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = f(\mathbf{x}) + \sum_{m_1=0}^{\infty} \cdots \sum_{m_D=0}^{\infty} \left(\left(\prod_{i=1}^{D} \frac{(h_i \varepsilon_i)^{m_i}}{m_i!} \right) \frac{\partial^{\sum_j m_j}}{\prod_k \partial {x_k}^{m_k}} f(\mathbf{x}) \right) \tag{5.6}$$

Here, $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{S} \in \mathbb{R}^{D \times D}$ is a diagonal matrix representing step-sizes $S_{ii} = h_i$ in each dimension[2] and $\boldsymbol{\varepsilon} = [\varepsilon_1, \ldots, \varepsilon_D]^T$ contains the nilpotent elements. This can be rewritten as:

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = f(\mathbf{x}) + \sum_i h_i \varepsilon_i \frac{\partial f}{\partial x_i} + \sum_{i,j} h_i h_j \frac{\varepsilon_i \varepsilon_j}{2!} \frac{\partial^2 f}{\partial x_i \partial x_j} + \ldots \tag{5.7}$$

$$= f(\mathbf{x}) + \sum_i h_i \varepsilon_i \frac{\partial f}{\partial x_i} \tag{5.8}$$

Simply define the operator $\mathrm{Nil}_i(\cdot)$ to access the coefficient of ε_i, and each first derivative can be accessed as:

$$\mathrm{Nil}_i\big(f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon})\big)/h_i = \frac{\partial f}{\partial x_i} \tag{5.9}$$

[2] Other configurations of $\mathbf{S}$ can be used as a flexible framework for directional derivatives; see, e.g., [78], who phrase the problem without special algebra as $\nabla_{\mathbf{z}} f(\mathbf{x} + \mathbf{S}\mathbf{z})$ such that $\mathbf{z}$ need not even be of the same dimensions as $\mathbf{x}$.

It is possible to use a similar approach to find high-order derivatives – although the work thus far has concerned itself with nilpotent elements that square to zero, other types are possible. Define a *nilpotent element of degree P* to be any element ε_j with the property ${\varepsilon_j}^P = 0$ but ${\varepsilon_j}^{P-1} \neq 0$. When such an element is used in an AD evaluation, the univariate Taylor series becomes:

$$f(x + h\varepsilon_j) = f(x) + \frac{h\varepsilon_j}{1!}\frac{\partial f(x)}{\partial x} + \frac{(h\varepsilon_j)^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + \cdots + \frac{(h\varepsilon_j)^{P-1}}{(P-1)!}\frac{\partial^{P-1} f(x)}{\partial x^{P-1}} \tag{5.10}$$

The evaluation of $f(x + h\varepsilon_j)$ yields a P-term polynomial in ε_j (counting the real-valued term); the ${\varepsilon_j}^P$ term and all following terms disappear. As can be seen, the m^{th} non-real term will yield the m^{th}-order derivative of f times a known coefficient. By temporarily defining yet another operator $\text{Nil}_{j^m}(\cdot)$ to access the ${\varepsilon_j}^m$ term, it can be seen that

$$\frac{\partial^m f(x)}{\partial x^m} = m!\text{Nil}_{j^m}\big(f(x + \varepsilon_j)\big)/h^m \tag{5.11}$$

Polynomials of this form, whether they encode a derivative or not, describe a new class of hypercomplex algebra. Although the entire polynomial can be described in terms of the bases $\{1, \varepsilon_j\}$, these are not the only basis elements of the algebra – the fact that P coefficients are required to represent a result such as Equation 5.10 implies a total of P dimensions: each non-zero power of ε_j is a basis unto itself, albeit non-orthogonal with every other basis. Before continuing to mixed derivatives, it is important to address this complication.

5.2 Tensor Products and Indexing Notation

As seen, non-real elements with various properties can compute derivatives. However, for an algebra to be closed, every possible product of two basis elements must result in some element already in the set or a multiple thereof. Consider the vectorized algebra defined above. If the orthogonality restriction of Equation 5.5 is removed from the definition, every product of distinct non-reals forms its own dimension; the resulting algebra is not D-dimensional, as might be intended, but (2^D)-dimensional (and if the square of individual elements wasn't eliminated by nilpotency, the dimensionality would be infinite – i.e., the algebra would not be closed). This is precisely the case with the multicomplex numbers from Section 4.2.1, which are quite inefficient in the calculation of derivatives – an element of $\mathbb{C}_{(q)}$ can compute a single q^{th}-order derivative, but the calculation of multiple derivatives of the same order requires a great many redundant calculations of lower-order values. It is better to broadly define a set of rules to limit products to a certain order, as with QCG; this allows for redundancy-free calculation and the omission of unwanted higher orders. It also corresponds to differentiation algorithms that are not written in terms of special algebras: rather than consider all possible multivariate derivatives and then throw out those that are unwanted, most code is written the other way around, by choosing which mixed derivatives are wanted and then finding which coefficients are needed to compute (e.g.) a product rule. Looking even further, the best algebras would allow for sparse sets of derivatives, requiring element-by-element rules to selectively omit variables. Finally, note that the high-order algebras considered so far, when run through a real-valued function, do not directly produce derivatives – instead, they calculate the desired value times some coefficient, which must be divided out in post-processing. Given all this, the concept of a *tensor product* of entire algebras might be more instructive, if only because it makes it easier to distinguish between the multiplication of two elements $\hat{e}_i\hat{e}_j$ and the element that actually results from that multiplication.

Given one algebra $\mathbb{H}_{(D_1)}$ with basis elements $\{1, \hat{e}_1, \ldots, \hat{e}_{D_1}\}$ and another algebra $\mathbb{H}_{(D_2)}$ with basis elements $\{1, \hat{d}_1, \ldots, \hat{d}_{D_2}\}$, the tensor product can be denoted as $\mathbb{H}_{(D_1)} \otimes \mathbb{H}_{(D_2)}$ and results in another algebra with $(D_1+1)(D_2+1)$ dimensions, corresponding to the basis elements $\{1, \hat{e}_1, \ldots, \hat{e}_{D_1}\} \times \{1, \hat{d}_1, \ldots, \hat{d}_{D_2}\}$ enumerated as:

$$\left\{ \begin{array}{cccc} 1 \otimes 1, & 1 \otimes \hat{d}_1, & \ldots, & 1 \otimes \hat{d}_{D_2}, \\ \hat{e}_1 \otimes 1, & \hat{e}_1 \otimes \hat{d}_1, & \ldots, & \hat{e}_1 \otimes \hat{d}_{D_2}, \\ \vdots & & & \\ \hat{e}_{D_1} \otimes 1, & \hat{e}_{D_1} \otimes \hat{d}_1, & \ldots, & \hat{e}_{D_1} \otimes \hat{d}_{D_2} \end{array} \right\} \tag{5.12}$$

Numbers in the new algebra are thus defined by $(D_1 + 1)(D_2 + 1)$ real coefficients $\{x_{0,0}, \ldots, x_{D_1,D_2}\}$. Just as in the original algebras, multiplication must be closed – i.e., any product of tensor elements must yield another tensor element already in $\mathbb{H}_{(D_1)} \otimes \mathbb{H}_{(D_2)}$. This requirement is satisfied if $\mathbb{H}_{(D_1)}$ and $\mathbb{H}_{(D_2)}$ are themselves closed and multiplication of the new basis elements and their coefficients is performed piecewise using underlying rules for the original algebras – that is, for two numbers x and y,

$$\forall i,j,k,l : (x_{i,j}\hat{e}_i \otimes \hat{d}_j)(y_{k,l}\hat{e}_k \otimes \hat{d}_l) \tag{5.13}$$

$$= x_{i,j}y_{k,l}(\hat{e}_i\hat{e}_k) \otimes (\hat{d}_j\hat{d}_l) \tag{5.14}$$

Of special interest to this work is the case of an algebra tensored with itself – that is, $\mathbb{H}_{(D)} \otimes \mathbb{H}_{(D)}$, which gives an easy framework for discussing mixed derivatives with respect to the same set of inputs. For the base algebra $\mathbb{H}_{(D)}$ to be closed, every scalar (i.e., ordinary) product $\hat{e}_i\hat{e}_j$ must result in some element already in the set; in contrast, the tensor elements are entirely new – in other words, $\hat{e}_i\hat{e}_j \neq \hat{e}_i \otimes \hat{e}_j$. Note that the tensor product is not intrinsically commutative, i.e., $(\hat{e}_i \otimes \hat{e}_j) \neq (\hat{e}_j \otimes \hat{e}_i)$, even if multiplication of the new elements remains so, i.e., $(\hat{e}_i \otimes \hat{e}_j)(\hat{e}_k \otimes \hat{e}_l) = (\hat{e}_k \otimes \hat{e}_l)(\hat{e}_i \otimes \hat{e}_j)$. This means that even if $\hat{e}_i$ and $\hat{e}_j$ are orthogonal in the base algebra, $(\hat{e}_i \otimes 1)(1 \otimes \hat{e}_j) = (\hat{e}_i \otimes \hat{e}_j)$. A piecewise definition of multiplication can thus be used to turn the vectorized first-order algebra from Equation 5.1 into an algebra capable of both first and second derivatives.

Note that the tensor of the two real-valued basis elements $(1 \otimes 1)$ is (just like 1 itself) a multiplicative identity, and thus can be treated simply as $(1 \otimes 1) = 1$. This property implies that tensor products of hypercomplex algebras are themselves hypercomplex. The new algebra can therefore be more easily denoted with the shorthand:

$$\mathbb{H}_{(D_1 D_2)} \equiv \mathbb{H}_{(D_1)} \otimes \mathbb{H}_{(D_2)} \tag{5.15}$$

$$x \in \mathbb{H}_{(D_1 D_2)} \implies x = \sum_{i=0}^{D_1} \sum_{j=0}^{D_2} x_{ij}\hat{e}_{ij} \tag{5.16}$$

$$\hat{e}_{ij} \equiv \hat{e}_i \otimes \hat{d}_j \tag{5.17}$$

$$\hat{e}_i \equiv \hat{e}_{i1} = \hat{e}_i \otimes 1 \tag{5.18}$$

$$\hat{d}_j \equiv \hat{e}_{1j} = 1 \otimes \hat{d}_j \tag{5.19}$$

$$\hat{e}_{00} \equiv 1 \tag{5.20}$$

Note that Equation 5.18 and Equation 5.19 are adopted because the original definitions of $\hat{e}_i$ and $\hat{d}_j$ are no longer useful within the new set – operations are only defined within an algebra, so that for $x \in \mathbb{H}_{(D_1 D_2)}$, $y \in \mathbb{H}_{(D_1)}$, and $z \in \mathbb{H}_{(D_2)}$, the products xy, xz, and yz have no meaning.

As an example of how this framework can be useful, consider the tensor product of the set of dual numbers $\mathbb{D}$ with another set $\mathbb{D}'$, identical except for a distinct nilpotent element ε_2; denote the original nilpotent as ε_1, such that

$$y \in \mathbb{D} \otimes \mathbb{D}' \implies \tag{5.21}$$

$$y = y_0(1 \otimes 1) + y_{1,0}(\varepsilon_1 \otimes 1) + y_{0,1}(1 \otimes \varepsilon_2) + y_{11}(\varepsilon_1 \otimes \varepsilon_2) \tag{5.22}$$

$$(\varepsilon_1 \otimes 1)^2 = (\varepsilon_1^2 \otimes 1) = (0 \otimes 1) = 0 \tag{5.23}$$

$$(1 \otimes \varepsilon_2)^2 = (1 \otimes \varepsilon_2^2) = (1 \otimes 0) = 0 \tag{5.24}$$

$$(\varepsilon_1 \otimes \varepsilon_2)^2 = (\varepsilon_1^2 \otimes \varepsilon_2^2) = (0 \otimes 0) = 0 \tag{5.25}$$

$$(\varepsilon_1 \otimes 1)(1 \otimes \varepsilon_2) = (\varepsilon_1 \otimes \varepsilon_2) \neq 0 \tag{5.26}$$

Numbers from the new algebra can be more simply denoted by using the shorthand:

$$\varepsilon_1 \equiv (\varepsilon_1 \otimes 1) \tag{5.27}$$

$$\varepsilon_2 \equiv (1 \otimes \varepsilon_2) \tag{5.28}$$

$$\varepsilon_{12} \equiv (\varepsilon_1 \otimes \varepsilon_2) \tag{5.29}$$

which results in

$$y = y_0 + y_1\varepsilon_1 + y_2\varepsilon_2 + y_{12}\varepsilon_{12} \tag{5.30}$$

This recreates the definition of the hyper-dual numbers from Section 4.4. The fact that tensor products of hypercomplex algebras yield yet more hypercomplex algebras implies that arbitrarily many tensor products can be combined; with the use of imaginary basis elements, such high-order products can likewise be used to recreate the multicomplex and quasi-complex numbers from Section 4.2.1 and Section 4.2.2. They can further be used to build a "multidual" algebra with a parallel structure to the multicomplex numbers:

$$\mathbb{D}_{(q)} \equiv \mathbb{D} \otimes \mathbb{D}' \otimes \cdots \otimes \mathbb{D}'^{q} \tag{5.31}$$

where $\mathbb{D}'^{p}$ is a dual-like algebra with distinct nilpotent element ε_p. Such a multidual number is a generalization of hyper-dual numbers to many dimensions, and will be revisited in Chapter 6.

As mentioned, strictly speaking the scalar product of two basis elements is distinct from the tensor product. However, the chosen shorthand notation blurs the line between them – for example, $\hat{e}_i\hat{d}_j = (\hat{e}_i \otimes 1)(1 \otimes \hat{d}_j) = (\hat{e}_i \otimes \hat{d}_j)$. This is intentional – the demonstration of multiduals aside, tensor products are intended in this work as a quick way to build differentiation systems where the number of independent variables has no relationship to the order of derivatives to be calculated. The shorthand is only truly a problem in the case of an algebra tensored with itself,[3] where there is no convention to distinguish between $\hat{e}_i \otimes 1$ and $1 \otimes \hat{e}_i$. This problem is done away with by noting that, while piecewise multiplication is the most common way to satisfy closure of the new algebra, there are others – in particular, tensor products of elements can be defined to be equal to other arbitrarily-chosen elements. Keeping in mind that the only goal herein is differentiation, this work simply specifies that when an algebra is tensored with itself, $\hat{e}_i \otimes 1 = 1 \otimes \hat{e}_i$ and $(\hat{e}_i \otimes 1)^2 = \hat{e}_i \otimes \hat{e}_i$. Similarly, all other piecewise multiplication rules are modified to impose tensor commutativity, such that $\hat{e}_i \otimes \hat{e}_j = \hat{e}_j \otimes \hat{e}_i$; this cuts the dimensionality of the system nearly in half and makes multivariate differentiation itself commutative, thereby avoiding redundancy by accounting for symmetry of the Hessian and higher-order derivative structures.

Even more general multiplication rules are possible – tensor products can also result in scalar multiples of other elements, or even zero. This last option can be used to introduce derivative sparsity by removing dimensions from consideration; alternatively, it can be used to assign arbitrary nilpotent degree to the new basis elements. It is useful to express this relationship by stipulating that any nonzero product of shorthand elements is in fact a function of the corresponding tensor product – that is, $\hat{e}_i\hat{e}_j \in \{0, g(\hat{e}_i \otimes \hat{e}_j)\}$ for some function g. This will be revisited in the next section, and becomes especially helpful for the construction of general matrix representations in Chapter 6.

[3] Note that the multiduals are *not* a case of an algebra tensored with itself – $\mathbb{D}$, $\mathbb{D}'$, etc, while similar in structure, are defined to be distinct.

Again, to a large extent all of this flexibility means that tensor products of algebras are simply a different framework for discussing systems capable of higher-order derivatives: one could instead carefully define rules for the scalar product of non-real basis elements (as was done for QCG) that limited the algebra to omit unwanted or redundant calculations. Use of tensor language provides a (hopefully) more understandable approach to targeting a certain number of first derivatives, a certain number of second, and so on – rather than choosing some set of non-real elements and discovering after the fact the dimensionality of the resulting system, this framework allows one to precisely sculpt a new algebra as a combination of first-order systems.

Before continuing with a description of other high-order tensors, it is necessary to adopt a multi-index notation to describe arbitrary tensor products of individual basis elements. Denote a collection of M different multiplicand indices by $\alpha = \{i_1, \ldots, i_M\}$ or $\beta = \{j_1, \ldots, j_M\}$; then, a cumulative tensor product can be described in shorthand as:

$$\hat{e}_\alpha \equiv \bigotimes_{m=1}^{M} \hat{e}_{i_m} \tag{5.32}$$

Again, tensor products of an algebra with itself will be constructed to be commutative. Therefore, in contrast to the multi-index notation more commonly used in other contexts, here α and β do not represent tuples (ordered lists of indices); instead, they should be considered *multisets* – that is, sets (which are defined to be unordered) that allow multiple copies of indices. Here, duplicates of an index indicate repeated multiplication by the same basis element. Because repeated multiplication by $\hat{e}_0 = 1$ does not change an element, zero is excluded from the index multisets; this implies that the real element is actually indexed by the empty set.

Because tensor products of basis elements are also basis elements (if nonzero), it is important to stipulate that each index within a multiset corresponds only to an original element that was *not* constructed via a product, i.e., an element that is indivisible by any other element. For concision, refer to any such indivisible element as a "prime" basis element, and to any non-prime as a "composite" basis element. Using this language, the elements ε_1 and ε_2 from the hyper-duals are prime and ε_{12} is composite. The product of two arbitrarily-indexed elements $\hat{e}_\alpha$ and $\hat{e}_\beta$ can then be indexed by a multiset sum, which is simply a concatenation of the two index lists:

$$\hat{e}_\alpha \otimes \hat{e}_\beta = \hat{e}_{\alpha \uplus \beta} \tag{5.33}$$

$$\alpha \uplus \beta \equiv \{i_1, \dots, i_{M_1}, j_1, \dots, j_{M_2}\} \tag{5.34}$$

$$M_1 = |\alpha| \tag{5.35}$$

$$M_2 = |\beta| \tag{5.36}$$

where the absolute value operator $|\cdot|$ has been used to represent the number of indices in each multiset, i.e., the number of prime elements in each multiplicand; this will also be referred to as the *order* of the basis element (not to be confused with a nilpotent element's *degree*), as it will equal the order of the corresponding derivative. Under this notation, powers of prime elements are represented by repeated indices – for example, the high-order univariate derivatives discussed earlier in Section 5.1 would correspond to nilpotent elements ε_{α_i} indexed by $\alpha_i = \{j, j, j, \dots\}$, such that α_i contains i copies of j. The same notation can be applied to accessor operators, so that the earlier operator Nil_{j^i} is equivalent to $\mathrm{Nil}_{\alpha_i}(x) = x_{\alpha_i}$.

In cases where operations on more than two arbitrary tensor elements are necessary, a subscript-of-a-subscript notation would rapidly become unwieldy. For this reason, every basis element – including composites – is additionally given a one-to-one mapping with a unique singleton index. To distinguish the two, $\{i, j, k, \dots\}$ will henceforth be used to indicate multi-indices, and $\{a, b, c, \dots\}$ will be used to indicate singleton indices – and so, for example, $\hat{e}_a\hat{e}_b = g(\hat{e}_c)$ should be considered a shorthand for $\hat{e}_{\alpha_a}\hat{e}_{\alpha_b} = g(\hat{e}_{\alpha_a} \otimes \hat{e}_{\alpha_b}) = g(\hat{e}_{\alpha_c})$ for arbitrary multi-index sets α_a, α_b, and α_c with $\alpha_c = \alpha_a \uplus \alpha_b$. A singleton index of zero corresponds to the empty multiset $\alpha_0 = \{\}$ and the real-valued identity element 1; beyond this, which singleton index corresponds to which multi-index is arbitrary and usually irrelevant for derivation.

Given this framework, it is easier to discuss hypercomplex algebras capable of finding any desired set of derivatives.

5.3 Generalized Hypercomplex Algebras for AD

To build an algebraic system capable of finding any derivative, begin with a first-order algebra $\mathbb{H}_{(D,1)}$ defined by D distinct nilpotents $\{\varepsilon_1, \dots, \varepsilon_D\}$; this corresponds to a problem with D differentiated inputs. To find derivatives of order M, take the repeated tensor product of $\mathbb{H}_{(D,1)}$ with itself, expressed as a tensor power:

$$\mathbb{H}_{(D,M)} \equiv \mathbb{H}_{(D,1)}{}^{\otimes M} \tag{5.37}$$

Impose tensor commutativity and specify that all tensor basis elements ε_a – both prime and composite – have nilpotent degree $P_a = M - |\alpha_a| + 2$, which ensures that every enumerated element is nonzero while keeping multiplication closed. Insert numbers from this algebra into the same generic multivariate Taylor series used in Equation 5.6:

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = f(\mathbf{x}) + \sum_{m_1=0}^{\infty} \cdots \sum_{m_D=0}^{\infty} \left(\left(\prod_{i=1}^{D} \frac{(h_i \varepsilon_i)^{m_i}}{m_i!} \right) \frac{\partial^{\sum_j m_j}}{\prod_k \partial x_k{}^{m_k}} f(\mathbf{x}) \right) \tag{5.38}$$

$$= f(\mathbf{x}) + \sum_i h_i \varepsilon_i \frac{\partial f}{\partial x_i} + \sum_{i,j} h_i h_j \frac{\varepsilon_i \varepsilon_j}{2!} \frac{\partial^2 f}{\partial x_i \partial x_j} + \sum_{i,j,k} h_i h_j h_k \frac{\varepsilon_i \varepsilon_j \varepsilon_k}{3!} \frac{\partial^3 f}{\partial x_i \partial x_j \partial x_k} + \dots \tag{5.39}$$

where once again $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{S} \in \mathbb{R}^{D \times D}$ is a diagonal matrix with $S_{ii} = h_i$, and $\boldsymbol{\varepsilon} = [\varepsilon_1, \dots, \varepsilon_D]^T$. To simplify notation, denote multi-index powers and derivatives with:

$$\alpha = \{i_1, \dots, i_P\} \tag{5.40}$$

$$h^\alpha \equiv \prod_{p=1}^{P} h_{i_p} \tag{5.41}$$

$$\partial_x^\alpha \equiv \prod_{p=1}^{P} \frac{\partial}{\partial x_{i_p}} \tag{5.42}$$

Accounting for the nilpotent elements, Equation 5.38 then simplifies to the finite series:

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = f(\mathbf{x}) + \sum_{a=1}^{N} \frac{h^{\alpha_a}}{C(\alpha_a)} \left(\partial_x^{\alpha_a} f(\mathbf{x}) \right) \varepsilon_a \tag{5.43}$$

where N is the total number of nilpotent elements of any degree or order and $C(\alpha_a)$ is an integer function that depends on how often individual prime indices are repeated inside α_a. If multiplication and differentiation were not commutative, C would always be equal to $|\alpha_a|!$, but as it is, any element without repeated prime factors will have $C(\alpha_a) = 1$ because it appears multiple times in the summation in Equation 5.38; those with repeats will have $1 < C(\alpha_a) \leq |\alpha_a|!$.

It can now be seen that a number $x \in \mathbb{H}_{(D,M)}$ from the new hypercomplex algebra is defined by the collection of real coefficients $\{x_0, \ldots, x_N\}$ where, for AD-style evaluation of $f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon})$,

$$x_a = \begin{cases} \operatorname{Re}\left(f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon})\right) & = f(\mathbf{x}) \qquad\qquad a = 0 \\ \operatorname{Nil}_a\left(f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon})\right) & = \dfrac{h^{\alpha_a}}{C(\alpha_a)} \partial_x^{\alpha_a} f(\mathbf{x}) \qquad a \geq 1 \end{cases} \tag{5.44}$$

However, there are yet more modifications to the tensor elements that can be used to make differentiation even more convenient. Recall that when building a tensor product of simpler algebras, products of the new tensor elements can be mapped to other elements *or to multiples thereof* – one can trivially specify that $\varepsilon_a\varepsilon_b = g(\varepsilon_a \otimes \varepsilon_b) = g(\varepsilon_c) = C(\alpha_c)\varepsilon_c$. This removes the denominator from Equation 5.44, but more importantly, it will simplify the derivation of arbitrary matrix representations in Chapter 6.

This algebra, as defined, can find every mixed derivative up to order M. To find sparse subsets of these, define an alternate $\mathbb{H}_{(D,M)}{}'$ where some of the tensor elements are defined to be zero or else have smaller nilpotency degree. Consider two examples:

(1) If the first prime nilpotent element has degree $P_1 = M + 1$ but all other $P_{i\neq 1} = 2$ and all $\varepsilon_{i\neq 1}$ are orthogonal both to each other and to ε_1 – i.e., $\varepsilon_i \otimes \varepsilon_j = 0$ for $(i > 1, j \geq 1)$ – then the set is limited to the gradient and all high-order univariate derivatives of f with respect to x_1.

(2) If example (1) is modified such that $\varepsilon_{i\neq 1}$ are not orthogonal to ε_1 but remain so to each other – i.e., $\varepsilon_i \otimes \varepsilon_j = 0$ for $(i \neq 1, j \neq 1)$ but $\varepsilon_i \otimes \varepsilon_j \neq 0$ if $i = 1$ – then the set becomes the gradient and all univariate derivatives thereof with respect to x_1 – that is, every mixed derivative with at most a single ${}^{\partial}\!/_{\partial x_{i\neq 1}}$.

Unfortunately, this new hypercomplex scheme faces the same shortcomings laid out in Chapter 4 for standard dual and hyper-dual numbers – namely, scalar implementation would be indistinguishable from literal differentiation formula, which in turn are less efficient than evaluation using truncated polynomials. Instead, the advantage of this algebraic framework becomes clear when each ε_α is represented by an isomorphic matrix $\mathbf{E}^{(\boldsymbol{\alpha})}$, as will be explored in the next chapter.

Chapter 6

Differentiation via Matrix Representations of Algebras

Thus far, when discussing special algebras useful for differentiation, this work has repeatedly stated that "matrix representations" of the algebras exist. Formally, this means that an *isomorphism* $\mathcal{M}(\cdot)$ exists to map every element of a chosen scalar algebra S to a matrix of a particular structure. This map must be one-to-one, such that every distinct element $x \in S$ corresponds to a single matrix $\check{\mathbf{X}} = \mathcal{M}(x)$, and onto, such that every matrix $\check{\mathbf{Y}}$ with the given structure likewise corresponds to a single scalar element $y \in S$ – equivalent to stating that the map is invertible, with $\mathcal{M}^{-1}\left(\check{\mathbf{Y}}\right) = \mathcal{S}\left(\check{\mathbf{Y}}\right) = y$. Further, to classify as an isomorphism, the map must preserve structure with respect to some arbitrary operation $\bullet$ on S – that is, map to some other operator $*$ on the structured matrices such that $\mathcal{M}(x \bullet y) = \mathcal{M}(x) * \mathcal{M}(y)$. The set of matrices with the given structure then forms its own algebra M. In this work, the isomorphism must in fact form a map for each arithmetic operator: addition, multiplication of elements, multiplication by a scalar, and their respective inverses; as the goal is to use these elements of M in the context of linear algebra packages, these operations on M have to be the basic arithmetic of matrix math. Given an isomorphism with these properties – which will henceforth still be referred to simply as a "matrix representation", for concision – the two algebras are effectively identical unless one introduces a new operation.

This chapter begins with simple examples of matrix representations and a demonstration of differentiation there-with. It then discusses limitations of the approach, all of which amount to the use of operators that do not obey the arithmetic isomorphism or are undefined on either S or M; it also discusses ways to overcome many of these limitations. Next, it expands the discussion to representations of the algebras presented in Chapter 4 before extending their logic to build representations of the hypercomplex algebra types defined in Chapter 5. An algorithm is provided to build matrices capable of representing any such algebra, and thus capable of computing any desired combination of derivatives.

6.1 Introduction: 2×2 Matrix Representations

6.1.1 Complex and Dual Numbers Expressed as Matrices

Probably the most well-known matrix representation is that of the complex numbers:

$$\mathcal{M}_{\mathbb{C}}\left(a+bi\right) \equiv a\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + b\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{6.1}$$

This representation hinges on the fact that the second matrix squares to the negative identity. Following similar logic, it is likewise possible to represent dual numbers by using a matrix that squares to zero [98] and thus is itself nilpotent:

$$\mathcal{M}_{\mathbb{D}}\left(c+d\varepsilon\right) \equiv c\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + d\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} c & d \\ 0 & c \end{bmatrix} \tag{6.2}$$

Indeed, some educational sources [115,116] introduce dual numbers using this definition, foregoing the previous notation entirely. To fully prove these two forms can always represent their respective scalar types, consider three properties: First, simple addition, subtraction, and scalar multiplication or division are performed on an element-wise basis, just as with the canonical forms, and so always result in matrices of the same form – thus, the matrix forms are closed under those operations and respect the corresponding isomorphisms. Second, observe that multiplication of either matrix type is not only closed, but commutative:

$$\begin{gathered}(a_1+b_1i)(a_2+b_2i) = (a_1a_2-b_1b_2)+(a_1b_2+a_2b_1)i \\ \begin{bmatrix} a_1 & -b_1 \\ b_1 & a_1 \end{bmatrix}\begin{bmatrix} a_2 & -b_2 \\ b_2 & a_2 \end{bmatrix} = \begin{bmatrix} a_1a_2-b_1b_2 & -(a_1b_2+a_2b_1) \\ a_1b_2+a_2b_1 & a_1a_2-b_1b_2 \end{bmatrix} = \begin{bmatrix} a_2 & -b_2 \\ b_2 & a_2 \end{bmatrix}\begin{bmatrix} a_1 & -b_1 \\ b_1 & a_1 \end{bmatrix}\end{gathered} \tag{6.3}$$

$$(c_1+d_1\varepsilon)(c_2+d_2\varepsilon) = (c_1c_2-d_1d_2)+(c_1d_2+c_2d_1)\varepsilon$$

$$\begin{bmatrix} c_1 & d_1 \\ 0 & c_1 \end{bmatrix}\begin{bmatrix} c_2 & d_2 \\ 0 & c_2 \end{bmatrix} = \begin{bmatrix} c_1c_2 & c_1d_2 + c_2d_1 \\ 0 & c_1c_2 \end{bmatrix} = \begin{bmatrix} c_2 & d_2 \\ 0 & c_2 \end{bmatrix}\begin{bmatrix} c_1 & d_1 \\ 0 & c_1 \end{bmatrix} \tag{6.4}$$

Third and finally, note that the determinants ($a^2 + b^2$ for complex and c^2 for dual) of these matrices are non-zero only when the real components (and, in the complex case, the imaginary component) are also zero – in other words, multiplicative inverses exist in the same circumstances as in the canonical forms. Combine these properties with the associative and distributive properties of all square matrices, and it can be seen that these matrix representations are in fact isomorphic to their canonical forms – and so, many operations on larger matrices built of blocks of these forms will accurately compute complex or dual-numbered values. Specifically, these forms are compatible with any matrix operation which can be defined blockwise yet still produce a result identical to the element-wise result.

Using this insight, it becomes possible to use matrix-form dual numbers as a new type of automatic differentiation tool – one which is compatible with common linear algebra packages. To the author's knowledge, matrix-form duals have never been used in practice this way [1] in linear algebra problems. Of course, standard complex-step differentiation (CD) has an advantage both in ease of implementation and in computational efficiency: it does not need a matrix form to be used with such third-party packages, since these packages usually provide ordinary complex-number variants of most routines. The purpose of this work, however, is to demonstrate new ways to compute more than a single first-order derivative (following the hypercomplex discussion from Chapter 5), which is beyond CD. It still helps to motivate the work by beginning with an example of standard single-variable differentiation.

6.1.2 A Single-Variable Example – Derivative of a Matrix Inverse

Any matrix $\mathbf{Y} \in \mathbb{D}^{N\times M}$ with dual-valued scalar values can be represented with a purely real-valued matrix $\check{\mathbf{Y}}$ by applying the above isomorphism blockwise:

[1] One anonymous programmer [117] did use dual-number matrices and higher-order univariate constructions as the interior data structure of an operator-overloaded type, but this approach is incompatible with the types of low-level linear algebra libraries considered here, and thus introduces extra overhead for no return – traditional AD is more efficient for operator-overloading approaches.

Further, as will be discussed in Section Section 6.3.1, Lantoine et al [39] demonstrated an alternative dense-matrix form for their multicomplex variables (in turn cited from [96]) which included as a subset the 2×2 matrix representing standard complex numbers, but do not appear to have used the matrix representation in their experiments.

$$\check{\mathbf{Y}} = \mathcal{M}_{\mathbb{D}}\left(\mathbf{Y}\right) = \begin{bmatrix} \mathcal{M}_{\mathbb{D}}\left(Y_{1,1}\right) & \cdots & \mathcal{M}_{\mathbb{D}}\left(Y_{1,M}\right) \\ \vdots & \ddots & \vdots \\ \mathcal{M}_{\mathbb{D}}\left(Y_{N,1}\right) & \cdots & \mathcal{M}_{\mathbb{D}}\left(Y_{N,M}\right) \end{bmatrix} \tag{6.5}$$

$$\mathcal{M}_{\mathbb{D}}\left(Y_{i,j}\right) = \begin{bmatrix} \mathrm{Re}(Y_{i,j}) & \mathrm{Nil}(Y_{i,j}) \\ 0 & \mathrm{Re}(Y_{i,j}) \end{bmatrix} \tag{6.6}$$

To see this ability in action, examine the solution to the canonical matrix equation $\mathbf{Ax} = \mathbf{b}$ with two elements in each vector and consider the original "simulation input" to be the elements in **A** and **b**, with the simulation itself a code that computes the simple inverse (or, more likely, solves the system in place):

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{6.7}$$

This substitution yields

$$\check{\mathbf{x}} = \check{\mathbf{A}}^{-1}\check{\mathbf{b}} \tag{6.8}$$

$$\begin{bmatrix} \mathrm{Re}(x_1) & \mathrm{Nil}(x_1) \\ 0 & \mathrm{Re}(x_1) \\ \mathrm{Re}(x_2) & \mathrm{Nil}(x_2) \\ 0 & \mathrm{Re}(x_2) \end{bmatrix} = \begin{bmatrix} \mathrm{Re}(A_{11}) & \mathrm{Nil}(A_{11}) & \mathrm{Re}(A_{12}) & \mathrm{Nil}(A_{12}) \\ 0 & \mathrm{Re}(A_{11}) & 0 & \mathrm{Re}(A_{12}) \\ \mathrm{Re}(A_{21}) & \mathrm{Nil}(A_{21}) & \mathrm{Re}(A_{22}) & \mathrm{Nil}(A_{22}) \\ 0 & \mathrm{Re}(A_{21}) & 0 & \mathrm{Re}(A_{22}) \end{bmatrix}^{-1} \begin{bmatrix} \mathrm{Re}(b_1) & \mathrm{Nil}(b_1) \\ 0 & \mathrm{Re}(b_1) \\ \mathrm{Re}(b_2) & \mathrm{Nil}(b_2) \\ 0 & \mathrm{Re}(b_2) \end{bmatrix} \tag{6.9}$$

Although the size of the problem has grown, it can be shown that because of the pattern of zeros, computation of this larger matrix inverse is greatly simplified (for sparse algorithms) from a typical 4×4 problem.

To compute a derivative with respect to a single element, simply set one RHS nilpotent coefficient to one and all others to zero, then perform the computation normally. For example, for the derivative of both x-values with respect to A_{12}, the problem becomes:

$$\begin{bmatrix} \mathrm{Re}(x_1) & \mathrm{Nil}(x_1) \\ 0 & \mathrm{Re}(x_1) \\ \mathrm{Re}(x_2) & \mathrm{Nil}(x_2) \\ 0 & \mathrm{Re}(x_2) \end{bmatrix} = \begin{bmatrix} \mathrm{Re}(A_{11}) & 0 & \mathrm{Re}(A_{12}) & 1 \\ 0 & \mathrm{Re}(A_{11}) & 0 & \mathrm{Re}(A_{12}) \\ \mathrm{Re}(A_{21}) & 0 & \mathrm{Re}(A_{22}) & 0 \\ 0 & \mathrm{Re}(A_{21}) & 0 & \mathrm{Re}(A_{22}) \end{bmatrix}^{-1} \begin{bmatrix} \mathrm{Re}(b_1) & 0 \\ 0 & \mathrm{Re}(b_1) \\ \mathrm{Re}(b_2) & 0 \\ 0 & \mathrm{Re}(b_2) \end{bmatrix} \tag{6.10}$$

Using the derivative functionality of dual numbers established in Equation 4.72, the computed solution will be:

$$\check{\mathbf{x}} = \begin{bmatrix} x_1 & \partial x_1/\partial A_{12} \\ 0 & x_1 \\ x_2 & \partial x_2/\partial A_{12} \\ 0 & x_2 \end{bmatrix} \tag{6.11}$$

Here, x_1 and x_2 will have values identical to the solution of a real-valued system. More applicable to real-world tasks is the case where **A** and **b** are functions of some input parameter p.

$$\mathbf{x} = \mathbf{A}(p)^{-1}\mathbf{b}(p) \tag{6.12}$$

Assuming the real values of **A** and **b** are computed by user-written code, any standard AD tool is capable of computing derivatives to construct the augmented RHS matrices:

$$\check{\mathbf{x}} = \check{\mathbf{A}}(p)^{-1}\check{\mathbf{b}}(p) = \begin{bmatrix} \mathrm{Re}(A_{11}) & \partial A_{11}/\partial p & \mathrm{Re}(A_{12}) & \partial A_{12}/\partial p \\ 0 & \mathrm{Re}(A_{11}) & 0 & \mathrm{Re}(A_{12}) \\ \mathrm{Re}(A_{21}) & \partial A_{21}/\partial p & \mathrm{Re}(A_{22}) & \partial A_{22}/\partial p \\ 0 & \mathrm{Re}(A_{21}) & 0 & \mathrm{Re}(A_{22}) \end{bmatrix}^{-1} \begin{bmatrix} \mathrm{Re}(b_1) & \partial b_1/\partial p \\ 0 & \mathrm{Re}(b_1) \\ \mathrm{Re}(b_2) & \partial b_2/\partial p \\ 0 & \mathrm{Re}(b_2) \end{bmatrix} \tag{6.13}$$

Running this through any real-valued matrix solver routine will successfully apply the chain rule and compute $\{\partial x_1/\partial p, \partial x_2/\partial p\}$; importantly, these values will be just as precise and accurate as the values of **x** found from an unaltered system. If the user has post-processing code, the derivatives can be fed forward through yet more standard

AD to find derivatives of any quantities of interest.

Hereafter, matrix substitutions of this type will typically be described as differential operators operating on matrix functions:

$$\mathcal{DM}_p \equiv \begin{bmatrix} 1 & \partial/\partial p \\ 0 & 1 \end{bmatrix} \tag{6.14}$$

When this operator is applied to a matrix, it again replaces each scalar coefficient blockwise:

$$\mathcal{DM}_p(\mathbf{Y}(p)) \equiv \begin{bmatrix} \mathcal{DM}_p(Y_{1,1}) & \cdots & \mathcal{DM}_p(Y_{1,M}) \\ \vdots & \ddots & \vdots \\ \mathcal{DM}_p(Y_{N,1}) & \cdots & \mathcal{DM}_p(Y_{N,M}) \end{bmatrix} \tag{6.15}$$

The block arrangement defined in Equation 6.14 is capable of finding a derivative solely due its special structure; few rearrangements would compute anything of meaning. Moreover, it must interact solely with other blocks of the same structure (be they with a zero or non-zero derivative in the upper right position), otherwise the meaning of the elements will be destroyed. This restriction also applies to precisely *which* derivatives are stored in the upper right – the position must only ever track $\partial/\partial p$; it is not possible for a separate block to find another derivative. Put another way, at program initialization only one nilpotent should be set to one; the only follow-on non-zero nilpotents are those calculated by the natural flow of the program.

For this reason, Equation 6.14 is termed a *stencil* in this work: it must be applied identically to every element in a problem. As will be seen throughout this chapter, more sophisticated stencils are possible – specifically, one can find isomorphic structures for any of the algebras described in Chapter 5. Much of the core contribution of this work can be viewed simply as a search for better stencils.

6.2 Addressing Limitations

There are some obvious limitations to the stencil approach, but many of the worst are easily surmounted. For one, the above substitution transforms the **x** and **b** vectors into two-column matrices, which would often necessitate a different choice of algorithmic method – many functions, especially solvers, expect the right multiplicand to be a vector. This difficulty can be overcome by noting both that matrix multiplication is performed column-wise on the right multiplicand, and that the left columns of both **x** and **b** contain only redundant copies of the real values. A substitution operator for column vectors can thus be defined as

$$\mathcal{DV}_p \equiv \begin{bmatrix} \partial/\partial p \\ 1 \end{bmatrix} \tag{6.16}$$

This vector stencil can use the same data structures and function calls as the original problem and will still produce the correct real and nilpotent values after matrix-vector multiplication or system solution. For this reason, the stencils created in this work will always contain complete copies of all necessary coefficients in the rightmost column. In the case of dual numbers (but not all stencils), a similar substitution can be used for row vectors by using only the top row of $\mathcal{DM}\left(\mathbf{x}^T\right)$; if similar functionality is desired for arbitrary stencils, the derivation presented later in this chapter could easily be modified with new restrictions. Either way, observe that matrix-vector multiplication now has the same computational complexity as the product rule in standard differentiation, albeit at higher memory cost in the matrix.

More generally, as mentioned in the opening to the chapter, stencil representations may fail if operations other than basic matrix arithmetic are applied. This is the case for many (but not all) operations that operate row-by-row, column-by-column, or element by element. For example, the transpose operation applied to a matrix representation of a complex number corresponds to a conjugate of the scalar value. In the case of the CD or MCX methods, this breaks differentiation because the complex-conjugate operation is not complex-analytic; in the case of NMD, the transpose of any stencil matrix $\mathcal{DM}_p(\mathbf{Y})^T$ simply does not have a corresponding scalar variant, and thus will likewise break differentiation and likely scramble downstream real values of a code. (Note that if the derivatives of a given matrix are zero, no harm will be done.)

There are ways to address this, however, for both complex and dual-valued differentiation. In the case of CD, one must use complex scalars (not matrix representations) and avoid the conjugate transpose in favor of the real-valued transpose. For stencils, in certain circumstances it might be possible to apply the matrix operator only after the transpose is complete – i.e., find $\mathcal{DM}_p(\mathbf{Y}^T)$, not $\mathcal{DM}_p(\mathbf{Y})^T$ – but in general, a special operator $(\cdot)^{\check{T}}$ must be defined to compute the transpose blockwise. This operation is slower than the default transpose operation used in standard packages, but in practice remains a very small portion of the total cost of a program. Similarly simple functions can be written to perform many other operations (e.g., the diagonalization of a vector into a matrix) in blockwise fashion.

Interestingly, some algorithms that destroy the stencil structure can still propagate derivative information. For example, even though LU decomposition – which operates row-by-row and column-by-column – not only shuffles the matrix structure, but can suffer from additional matrix fill-in (that is, more than implied by the number of non-zeros in the stencil), it can still be used to find exact derivatives of linear solutions. This phenomenon will be investigated in Part III.

6.3 Representations of Other Existing Algebras

As stated in the introduction to Chapter 4, the scalar algebras discussed there can also be represented as matrices. Two such representations will quickly be reviewed below – the first, for multicomplex numbers, because it provides an already-existing toehold for any-order matrix differentiation, and the second, for multidual numbers, because it allows a much more efficient any-order tool. A simple stencil exclusively for use on univariate high-order derivatives will then be crafted, before moving on to the general multivariate case in Section 6.4.

6.3.1 Multicomplex Numbers

As presented in Section 4.2.1, multicomplex numbers are inductively defined as

$$z \in \mathbb{C}_{(q)} \implies z = z_0 + z_q i_q \quad \left| \begin{array}{l} i_q^2 = -1 \\ z_0, z_q \in \mathbb{C}_{(q-1)} \\ \mathbb{C}_{(1)} \equiv \mathbb{C} \\ \mathbb{C}_{(0)} \equiv \mathbb{R} \end{array} \right. \tag{6.17}$$

As demonstrated in [96] and [39], this inductive definition makes construction of high-order matrix representations relatively easy. Begin with a standard 2×2 complex matrix representing an element of $\mathbb{C}_{(q)}$:

$$\mathcal{M}_{\mathbb{C}_{(q)}}(z) = \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_0 + z_q i_q\right) \equiv \begin{bmatrix} z_0 & -z_q \\ z_q & z_0 \end{bmatrix} \tag{6.18}$$

As this is an element of $\mathbb{C}_{(q)}$, the scalar values are not real numbers, but elements of $\mathbb{C}_{(q-1)}$. Using the same indexing scheme as in Section 4.2.1, one can build matrix forms of z_0 and z_q and then perform a second blockwise substitution:

$$\mathcal{M}_{\mathbb{C}_{(q)}}\left(z_0\right) = \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_{0,0} + z_{0,q-1} i_{q-1}\right) = \begin{bmatrix} z_{0,0} & -z_{0,q-1} \\ z_{0,q-1} & z_{0,0} \end{bmatrix} \tag{6.19}$$

$$\mathcal{M}_{\mathbb{C}_{(q)}}\left(z_q\right) = \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_{0,q} + z_{q,q-1} i_{q-1}\right) = \begin{bmatrix} z_{0,q} & -z_{q,q-1} \\ z_{q,q-1} & z_{0,q} \end{bmatrix} \tag{6.20}$$

$$\mathcal{M}^{(2)}_{\mathbb{C}_{(q)}}(z) = \begin{bmatrix} \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_0\right) & -\mathcal{M}_{\mathbb{C}_{(q)}}\left(z_q\right) \\ \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_q\right) & \mathcal{M}_{\mathbb{C}_{(q)}}\left(z_0\right) \end{bmatrix} = \begin{bmatrix} z_{0,0} & -z_{0,q-1} & -z_{0,q} & z_{q,q-1} \\ z_{0,q-1} & z_{0,0} & -z_{q,q-1} & -z_{0,q} \\ z_{0,q} & -z_{q,q-1} & z_{0,0} & -z_{0,q-1} \\ z_{q,q-1} & z_{0,q} & z_{0,q-1} & z_{0,0} \end{bmatrix} \tag{6.21}$$

From this, a way can be seen to define a recursive multicomplex matrix operator as:

$$\mathcal{M}^{(m+1)}_{\mathbb{C}_{(q)}}(z) = \begin{bmatrix} \mathcal{M}^{(m)}_{\mathbb{C}_{(q)}}\left(z_0\right) & -\mathcal{M}^{(m)}_{\mathbb{C}_{(q)}}\left(z_q\right) \\ \mathcal{M}^{(m)}_{\mathbb{C}_{(q)}}\left(z_q\right) & \mathcal{M}^{(m)}_{\mathbb{C}_{(q)}}\left(z_0\right) \end{bmatrix} \tag{6.22}$$

This recursive operator is then applied inductively from $m = 1$ to $m = q$ until the matrix is composed only of real values. This representation is isomorphic to the first definition of multicomplex numbers – one can successfully compute the product of two multicomplex numbers by multiplying two real-valued matrices of this form, and if one cared to perform any linear algebra manipulations on multicomplex-valued matrices, a blockwise substitution of this form would turn the problem into one with real values.

When building the input matrix or examining the output results, it becomes (possibly) more intuitive to reinterpret "the product of two distinct imaginary terms" as "the imaginary component of another imaginary" (in the above, $z_{q,q-1}$) – that is, to find a given product of two or more terms, one looks for the block corresponding to the first term, and then, within that block, finds the sub-block or real value corresponding to the next term. This corresponds to the alternate tensor representation of multicomplex numbers, wherein the tensor product of two imaginary elements $i_1 \otimes i_2$ belongs to an entirely different algebra from the original i_1 and i_2 – the former has a 4×4 representation, while the latter has 2×2; instead, one must refer to $i_1 \otimes 1$, or "the real component of an imaginary value".

From this procedure, one can see that blockwise induction provides at least one way to represent a multivariate algebra – one that has immediate, obvious parallels to dual numbers. Before moving on, it is important to acknowledge that in their work on multicomplex differentiation, Lantonine et al [39] mentioned in passing that the above matrix representation allows one to use real-valued linear algebra operations to find derivatives with MCX. However, they appear not to have utilized this capability in their demonstrations – likely due to the fact that MCX would have required the evaluation of dense matrices of dimension $2^q \times 2^q$.

6.3.2 Multidual Numbers

As seen with multicomplex numbers, there is an intuitive way to recursively build higher-order matrix representations of many variables. To strip out much (but not all) of the redundant information contained in a MCX stencil, begin by re-stating the multidual definition from Equation 5.31 with the same inductive pattern as for $\mathbb{C}_{(q)}$:

$$y \in \mathbb{D}_{(q)} \implies y = y_0 + y_q \varepsilon_q \left| \begin{array}{l} \varepsilon_q^2 = 0 \\ y_0, y_q \in \mathbb{D}_{(q-1)} \\ \mathbb{D}_{(1)} \equiv \mathbb{D} \\ \mathbb{D}_{(0)} \equiv \mathbb{R} \end{array} \right. \tag{6.23}$$

To apply this method to dual numbers, begin with the original substitution:

$$\mathcal{M}^{(1)}_{\mathbb{D}_{(q)}}(y) = \mathcal{M}^{(1)}_{\mathbb{D}_{(q)}}\left(y_0 + y_q \varepsilon_q\right) \equiv \begin{bmatrix} y_0 & y_q \\ 0 & y_0 \end{bmatrix} \tag{6.24}$$

Define a *recursive multidual (RMD)* stencil as:

$$\mathcal{M}_{\mathbb{D}_{(q)}}^{(m+1)}(y) = \begin{bmatrix} \mathcal{M}_{\mathbb{D}_{(q)}}^{(m)}(y_0) & \mathcal{M}_{\mathbb{D}_{(q)}}^{(m)}(y_q) \\ \mathbf{0} & \mathcal{M}_{\mathbb{D}_{(q)}}^{(m)}(y_0) \end{bmatrix} \tag{6.25}$$

where $\mathbf{0}$ is an all-zero matrix of the same size as the operators. When $m = q$, the substitution has reached maximum recursive depth and the matrix is real-valued, in which case the operator will simply be denoted as $\mathcal{M}_{\mathbb{D}_{(q)}}$. Cases where $m < q$ are non-real and only useful as intermediate steps in stencil construction; $m > q$ is undefined. The isomorphism to a differential operator still holds – this time to an operator on a function of many inputs:

$$\mathcal{DM}_{\mathbf{x}}^{(1)}\left(y(\mathbf{x})\right) \equiv \begin{bmatrix} 1 & \partial/\partial x_1 \\ 0 & 1 \end{bmatrix} \tag{6.26}$$

$$\mathcal{DM}_{\mathbf{x}}^{(m+1)}\left(y(\mathbf{x})\right) \equiv \begin{bmatrix} \mathcal{DM}_{\mathbf{x}}^{(m)}(y) & \frac{\partial}{\partial x_{m+1}}\left(\mathcal{DM}_{\mathbf{x}}^{(m)}(y)\right) \\ 0 & \mathcal{DM}_{\mathbf{x}}^{(m)}(y) \end{bmatrix} \tag{6.27}$$

Here, m ranges from 1 to the number of elements in the input vector $\mathbf{x}$. Applying this operation for a two-element input vector $[x_1, x_2]^T$ yields:

$$\mathcal{DM}_{\mathbf{x}}^{(2)}\left(y(x)\right) \equiv \begin{bmatrix} 1 & {}^{\partial}/_{\partial x_1} & {}^{\partial}/_{\partial x_2} & {}^{\partial^2}/_{\partial x_{12}^2} \\ 0 & 1 & 0 & {}^{\partial}/_{\partial x_2} \\ 0 & 0 & 1 & {}^{\partial}/_{\partial x_1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.28}$$

Just as with the multicomplex case, substituting stencils of this form into linear algebra problems can compute mixed derivatives; note that if $x_1 = x_2$, the operator will find univariate derivatives (albeit less efficiently than the stencil described ahead in Section 6.3.3). The substitution can be carried out arbitrarily many times, each time applying the operator with respect to a different input. Just as with the first-order operator, column stencils can sometimes discard all but the last column of the matrix form:

$$\mathcal{DV}_{\mathbf{x}}^{(2)}\left(y(x)\right) \equiv \begin{bmatrix} \partial^2/\partial x_{12}^2 \\ \partial/\partial x_2 \\ \partial/\partial x_1 \\ 1 \end{bmatrix} \tag{6.29}$$

This is a fairly naive construction technique which, for q different independent differentiation variables, grows stencil dimensions to $2^q \times 2^q$, just as MCX does. It can already be seen, however, that this method yields sparse matrices – and the deeper the recursion, the sparser the matrix. Along with the univariate stencil developed in Section 6.3.3, it provides a ready example to consider when developing a wider stencil approach and was invaluable in the earliest stages of developing NMD.

6.3.3 High-Order Univariate Nilpotents

Consider the 2×2 matrix used to represent the nilpotent element of standard dual numbers:

$$\mathcal{M}_{\mathbb{D}}\left(\varepsilon\right) \equiv \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{6.30}$$

This matrix can be thought of as the 2×2 instance of an *upper shift matrix* $\mathbf{U^{(S)}}$ – that is, a matrix with ones along the superdiagonal (immediately above the main diagonal) and zero everywhere else. The name comes from the fact that multiplication by a shift matrix moves all elements of another matrix or vector by one position – for example:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 0 & 0 & 0 \end{bmatrix} \tag{6.31}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 4 & 5 \\ 0 & 7 & 8 \end{bmatrix} \tag{6.32}$$

As can be seen, premultiplication by $\mathbf{U}^{(\mathbf{S})}$ shifts a matrix up one row and introduces a new row of zeros; post-multiplication shifts a matrix to the right and introduces a new column of zeros.

Shift matrices – like all triangular matrices that exist entirely above the main diagonal – are nilpotent, for reasons that are clear to see. Squaring a shift matrix moves the superdiagonal a further step away from the main, and each successive power moves the diagonal one step further, until it disappears entirely:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.33}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.34}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.35}$$

From this, it can be seen that $N \times N$ upper shift matrices are nilpotent elements of degree N. Following the same logic as [117], this provides one possible matrix structure for the univariate algebra defined in Section 5.1. The new basis elements – termed *basis stencils* or *unit stencils* – thus take the form:

$$\mathcal{M}\left(1\right) \equiv \mathbf{E}^{(0)} = \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.36}$$

$$\mathcal{M}\left(\varepsilon_1\right) \equiv \mathbf{E}^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.37}$$

$$\mathcal{M}\left({\varepsilon_1}^2\right) = \mathcal{M}\left(\varepsilon_{11}\right) = {\mathbf{E}^{(1)}}^2 \equiv \mathbf{E}^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.38}$$

$$\mathcal{M}\left({\varepsilon_1}^3\right) = \mathcal{M}\left(\varepsilon_{111}\right) = {\mathbf{E}^{(1)}}^3 \equiv \mathbf{E}^{(3)} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.39}$$

where the parenthetical matrix labels use the same singleton index shorthand described in Chapter 5 for tensor elements, i.e., actually represent the index multisets $\alpha_1 = \{1\}$, $\alpha_2 = \{1, 1\}$, and $\alpha_3 = \{1, 1, 1\}$. (Note that setting ${\varepsilon_1}^3 = \varepsilon_{111} = \varepsilon_1 \otimes \varepsilon_1 \otimes \varepsilon_1$, etc., is equivalent to setting $g(\cdot)$ from Section 5.2 to the identity function.) The expansion from Equation 5.10 still applies:

$$f\Big(\mathcal{M}\left(x + h\varepsilon_1\right)\Big) = f\left(x\mathbf{E}^{(0)} + h\mathbf{E}^{(1)}\right) \tag{6.40}$$

$$= f(x)\mathbf{E}^{(0)} + \frac{h}{1!}\frac{\partial f(x)}{\partial x}\mathbf{E}^{(1)} + \frac{h^2}{2!}\frac{\partial^2 f(x)}{\partial x^2}\mathbf{E}^{(2)} + \frac{h^3}{3!}\frac{\partial^3 f(x)}{\partial x^3}\mathbf{E}^{(3)} \tag{6.41}$$

One possible high-order stencil for an Nth-order univariate derivative is thus the $(N + 1) \times (N + 1)$ operator:

$$\mathcal{DM}_x^{(N)} \equiv \mathbf{E}^{(0)} + \sum_{a=1}^{N} \frac{1}{a!}\mathbf{E}^{(a)}\frac{\partial^a}{\partial x^a} = \begin{bmatrix} 1 & {}^{\partial}/_{\partial x} & \frac{{}^{\partial^2}/_{\partial x^2}}{2} & \frac{{}^{\partial^3}/_{\partial x^3}}{6} & \cdots & \frac{{}^{\partial^N}/_{\partial x^N}}{N!} \\ 0 & 1 & {}^{\partial}/_{\partial x} & \frac{{}^{\partial^2}/_{\partial x^2}}{2} & \ddots & \vdots \\ 0 & 0 & 1 & {}^{\partial}/_{\partial x} & \ddots & \frac{{}^{\partial^3}/_{\partial x^3}}{6} \\ 0 & 0 & 0 & 1 & \ddots & \frac{{}^{\partial^2}/_{\partial x^2}}{2} \\ 0 & 0 & 0 & 0 & \ddots & {}^{\partial}/_{\partial x} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.42}$$

This is only a very simple example – note that unit stencils need not be composed entirely of zeros and ones to be isomorphic to particular basis elements. For example, consider the following alternate definition of $\mathbf{E^{(1)}}$ for $N = 5$:

$$\mathcal{M}\left(\varepsilon_1\right) \equiv \mathbf{E^{(1)}} = \begin{bmatrix} 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.43}$$

This matrix is still nilpotent with degree six, and hence can represent the same scalar nilpotent element. Next, recall from Section 5.3 that composite tensor-product basis elements need not correspond directly to the literal product of other elements – scalar multiples are also possible. To make this more clear, consider another alternate definition for the remaining basis stencils (again, see Section 5.2 for the usage of g and Section 5.3 for C):

$$\varepsilon_1{}^a = g(\varepsilon_1{}^{\otimes a}) = C(\alpha_a)\varepsilon_1{}^{\otimes a} = a!\varepsilon_1{}^{\otimes a} \tag{6.44}$$

$$\varepsilon_1{}^{\otimes a} = \frac{1}{a!}\varepsilon_1{}^a \tag{6.45}$$

$$\mathcal{M}\left(\varepsilon_1{}^{\otimes a}\right) = \mathcal{M}\left(\varepsilon_a\right) = \mathbf{E^{(a)}} \tag{6.46}$$

$$\mathbf{E^{(a)}} \equiv \frac{1}{a!}\mathbf{E^{(1)}}^a \tag{6.47}$$

Of course, this logic simply folds the coefficient from Equation 6.42 into the definition of the higher-order unit stencils, but combined with the new $\mathbf{E^{(1)}}$ given by Equation 6.43, it leads to a more elegant full stencil:

$$\mathcal{DM}_x^{(N)} \equiv \mathbf{E^{(0)}} + \sum_{n=1}^{N} \mathbf{E^{(n)}}\frac{\partial^n}{\partial x^n} = \begin{bmatrix} 1 & 5\partial/\partial x & 10\partial^2/\partial x^2 & 10\partial^3/\partial x^3 & 5\partial^4/\partial x^4 & \partial^5/\partial x^5 \\ 0 & 1 & 4\partial/\partial x & 6\partial^2/\partial x^2 & 4\partial^3/\partial x^3 & \partial^4/\partial x^4 \\ 0 & 0 & 1 & 3\partial/\partial x & 3\partial^2/\partial x^2 & \partial^3/\partial x^3 \\ 0 & 0 & 0 & 1 & 2\partial/\partial x & \partial^2/\partial x^2 \\ 0 & 0 & 0 & 0 & 1 & \partial/\partial x \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.48}$$

Of course, these stencil properties were not selected at random; they were carefully chosen so that the rightmost column contains only the derivatives, without any coefficient. Further, although it was not an explicit goal in any algorithm, note that for the special case of matrix-vector multiplication the new stencil has recreated Leibniz's rule (i.e., the high-order product rule). More than this, though, the algebraic analysis has proven that the entire stencil – not just the last column – will be preserved across matrix-matrix operations. In Section 6.4, this approach will make derivation easier for stencils with mixed derivatives.

6.4 Representing Aribtrary Hypercomplex Algebras with Stencils

As discussed in the introduction to this chapter, for a matrix representation to be isomorphic to a particular algebra the core matrix arithmetic operations – addition, multiplication of elements, and multiplication by a scalar – must produce a matrix with a structure that maps to the correct corresponding scalar sum or product. Note that addition and scalar multiplication are trivially isomorphic, meaning that only matrix multiplication needs to be examined in any depth. Thus, to construct an arbitrary representation, begin by considering the multiplication of any two scalar hypercomplex numbers x and y:

$$y = \sum_{a=0}^{N} y_a \varepsilon_a \tag{6.49}$$

$$z = \sum_{b=0}^{N} z_b \varepsilon_b \tag{6.50}$$

$$\begin{aligned} f = yz &= \left(\sum_{a=0}^{N} y_a \varepsilon_a\right)\left(\sum_{b=0}^{N} z_b \varepsilon_b\right) \\ &= \sum_{a=0}^{N}\sum_{b=0}^{N} y_a z_b \varepsilon_a \varepsilon_b \\ &= \sum_{c=0}^{N} f_c \varepsilon_c \end{aligned} \tag{6.51}$$

Now presume y and z are each carrying derivatives with respect to some vector of parameters $\mathbf{x}$:

$$y = y(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = \sum_{a=0}^{N} \frac{h^{\alpha_a}}{C(\alpha_a)} \partial_x^{\alpha_a}(y)\varepsilon_a \tag{6.52}$$

$$z = z(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = \sum_{b=0}^{N} \frac{h^{\alpha_b}}{C(\alpha_b)} \partial_x^{\alpha_b}(z)\varepsilon_b \tag{6.53}$$

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = f(y(\mathbf{x}), z(\mathbf{x})) = y \cdot z = \left(\sum_{a=0}^{N} \frac{h^{\alpha_a}}{C(\alpha_a)} \partial_x^{\alpha_a}(y)\varepsilon_a\right)\left(\sum_{b=0}^{N} \frac{h^{\alpha_b}}{C(\alpha_b)} \partial_x^{\alpha_b}(z)\varepsilon_b\right) \tag{6.54}$$
$$= \sum_{c=0}^{N} f_c \varepsilon_c$$

where a, b, and c are singleton indices; N is the total number of nilpotent elements, including composites; and multi-index operations and shorthand are as defined throughout Chapter 5. This leads to

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = \sum_{c=0}^{N} f_c \varepsilon_c \tag{6.55}$$

$$= \sum_{c=0}^{N} \frac{h^{\alpha_c}}{C(\alpha_c)} \partial_x^{\alpha_c}(f)\varepsilon_c = \sum_{a=0}^{N}\sum_{b=0}^{N} \frac{h^{\alpha_a}h^{\alpha_b}}{C(\alpha_a)C(\alpha_b)} \partial_x^{\alpha_a}(y)\partial_x^{\alpha_b}(z)\varepsilon_a\varepsilon_b \tag{6.56}$$

The calculation of any given f_c in isolation is made more complicated by the fact that many different possible pairings of multiplicands $\varepsilon_a\varepsilon_b$ will yield the same ε_c – i.e., ε_c has many different *factor pairs*, which can be prime or composite. For example, for $\alpha_c = \{1, 1, 2, 3\}$, the following products all hold:

$$g(\varepsilon_{1123}) = \varepsilon_0\varepsilon_{1123} = 1 \cdot \varepsilon_{1123}$$
$$= \varepsilon_1\varepsilon_{123}$$
$$= \varepsilon_{11}\varepsilon_{23}$$
$$= \varepsilon_{12}\varepsilon_{13}$$
$$= \varepsilon_2\varepsilon_{113}$$
$$= \varepsilon_3\varepsilon_{112}$$

This same logic implies that the products of powers $h^{\alpha_a}h^{\alpha_b}$ are all equal to h^{α_c}. In contrast, pairs of coefficient functions $C(\alpha_a)C(\alpha_b)$ are *not* guaranteed to equal $C(\alpha_c)$. For example, as mentioned in Section 5.3, any composite element without repeated prime indices will have $C = 1$ to account for summation of equivalent commutative products in the multivariate Taylor series, but those that do contain repeats will have $C > 1$. So, e.g., if $\alpha_a = \alpha_b = \{1,2,3\}$, then $C(\alpha_a) = C(\alpha_b) = 1$ – and yet, $C(\alpha_c) = C(\{1,1,2,2,3,3\}) \neq 1$. This is quickly overcome by redefining scalar multiplication of nilpotents following the rule established in Section 5.3: if one specifies that for all ε_p and ε_q, $\varepsilon_p\varepsilon_q = g(\varepsilon_p \otimes \varepsilon_q) = g(\varepsilon_r) = C(\alpha_r)\varepsilon_r$, then the C-terms will cancel in every Taylor evaluation, reducing Equation 6.56 to

$$f(\mathbf{x} + \mathbf{S}\boldsymbol{\varepsilon}) = \sum_{c=0}^{N} h^{\alpha_c}\partial_x^{\alpha_c}(f)\varepsilon_c = \sum_{a=0}^{N}\sum_{b=0}^{N} h^{\alpha_c}\partial_x^{\alpha_a}(y)\partial_x^{\alpha_b}(z)\varepsilon_a\varepsilon_b = \sum_{a=0}^{N}\sum_{b=0}^{N} h^{\alpha_c}\partial_x^{\alpha_a}(y)\partial_x^{\alpha_b}(z)C(\alpha_c)\varepsilon_c \tag{6.57}$$

Note that if all h are initialized to 1, then each coefficient of y, z, and f is exactly equal to a corresponding derivative. Putting aside the actual calculation of C for the moment, using this scheme it is possible to calculate yz via a brute-force double loop over all basis elements, summing the results of each individual pair of factors as they are encountered. However, to represent elements in matrix form (i.e., as stencils), each f_c must be computed in isolation as the inner product of one stencil row and one stencil column; this implies that any stencil-building algorithm must be capable of enumerating all factor pairs $(\varepsilon_a, \varepsilon_b)$, just as in the example for ε_{1123}. The row-column matching also requires a distinction between left factors and right factors, which will come in pairs indexed by a and b, as above. (Making this distinction is equivalent to temporarily removing the commutativity property of the algebra.)

It is possible to find a full list of factor pairs by exhaustively computing all two-set partitions of α_c. The first step in this process is to compute all the left factors α_a by running an n-choose-k algorithm $\texttt{nCk}(\alpha_c, \texttt{k})$ – i.e., one that finds all k-index combinations of the multiset of $n = |\alpha_c|$ indices – for each $k = \{0, ..., n\}$. After this, each paired right factor α_b can be found by a multiset difference operation $\alpha_b = \alpha_c \setminus \alpha_a$, which amounts to removing a copy of every index found in α_a from α_c.

If α_c contains n indices, the resulting list of factors will contain 2^n pairs. However, if α_c contains any repeated indices, the list will likewise contain repeated pairs. For example, consider the factors of ε_{123} and ε_{112}, as required to compute f_{123} and f_{112}:

$$f_{123}\varepsilon_{123} = y_0 z_{123}\varepsilon_0\varepsilon_{123} + y_1 z_{23}\varepsilon_1\varepsilon_{23} + y_2 z_{13}\varepsilon_2\varepsilon_{13} + y_3 z_{12}\varepsilon_3\varepsilon_{12}$$

$$+y_{12}z_3\varepsilon_{12}\varepsilon_3 + y_{13}z_2\varepsilon_{13}\varepsilon_2 + y_{23}z_1\varepsilon_{23}\varepsilon_1 + y_{123}z_0\varepsilon_{123}\varepsilon_0$$

$$\begin{aligned} f_{112}\varepsilon_{112} =& y_0z_{112}\varepsilon_0\varepsilon_{112} + y_1z_{12}\varepsilon_1\varepsilon_{12} + y_1z_{12}\varepsilon_1\varepsilon_{12} + y_2z_{11}\varepsilon_2\varepsilon_{11} \\ &+y_{11}z_2\varepsilon_{11}\varepsilon_2 + y_{12}z_1\varepsilon_{12}\varepsilon_1 + y_{12}z_1\varepsilon_{12}\varepsilon_1 + y_{112}z_0\varepsilon_{112}\varepsilon_0 \end{aligned}$$

The second and third factor pairs of ε_{112} are duplicates, as are the sixth and seventh. (If one accounts for commutativity, there are even more duplicates, but for matrix-math purposes left and right factors must remain distinct.) The sum can thus be more compactly expressed as:

$$\begin{aligned} f_{112}\varepsilon_{112} =& y_0z_{112}\varepsilon_0\varepsilon_{112} + 2y_1z_{12}\varepsilon_1\varepsilon_{12} + y_2z_{11}\varepsilon_2\varepsilon_{11} \\ &+y_{11}z_2\varepsilon_{11}\varepsilon_2 + 2y_{12}z_1\varepsilon_{12}\varepsilon_1 + y_{112}z_0\varepsilon_{112}\varepsilon_0 \end{aligned}$$

So, rather than run the full double loop, it is possible to pre-compute a list of factor pairs and coefficients to be used during each multiplication; this can be expressed as a set of 3-tuples:

$$F(c) = \left\{ \left(a_1^{(c)}, b_1^{(c)}, m_1^{(c)}\right), \left(a_2^{(c)}, b_2^{(c)}, m_2^{(c)}\right), \dots \right\} \tag{6.58}$$

As an example, for $\alpha_c = \{1, 1, 2\}$ this becomes

$$\begin{aligned} f_{112}\varepsilon_{112} =& \sum_{k=1}^{|F(112)|} m_k y_{a_k} z_{b_k} \varepsilon_{a_k} \varepsilon_{b_k} \\ =& \quad y_0z_{112}\varepsilon_0\varepsilon_{112} \\ &+ 2y_1z_{12}\varepsilon_1\varepsilon_{12} \\ &+ \quad y_2z_{11}\varepsilon_2\varepsilon_{11} \\ &+ \quad y_{11}z_2\varepsilon_{11}\varepsilon_2 \\ &+ 2y_{12}z_1\varepsilon_{12}\varepsilon_1 \\ &+ \quad y_{112}z_0\varepsilon_{112}\varepsilon_0 \end{aligned}$$

$$F(c) = \left\{ \Big(\{\}, \{1, 1, 2\}, 1 \Big), \Big(\{1\}, \{1, 2\}, 2 \Big), \dots \right\} \tag{6.59}$$

Once this set is pre-computed, individual coefficients of hypercomplex products could be directly computed within a user program as

$$f_c \varepsilon_c = \sum_{a=1}^{N} \sum_{b=1}^{N} y_a z_b \varepsilon_a \varepsilon_b \tag{6.60}$$

$$= \sum_{(a_k, b_k, m_k) \in F(c)} m_k y_{a_k} z_{b_k} \varepsilon_{a_k} \varepsilon_{b_k} \tag{6.61}$$

This list-of-factors scheme allows faster implementation of scalar multiplication[2] , but it is only of use for matrix multiplication if each element index a, b, or c can be mapped to a valid corresponding (row, column) entry or set of entries; further, this mapping must force the inner products of rows and columns to compute the appropriate coefficients $\boldsymbol{C}$. There are innumerable ways to build such a mapping, although most are difficult to derive for an arbitrary list of basis elements. Fortunately, imposing two restrictions narrows the list of possible mappings in a way that makes derivation easier: first, assume that for N distinct nilpotents, the rightmost column of the complete $(N+1) \times (N+1)$ stencil contains a single copy of each coefficient (including the real) – i.e., assume that each unit stencil has a single entry in the last column, which also serves to allow for matrix-vector multiplication of stencils. Second, specify that this column has no coefficients, i.e., that those rightmost unit stencil entries are all one. This forces all calculation of $\boldsymbol{C}$ into the other columns; if one can satisfy the two requirements, $\boldsymbol{C}$ will be found implicitly.

All that remains is to build a representation that under multiplication places a copy of each f_c in the right column – a task that is comparatively easy compared to all of the preceding discussion. First, define an index function $i(c)$ which maps one-to-one from an element's singleton index to that element's location (i.e, row) in the rightmost column. To sort all non-zeros into the upper triangular part of the stencil, the algorithm built in this work sorts the rows such that the highest-order elements (i.e., those with the most prime factors) are mapped to the top of the column.

[2] It is faster by virtue of the fact that it again indirectly implements the Leibniz formula for multivariable derivatives – m amounts to a coefficient in the rule, and $\boldsymbol{C}$ amounts to a method to implicitly cancel any difficult-to-compute denominators in the Taylor series. Note, however, that (1) pre-computing the number of repeated factors is generally simpler that the actual formula (which is rather complicated for the multivariable case), and (2) the hypercomplex framework grants a firmer analytic framework – it makes it clearer that the algebraic structure admits an isomorphic stencil that can repeat itself across successive additions, products, and inversions, and gives a way to address functions that are non-isomorphic and thus might break the chain of computation.

Note that – just as with shift matrices – limiting the stencil to the upper triangle automatically makes every element other than $\mathbf{E^{(0)}}$ nilpotent, which again makes derivation easier. However, $i(c)$ can in truth be arbitrary – as it merely defines the column's ordering, different definitions will simply result in different permutations of the same stencil.

Next, for each row i, arrange factor coefficients across columns j such that the inner product with the rightmost column computes Equation 6.61:

$$\check{\mathbf{Y}} = \mathcal{DM}\left(y\right) \tag{6.62}$$

$$\check{\mathbf{Z}} = \mathcal{DM}\left(z\right) \tag{6.63}$$

$$\check{\mathbf{F}} = \mathcal{DM}\left(f\right) = \check{\mathbf{Y}}\check{\mathbf{Z}} \tag{6.64}$$

$$\forall c \, : \, \left[\check{\mathbf{Y}}_{(i(c),\cdot)}\right]\left[\check{\mathbf{Z}}_{(\cdot,N)}\right] = \sum_{j=1}^{N} \check{Y}_{(i(c),j)}\check{Z}_{(j,N)} = \sum_{(a,b,m)\in F(c)} m y_a z_b \tag{6.65}$$

Note that $|F(c)| \leq N$, as there cannot be more factors than elements, and that every z_b has been forced to appear in the rightmost column $\check{\mathbf{Z}}_{(\cdot,N)}$. And so, by singling out each non-zero $y_a z_b$, it is possible to define a map $j(a)$ that assigns the appropriate m-value to every scalar entry $y_a = \check{Y}_{(i(c),j(a))}$ in the left matrix multiplicand from Equation 6.65. To pair y_a and z_b, note that the right matrix is row-indexed by j and simply stipulate that $j(a) = i(b)$. Thus, only one index map is required.

To break $\mathcal{DM}\left(\cdot\right)$ into its constituent unit stencils, each $(i(c), i(b))$ (row, column) entry must correspond to the matching left-hand basis element $\mathbf{E^{(a)}}$, so that

$$\forall (a, b, m) \in F(c) \, : \, E^{(a)}_{i(c),\, i(b)} = m \tag{6.66}$$

In practice, it is easiest to build the stencils by looping over c, rather than a – that is, by building them in parallel, rather than sequentially. The full procedure is shown in pseudocode in Algorithm 6.1, which can be used to build matrix isomorphisms for any of the nilpotent algebras presented in Chapter 5 – and, thus, yields a ways to find any derivative of a matrix function.

Algorithm 6.1 A Stencil-Building Algorithm Compatible with Column Vectors

```
1. Given:
    $\{\boldsymbol{\beta}^{(1)}, \dots, \boldsymbol{\beta}^{(M)}\}$                          // Collection of desired element index multisets
    s.t. $\forall p : \boldsymbol{\beta}^{(p)} \equiv \{\beta_{(p,1)}, \dots, \beta_{(p,N_p)}\}$   // Sets-of-multisets – elements grouped by order $p$
    s.t. $\forall p, d : \beta_{(p,d)} \equiv \{i_1^{(d)}, \dots, i_p^{(d)}\}$             // Each order-$p$ element is defined by $p$ indices

2. Initialize:
    $\forall p : \boldsymbol{\gamma}^{(p)} := \boldsymbol{\beta}^{(p)}$                        // Start set of *required* elements, including missing factors
    for $p = M, (M-1), \dots, 2$                          // Factorize elements of each order $p$, starting with highest
        for $d = 1, \dots, |\boldsymbol{\gamma}^{(p)}|$                        // Every element $d$ of given order
            for $q = (p-1), \dots, 1$                     // Factors of every possible order $q$
                $\boldsymbol{\gamma}^{(q)} := \boldsymbol{\gamma}^{(q)} \cup \mathtt{nCk}(\gamma_{(p,d)}, q)$        // $n$-choose-$q$ enumerates all possible $q$-element factors of $\gamma_{(p,d)}$
    Result: $\boldsymbol{\gamma}^{(1)}, \dots, \boldsymbol{\gamma}^{(M)}$                     // Multi-index set of every tensor element in the algebra
    Result: $N := \sum_p |\boldsymbol{\gamma}^{(p)}|$                     // Number of non-real elements (both prime and composite)

3. Find Factor Pairs:
    $\mathbf{M} := \mathbf{0} \in \mathbb{Z}^{N \times N}$                          // Initialize table of integer coefficients $m$
    for $c = 1, \dots, N$                                // For each element's list of factor pairs
        $F(c) := \{\}$                                   // Initialize empty list
    for $p = 1, \dots, M$                                // Factorize elements of each order $p$ into $\varepsilon_c = \sum_{(a,b,m)} m\varepsilon_a\varepsilon_b$
        for $q = 1, \dots, |\boldsymbol{\gamma}^{(p)}|$                        // Every element $q$ of given order
            $\alpha_c := \gamma_{(p,q)}$                            // Look up product singleton index $c$
            for $r = 0, \dots, q$                        // For every possible order of left multiplicand
                $\boldsymbol{\eta} := \mathtt{nCk}(\alpha_c, r)$                  // Enumerate all possible order-$r$ left multiplicands of $\alpha_c$
                for $s = 1, \dots, |\boldsymbol{\eta}|$                  // For every enumerated left multiplicand $s$
                    $\alpha_a := \eta_s$                         // Look up left multiplicand singleton index $a$
                    $\alpha_b := \alpha_c \setminus \alpha_a$                    // Find right multiplicand with multiset difference; look up $b$
                    $F(c) := F(c) \cup \{(a,b)\}$            // Append to factor list
                    $M_{c,b} := M_{c,b} + 1$                 // Increment coefficient $m$
    Result: $F$                                          // Completed lists of all factor pairs for each element
    Result: $\mathbf{M}$                                          // Summation coefficients for each factor pair

4. Build Stencils:
    for $a = 1, \dots, N$                                // For each element unit stencil
        $\mathbf{E}^{(a)} := \mathbf{0} \in \mathbb{Z}^{N \times N}$                     // Initialize empty stencil
    for $c = 1, \dots, N$                                // For each element's list of factor pairs
        for $p = 1, \dots, |F(c)|$                       // For each factor pair of $\varepsilon_c$
            $(a,b) := F(c)_p$                            // Extract factor indices
            $m := M_{c,b}$                               // Extract factor coefficient
            $i := i(c)$                                  // Unit stencil row index
            $j := i(b)$                                  // Unit stencil column index
            $E^{(a)}_{i,j} := m$                              // Insert unit stencil coefficients

5. Output:
    $\{\mathbf{E}^{(\alpha_1)}, \dots, \mathbf{E}^{(\alpha_N)}\}$                         // Collection of all unit stencils
```

Chapter 7

Hybrid Methods and Directional Derivatives

As discussed in Section 3.3.2, AD tools for a given order of derivative can be nested one inside another to find higher orders. This nesting approach is normally used inductively with a single type of tool, but intuitively, it can be used to build arbitrary hybrids of different methods. Recall that, in addition to presenting Nilpotent Matrix Differentiation for higher orders, this work has also reviewed two existing first-order matrix-compatible forms of differentiation: the adjoint method (Section 3.1.2) and complex-step differentiation (Section 4.1). The three pairwise hybrids of these methods each have promising uses, as discussed below; arguably, each combination shows more promise for practical high-order differentiation than NMD in isolation.

For the sake of simplicity, the derivation of NMD has thus far neglected directional derivatives; however, the upcoming interplay of methods makes them difficult to ignore any further. For this reason, each method will be presented from the start in a directional context, which finally gives rise to the long-promised Hessian-vector and vector-Hessian-vector products.

7.1 High-Order Forward Directional Derivatives

Given a function $f(\mathbf{x})$ with $\mathbf{x} \in \mathbb{R}^{N_x}$, to compute a univariate derivative with respect to a single x_m one could evaluate the function with a single non-real element $\hat{e}_m$:

$$f(\mathbf{x} + h\mathbf{u^{(m)}}\hat{e}_m) = f(\mathbf{x}) + \left(h\frac{\partial f}{\partial x_m}\right)\hat{e}_m + \left(\frac{h^2}{2}\frac{\partial^2 f}{\partial x_m^2}\right){\hat{e}_m}^2 + \dots \tag{7.1}$$

where $\mathbf{u^{(m)}}$ is a unit vector in the m^{th} direction and the series would be truncated if $\hat{e}_m$ is nilpotent. More generally, one can find a derivative in any direction defined by a vector $\mathbf{v}$ the same way:

$$f(\mathbf{x}+h\mathbf{v}\hat{e}_m) = f(\mathbf{x}) + h\left(\sum_i v_i \frac{\partial f}{\partial x_i}\right)\hat{e}_m + h^2\left(\sum_{i,j}\frac{v_i v_j}{2}\frac{\partial^2 f}{\partial x_i \partial x_j}\right){\hat{e}_m}^2 + \dots \tag{7.2}$$

$$= f(\mathbf{x}) + h\left(\mathbf{v}^T\nabla f\right)\hat{e}_m + \frac{h^2}{2}\left(\mathbf{v}^T\mathbf{H}\mathbf{v}\right){\hat{e}_m}^2 + \mathcal{O}\left(h^3\right){\hat{e}_m}^3 \tag{7.3}$$

Although this expansion has been limited to the second-order (Hessian) terms, higher-order directional tensors[1] can of course still be found – they merely cannot be expressed as vector products of the standard tensors. However, since analysis of second-order derivatives is sufficient to demonstrate hybrid methodology, third-order tensors and higher will be ignored in this chapter. If NMD can add a second derivative order to a first-order method, it can add arbitrarily many more.

To find derivatives in two directions $\mathbf{v}$ and $\mathbf{w}$:

$$\begin{aligned} f(\mathbf{x}+h_m\mathbf{v}\hat{e}_m + h_n\mathbf{w}\hat{e}_n) = f(\mathbf{x}) &+ \sum_i \left(h_m v_i\hat{e}_m + h_n w_i \hat{e}_n\right)\frac{\partial f}{\partial x_i} \\ &+ \sum_{i,j}\frac{1}{2}\left(h_m v_i\hat{e}_m + h_n w_i \hat{e}_n\right)\left(h_m v_j\hat{e}_m + h_n w_j \hat{e}_n\right)\frac{\partial^2 f}{\partial x_i \partial x_j} \\ &+ \dots \end{aligned} \tag{7.4}$$

$$\begin{aligned} = f(\mathbf{x}) &+ h_m\left(\sum_i v_i\frac{\partial f}{\partial x_i}\right)\hat{e}_m + {h_m}^2\left(\sum_{i,j}\frac{v_i v_j}{2}\frac{\partial^2 f}{\partial x_i \partial x_j}\right){\hat{e}_m}^2 \\ &+ h_n\left(\sum_i w_i\frac{\partial f}{\partial x_i}\right)\hat{e}_n + {h_n}^2\left(\sum_{i,j}\frac{w_i w_j}{2}\frac{\partial^2 f}{\partial x_i \partial x_j}\right){\hat{e}_n}^2 \\ &+ h_m h_n\left(\sum_{i,j} v_i w_j\frac{\partial^2 f}{\partial x_i \partial x_j}\right)\hat{e}_m\hat{e}_n + \dots \end{aligned} \tag{7.5}$$

$$\begin{aligned} = f(\mathbf{x}) &+ h_m\left(\mathbf{v}^T\nabla f\right)\hat{e}_m + \frac{{h_m}^2}{2}\left(\mathbf{v}^T\mathbf{H}\mathbf{v}\right){\hat{e}_m}^2 + \mathcal{O}\left({h_m}^3\right){\hat{e}_m}^3 \\ &+ h_n\left(\mathbf{w}^T\nabla f\right)\hat{e}_n + \frac{{h_n}^2}{2}\left(\mathbf{w}^T\mathbf{H}\mathbf{w}\right){\hat{e}_n}^2 + \mathcal{O}\left({h_n}^3\right){\hat{e}_n}^3 \end{aligned} \tag{7.6}$$

[1] The word “tensor” in this context refers to generalizations of vectors and matrices to higher dimensions, not to tensor products of algebras.

$$+ h_m h_n \left(\mathbf{v}^T \mathbf{H} \mathbf{w}\right) \hat{e}_m \hat{e}_n$$

$$+ \mathcal{O}\left(h_m{}^2 h_n\right) \hat{e}_m{}^2 \hat{e}_n + \mathcal{O}\left(h_m h_n{}^2\right) \hat{e}_m \hat{e}_n{}^2$$

where four different error terms have been broken out for later analysis. Note that, in addition to computing a univariate quadratic term in each direction, this approach has found one mixed term $\mathbf{v}^T\mathbf{H}\mathbf{w}$, which corresponds to finding the off-diagonal Hessian value in the standard 2-dimensional formulation.

Recall the example optimization problem used to introduce NMD in Section 1.2. The goal there was to compute a directional Newton update (see Section 2.1.1 or Section 3.1.4) for a line-search:

$$\alpha_k = -\frac{\nabla f_k{}^T \mathbf{d}_i}{\mathbf{d}_i{}^T \mathbf{H}_k \mathbf{d}_i} \tag{7.7}$$

as well as Andrei's [14] Nonlinear Conjugate Gradient update step (see Section 2.1.2):

$$\beta_i = \frac{\mathbf{g}_i{}^T \mathbf{H}_i \mathbf{d}_i - \nabla f_i{}^T \mathbf{d}_i}{\mathbf{d}_i{}^T \mathbf{H}_i \mathbf{d}_i} \tag{7.8}$$

Although the details were omitted in Chapter 1, note that ∇f_k is the gradient as it exists at $\mathbf{x}_k$, which itself is the estimate of the ideal value of $\mathbf{x}$ during iteration k of an inner or outer loop; $\mathbf{H}_i$ is the Hessian as it exists at $\mathbf{x}_i$; and $\mathbf{g}_i = \nabla f_i$ is a full evaluation of the gradient, stored as a vector in memory. The line-search formula for α_k will be evaluated many times for each i – Equation 7.7 is in fact the update to an inner loop used to find an ideal α_i, used in turn to update the search location with $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$; only a single evaluation of β is then needed to update the search direction with $\mathbf{d}_{i+1} = \mathbf{g}_{i+1} + \beta_i \mathbf{d}_i$. As different derivatives are required for α and β, in practice a smaller stencil would be used for the inner loop.

When calculating β in Section 1.4, both $\mathbf{g}^T\mathbf{H}\mathbf{d}$ and $\mathbf{d}^T\mathbf{H}\mathbf{d}$ were found simultaneously with a single NMD stencil. The hypercomplex algebra corresponding to that stencil can be derived by analyzing Equation 7.6: if $\mathbf{v} = \mathbf{d}$ and $\mathbf{w} = \mathbf{g}$, one can remove all undesired terms by setting:

$$\hat{e}_m = \varepsilon_1 \neq 0 \tag{7.9}$$

$$\hat{e}_n = \varepsilon_2 \neq 0 \tag{7.10}$$

$$\varepsilon_1{}^3 = 0 \neq \varepsilon_1^2 \tag{7.11}$$

$$\varepsilon_2{}^2 = 0 \tag{7.12}$$

$$\varepsilon_1 \varepsilon_2 = \varepsilon_{12} \neq 0 \tag{7.13}$$

$$\varepsilon_1{}^2 \varepsilon_2 = \varepsilon_{112} = 0 \tag{7.14}$$

Counting the real element, this algebra has five basis elements and hence requires a 5×5 stencil, as was shown in Equation 1.10:

$$\mathcal{DM}(\,\cdot\,) = \begin{bmatrix} 1 & 0 & {}^{\partial}/_{\partial r_1} & {}^{\partial}/_{\partial r_2} & {}^{\partial^2}/_{\partial r_1 \partial r_2} \\ 0 & 1 & 0 & 2\left({}^{\partial}/_{\partial r_1}\right) & {}^{\partial}/_{\partial r_1{}^2} \\ 0 & 0 & 1 & 0 & {}^{\partial}/_{\partial r_2} \\ 0 & 0 & 0 & 1 & {}^{\partial}/_{\partial r_1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.15}$$

Recall that this stencil originally corresponded to the directional equation $f(\mathbf{x} + r_1\mathbf{d} + r_2\mathbf{g})\,|_{r_1=r_2=0}$, which is simply the real-valued equivalent of Equation 7.4; any AD tool would be initialized as if r_1 and r_2 were the program inputs.

7.2 A Nilpotent-Complex Hybrid

Now, however, consider a different hypercomplex algebra using one nilpotent element $\hat{e}_m = \varepsilon_1$ of degree three and a single imaginary element $\hat{e}_n = i$. Assume h_m is non-negligible, but $h_n \ll 1$ such that $h_n{}^2 \approx 0$. Then Equation 7.6 becomes:

$$f(\mathbf{x} + h_m \mathbf{v}\varepsilon_1 + h_n \mathbf{w} i) = f(\mathbf{x}) + h_m \left(\mathbf{v}^T \nabla f\right) \varepsilon_1 + h_n \left(\mathbf{w}^T \nabla f\right) i \tag{7.16}$$

$$+\frac{{h_m}^2}{2}\left(\mathbf{v}^T\mathbf{H}\mathbf{v}\right)\varepsilon_1{}^2 + h_n h_m\left(\mathbf{v}^T\mathbf{H}\mathbf{w}\right)i\varepsilon_1$$
$$+\mathcal{O}\left({h_n}^2\right)+\mathcal{O}\left({h_n}^2 h_m\right)\varepsilon_1+\mathcal{O}\left(h_n{h_m}^2\right)i\varepsilon_1{}^2$$

This also computes the desired quantities, although (after accounting for the fact that $\varepsilon_1{}^3$ is precisely zero) there are still three remaining error terms. The impact of these errors can best be understood by isolating individual output terms with the operators Re$(\cdot)$, Im$(\cdot)$, Nil$_1(\cdot)$, and Nil$_{11}(\cdot)$:

$$\operatorname{Re}\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right) = f(\mathbf{x})+\mathcal{O}\left({h_n}^2\right) \tag{7.17}$$
$$\operatorname{Im}\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right) = h_n\left(\mathbf{w}^T\nabla f\right) \tag{7.18}$$
$$\operatorname{Re}\left(\operatorname{Nil}_1\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right)\right) = h_m\left(\mathbf{v}^T\nabla f\right)+\mathcal{O}\left({h_n}^2\right) \tag{7.19}$$
$$\operatorname{Im}\left(\operatorname{Nil}_1\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right)\right) = h_n h_m\left(\mathbf{w}^T\mathbf{H}\mathbf{v}\right) \tag{7.20}$$
$$\operatorname{Re}\left(\operatorname{Nil}_{11}\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right)\right) = \frac{{h_m}^2}{2}\left(\mathbf{v}^T\mathbf{H}\mathbf{v}\right) \tag{7.21}$$
$$\operatorname{Im}\left(\operatorname{Nil}_{11}\left(f(\mathbf{x}+h_m\mathbf{v}\varepsilon_1+h_n\mathbf{w}i)\right)\right) = \mathcal{O}\left(h_n{h_m}^2\right) \tag{7.22}$$

The nilpotent and imaginary terms together create a hypercomplex algebra defined by not five but six different dimensions. Three of these dimensions can exactly compute derivative terms, including both of the desired second derivatives, and two of them (just as with ordinary CD) are effectively exact due to the very small size of $\mathcal{O}\left({h_n}^2\right)$. The only significant error[2] is pushed into the sixth dimension – i.e., the imaginary part of the square of the prime nilpotent – where it can safely be ignored. A matrix representation of this algebra can recover derivatives with the 3×3 tensor (defined by using Algorithm 6.1 and adding an imaginary component):

$$\mathcal{DM}'(\,\cdot\,) = \begin{bmatrix} 1+hi\,{}^{\partial}/_{\partial r_1} & 2\,{}^{\partial}/_{\partial r_2}+2hi\,{}^{\partial^2}/_{\partial r_1\partial r_2} & {}^{\partial^2}/_{\partial {r_2}^2} \\ 0 & 1+hi\,{}^{\partial}/_{\partial r_1} & {}^{\partial}/_{\partial r_2}+hi\,{}^{\partial^2}/_{\partial r_1\partial r_2} \\ 0 & 0 & 1+hi\,{}^{\partial}/_{\partial r_1} \end{bmatrix} \tag{7.23}$$

[2] In point of fact, the “error” is actually a single third-order mixed derivative, but as this was not requested it is considered superfluous.

where $h = h_n$ can be divided out to recover derivatives from the imaginary component. Despite the extra imaginary term, if the cost of increasing a matrix's size is greater than the switch to complex math, this stencil could be significantly faster than that given by Equation 7.15.

In practice, a problem of this form would be computed by running the user program with $r_1 = 0 + hi$ and $r_2 = 0 + \varepsilon_1$, with ε_1 being the first derivative channel of an operator-overloaded AD type configured to track one first and one second univariate derivative. At the hand-off from scalar to matrix math, every overloaded variable $\hat{z}$ would be carrying three complex values: $z_0 + hi^{\partial z}/_{\partial r_1}$, $^{\partial z}/_{\partial r_2} + hi^{\partial^2 z}/_{\partial r_1 \partial r_2}$, and $^{\partial z^2}/_{\partial r_2{}^2} + hiz_{err}$; these would be inserted by a standard NMD function (that is, the same function that would otherwise handle purely real-valued NMD) into the appropriate stencil location(s) in an expanded matrix or vector.

If two separate NMD-enabled runs of the program are used, still-simpler stencils can be used. Consider a standard 2×2 matrix representing a dual number, now augmented with imaginary values; omitting the hypercomplex derivation, this "dual-complex" number can be used as a derivative stencil:

$$\mathcal{DM}''(\,\cdot\,) = \begin{bmatrix} 1 + hi^{\partial}/_{\partial r_1} & ^{\partial}/_{\partial r_2} + hi^{\partial^2}/_{\partial r_1 \partial r_2} \\ 0 & 1 + hi^{\partial}/_{\partial r_1} \end{bmatrix} \tag{7.24}$$

In this case, the overloaded AD type[3] need only track a single first derivative. Before the first run of the program, the directional inputs would be initialized just as above to compute the mixed second derivative; before the second run, they would be initialized with $r_1 = 0$, $r_2 = 0 + hi + \varepsilon_1$ to compute the univariate second derivative.

Before moving on, note that the Newton line-search step $\alpha = {}^{-\mathbf{d}^T \nabla f}/_{\mathbf{d}^T \mathbf{H} \mathbf{d}}$ – a very useful quantity with broad applicability to many other optimization frameworks besides NCG – can be found using just the terms calculated by a single forward pass of Equation 7.24. Further note that the cost of this stencil (like any other) is independent of N_x, meaning that gradient-vector and vector-Hessian-vector products can be found in constant time. The full Hessian is

[3] If a C++ code has already been modified to use the CD method via Martins' [63] "cplx" variable type, it could be straightforwardly converted to use a dual-complex type, as cplx is simply a wrapper for C++'s standard complex<double>; this in turn is built using the language's templating ability, and can thus be redefined as complex<T> for any scalar type T. Redefining cplx as a wrapper for complex<ADtype> would be trivial.

not required, nor is the full gradient[4] – and therefore, neither is an adjoint method.

As a final aside, observe that the terms calculated above can be used to form a quadratic interpolant, i.e., the Taylor approximation:

$$f(\mathbf{x} + t\mathbf{v}) \approx \hat{f}(t) = f(\mathbf{x}) + t\mathbf{v}^T \nabla f + \frac{t^2}{2}\mathbf{v}^T \mathbf{H}\mathbf{v} + \mathcal{O}\left(t^3\right) \tag{7.25}$$

where multiple values of t can be tested[5] after a single run of NMD (or, for scalar codes, a single run of second-order vanilla AD). This single-variable equation is one of the simplest types of surrogate model discussed in Section 2.3; a two-variable interpolant (i.e., a quadratic approximation of f within the space spanned by $\mathbf{w}$ and $\mathbf{v}$) could be built by computing every term from Equation 7.6, and higher-dimensional interpolants could be created with the addition of yet more non-real offset direction vectors. The availability of such Taylor approximations is perhaps unsurprising, as the derivative terms are all derived from a Taylor series. More interesting is the fact that, as a forward method, the terms are also found for all intermediate values of a user program. In cases where $f(\mathbf{x})$ actually represents a quantity of interest $f(\mathbf{u}(\mathbf{x}))$ extracted from a larger state-vector $\mathbf{u}$ – e.g., a PDE solved over a million-variable mesh – then this solution state, too, is imbued by NMD with the information needed to build a quadratic estimate of $\mathbf{u}(\mathbf{x} + t\mathbf{v})$ for any t:

$$u_j(\mathbf{x} + t\mathbf{v}) \approx \hat{u_j}(t) = u_j(\mathbf{x}) + t\,\nabla {u_j}^T \mathbf{v} + \frac{t^2}{2}\mathbf{v}^T \left(\mathbf{H}_{\mathbf{x}}\left[u_j(\mathbf{x})\right]\right)\mathbf{v} + \mathcal{O}\left(t^3\right) \tag{7.26}$$

For the full vector $\mathbf{u}$, the gradient term $\nabla {u_j}^T \mathbf{v}$ instead uses a Jacobian matrix $\mathbf{J}\left[\mathbf{u}(\mathbf{x})\right]\mathbf{v}$, and the Hessian $\mathbf{H}\left[\mathbf{u}(\mathbf{x})\right]$ is in fact a rank-three tensor, meaning the vector-Hessian-vector product yields a vector – but this product is still implicit (and still found with a cost independent of N_u); no complicated tensor operations need to be considered. Again, this surrogate model of the state-vector can be generalized to any number of direction vectors or any order of derivative.

[4] Unfortunately, Andrei's NCG update step β still requires a previous evaluation of the full gradient $\mathbf{g} = \nabla f$ to find the $\mathbf{g}^T\mathbf{H}\mathbf{d}$ term. NMD (or any other AD method) can find the products of derivatives with vectors, but not the products of derivatives with other derivatives. Even for NCG, though, note that NMD may still be a superior option in the inner (line-search) loop, which uses only the Newton step.

[5] Note that the Newton update is a specific case of this – α is the solution of ${}^{\partial \hat{f}}/_{\partial t} = 0$.

7.3 Nilpotent and Complex Adjoint Methods

Throughout Chapter 2, multiple applications were presented that made use of the product of the Hessian with a vector. Section 3.1.4 introduced Pearlmutter's [19] differentiation operator, the intrusive application of which could turn a gradient algorithm into one for **Hv**:

$$\mathcal{R}_{\mathbf{v}}\{f(\mathbf{x})\} \equiv \frac{\partial}{\partial r}\Big(\nabla f(\mathbf{x}_i + r\mathbf{v})\Big)\bigg|_{r=0} = \mathbf{Hv} \tag{7.27}$$

In pursuit of an automatic alternative for computing this quantity, begin again with a generic hypercomplex evaluation, this time of ∇f with a non-real step in a cardinal direction $\mathbf{u}^{(\mathbf{n})}$, and examine the result element by element with a Taylor expansion:

$$\left[\nabla f(\mathbf{x} + h\mathbf{u}^{(\mathbf{n})}\hat{e}_n)\right]_j = \frac{\partial f}{\partial x_j} + h\frac{\partial^2 f}{\partial x_j \partial x_n}\hat{e}_n + \frac{h^2}{2}\frac{\partial^3 f}{\partial x_j \partial {x_n}^2}{\hat{e}_n}^2 + \dots \tag{7.28}$$

As expected, this approach simply differentiates the gradient with respect to x_n. Already of note, though, is the fact that the first $\hat{e}_n$ term contains a single entry of the Hessian. When examining all j, it can be seen that this formulation will compute an entire row or column of the matrix.

With an arbitrary direction vector **v**, the expansion becomes:

$$\left[\nabla f(\mathbf{x} + h\mathbf{v}\hat{e}_n)\right]_j = \frac{\partial f}{\partial x_j} + h\left(\sum_{k=1}^{N_x} v_k \frac{\partial^2 f}{\partial x_j \partial x_k}\right)\hat{e}_n + \frac{h^2}{2}\left(\sum_{k=1}^{N_x}\sum_{l=1}^{N_x} v_k v_l \frac{\partial^3 f}{\partial x_j \partial x_k \partial x_l}\right){\hat{e}_n}^2 + \dots \tag{7.29}$$

$$= \left[\nabla f(\mathbf{x})\right]_j + h\left[\mathbf{H}_{(j,\cdot)}\right]\mathbf{v}\hat{e}_n + \mathcal{O}\left(h^2\right){\hat{e}_n}^2 \tag{7.30}$$

In other words, the first $\hat{e}_n$ term now contains the inner product of a row of the Hessian with **v**; when examining all j, this constitutes the complete Hessian-vector product:

$$\nabla f(\mathbf{x} + h\mathbf{v}\hat{e}_n) = \nabla f(\mathbf{x}) + h\mathbf{Hv}\hat{e}_n + \mathcal{O}\left(h^2\right){\hat{e}_n}^2 \tag{7.31}$$

It can now be see that, given any algorithm that evaluates the gradient, a great many possible hypercomplex algebras could compute $\mathbf{Hv}$. Most importantly, this includes algebras utilizing nilpotent elements – i.e., those isomorphic to NMD stencils – and complex numbers when $h \ll 1$.

Complex numbers have been suggested in this context before, at least in terms of generic functions that produce the gradient. Such a scheme to compute $\mathbf{Hv}$ was mentioned in passing (somewhat misleadingly described as a "finite difference approximation") in Nash's 2000 survey [28] of Truncated Newton methods (see Section 2.1.3), which predated Martins' 2003 [63] introduction of the complex-step method to the broader aerospace engineering community. However, Nash did not appear to realize the power of the approach – e.g., he incorrectly described it as a $\mathcal{O}(h)$ method and wrote as if it did not have the same accuracy as traditional AD.

A more specific usage was reported by Bogdan Constantin in his 2014 M.S. thesis at MIT [118], where he applied the complex-step method as it exists today to an atmospheric chemistry-transport model. The code he considered, GEOS-Chem, already featured a sophisticated sensitivity-analysis (SA – see Section 2.2) adjoint procedure [119], constructed via a combination of scalar automatic differentiation, the discrete adjoint method described in Section 3.1.2, and the continuous adjoint approach (based on the physics of the underlying physical systems, not just their code). Although he successfully used CD to find second-order sensitivities (i.e., sensitivities of the adjoint sensitivities), his work was kept in a pure SA framework and did not make the connection to other uses of differentiation – indeed, the words "gradient" and "Hessian" were never used.

However, more general uses of $\mathbf{Hv}$ are possible, and – via Equation 7.31 – can be found via modification of any existing gradient algorithm. When the manual formula of Equation 7.27 is put to use in the very specific field of neural network deep learning (e.g., [16,41]), that gradient is found by the backpropagation algorithm, in which case scalar AD could be used in place of hand-differentiation. Engineering applications, in contrast, more commonly find gradients via the discrete adjoint method, which uses linear algebra – avoiding manual differentiation here requires either CD or NMD.

One version of the discrete adjoint method was described in detail in Section 3.1.2. Once again, very briefly, given a function f of inputs $\mathbf{x}$, solution state-vector $\mathbf{u}(\mathbf{x})$, and residual $\mathbf{r}(\mathbf{u},\mathbf{x})$, the method amounts to:

$$\left.\begin{aligned} \frac{\partial \mathbf{r}}{\partial \mathbf{u}}^T \boldsymbol{\lambda} &= -\frac{\partial f}{\partial \mathbf{u}}^T \\ \nabla f &= \boldsymbol{\lambda}^T \frac{\partial \mathbf{r}}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{x}} \end{aligned} \;\right|\; \begin{aligned} &\mathbf{x} \in \mathbb{R}^{N_x} \\ &\mathbf{u} \in \mathbb{R}^{N_u} \\ &\mathbf{r}(\mathbf{u}, \mathbf{x}) = \mathbf{0} \in \mathbb{R}^{N_u} \\ &f(\mathbf{u}, \mathbf{x}) \in \mathbb{R} \end{aligned} \tag{7.32}$$

When $\mathbf{u}$ is found via the solution of some system $\mathbf{A}(\mathbf{x})\mathbf{u} = \mathbf{b}(\mathbf{x})$, the residual is just $\mathbf{r} = \mathbf{A}(\mathbf{x})\mathbf{u} - \mathbf{b}(\mathbf{x})$, meaning that $\partial\mathbf{r}/\partial\mathbf{u} = \mathbf{A}(\mathbf{x})$ is already available; however, the other derivatives $\partial\mathbf{r}/\partial\mathbf{x}$, $\partial f/\partial\mathbf{u}$ and $\partial f/\partial\mathbf{x}$ must be manually implemented by the user. Each quantity, though, is expressed as a matrix or vector, and thus can be made to carry *additional* derivatives via NMD. In the case of $\mathbf{A}$, the derivative-augmented matrix $\check{\mathbf{A}}$ would be built by the forward run of the program; in the other cases, presuming the derivative matrices aren't themselves constructed via matrix operations (in which case NMD would remain the method of choice), first-order scalar AD could be employed during their construction, yielding derivatives-of-derivatives to arrange in stencils. Just as with general (forward-mode) NMD, a blockwise variant of the transpose operation in Equation 7.32 would be necessary to preserve the hypercomplex isomorphism – but again, the cost of this is trivial compared to the cost of the solve.

Alternatively, if the program inputs and all successive operations are real-valued, CD could be used as long as care is taken to use only complex-analytic functions; this includes both the scalar difficulties addressed by Martins [63] in Section 4.1, and exclusive use of a real-valued transpose in Equation 7.32 – a complex-conjugate transpose, which is not analytic, would ruin differentiation.

Chapter 2 presented several powerful applications of $\mathbf{Hv}$, but at least two more possibilities should be pointed out. First and most obviously, the expansion from Equation 7.31 can be used to build a surrogate model for the gradient, similar to Equation 7.25:

$$\nabla f(\mathbf{x} + t\mathbf{v}) \approx \hat{g}(t) = \nabla f + t\mathbf{Hv} + \mathcal{O}\left(t^2\right) \tag{7.33}$$

Note here that ∇f would be found alongside $\mathbf{Hv}$, as it constitutes the real part of the NMD or CD adjoint. Just as in Equation 7.25, many values of t can be evaluated after a single evaluation of the modified adjoint.[6] This is a linear extrapolation, and must therefore be held comparatively near $\mathbf{x}$; however, as a second possibility, note that $\mathbf{Hv}$ and ∇f could likewise be used to build a surrogate model for the actual output that is broader than Equation 7.25:

$$\begin{aligned} f(\mathbf{x} + t\mathbf{v} + \mathbf{w}) \approx\ & \hat{f}(t, \mathbf{w}) && (7.34) \\ =\ & f(\mathbf{x}) + t\mathbf{v}^T \nabla f + \mathbf{w}^T \nabla f && (7.35) \\ & + \frac{t^2}{2}\mathbf{v}^T \mathbf{H}\mathbf{v} + \frac{t}{2}\mathbf{w}^T \mathbf{H}\mathbf{v} && (7.36) \\ & + \mathcal{O}\big(||\mathbf{w}||^2\big) + \mathcal{O}\big(t^3\big) && (7.37) \end{aligned}$$

where many values of both t and $\mathbf{w}$ could be provided after a single hybrid adjoint evaluation. Due to the poorer order of accuracy with respect to the magnitude of the direction vector $||\mathbf{w}||$, this vector must be held relatively short, which limits the surrogate to points somewhat near the line $\mathbf{x} + t\mathbf{v}$. This effectively makes Equation 7.34 a method for perturbing Equation 7.25 in any direction in the search space, rather than simply along the vector $\mathbf{v}$. At the perturbed points, the surrogate would be imbued with knowledge of how ∇f curves toward $\mathbf{v}$. Again, finding multiple products $(\mathbf{Hv}^{(1)}, \mathbf{Hv}^{(2)}, \dots)$ would expand the surrogate from points near a line to points near some subspace.

Practical computation of both an NMD-adjoint hybrid and a CD-adjoint hybrid will be made clearer with a demonstration in Chapter 8. Before moving on, though, compare these hypercomplex adjoint approaches to the second-order discrete adjoint method reviewed in Section 3.1.3. As stated there, that method could find the full Hessian with cost-complexity $\mathcal{O}\big(N_x\big)$, which was superior to forward-mode methods with necessary complexity $\mathcal{O}\big({N_x}^2\big)$. In the hypercomplex case, Equation 7.28 can be used to find a single column in $\mathcal{O}(1)$ – and, thus, the entire matrix can also be found in $\mathcal{O}\big(N_x\big)$. The second-order method, however, could not cost-effectively find subsets of $\mathbf{H}$ – finding even a single scalar entry in isolation required (at minimum) three solves of the system, and finding an entire column (let alone $\mathbf{Hv}$ for arbitrary $\mathbf{v}$) required the same $N_x + 1$ solves as would be required to find the full matrix. This comes despite

[6] Pearlmutter mentioned something very similar, but only in the context of deriving the Newton update step and assessing the accuracy thereof by comparing a predicted gradient to the true gradient at the new Newton location.

the fact that the hypercomplex variants (especially the CD hybrid) are far easier to implement. Even more strikingly, arbitrary-order derivatives – third, fourth, or higher – can be found relatively easily with a nilpotent-adjoint hybrid (or even a nilpotent-complex-adjoint hybrid) with a cost lower than forward-mode AD by a factor of N_x (that is, $\mathcal{O}\left(N_x{}^2\right)$ for third-order tensors, $\mathcal{O}\left(N_x{}^3\right)$ for fourth-order, etc). All of this can be done without any new derivations on the part of the user.

Part III

Demonstrating Nilpotent Matrix Differentiation

Chapter 8

Differentiating a Stochastic Diffusion Model

To demonstrate its use for actual problem-solving, all variants of Nilpotent Matrix Differentiation were used to study a system defined by a second-order elliptic partial differential equation. Specifically, the original "user" program was designed to find the equilibrium concentration of a two-dimensional diffusive process with a spatially-varying stochastic diffusivity field, where the field itself was in turn was drawn from a lognormal random process approximated by a Karhunen-Loève expansion (KLE, [120]). This problem was chosen as an NMD benchmark for four reasons: First, codes that solve such elliptic PDEs should be familiar to most engineers. Second, this same problem has already been studied in the context of system derivatives for uncertainty quantification [47]. Third, it strikes a good balance between complex behavior and ease of implementation, which was important when testing early experimental variants of NMD or competing methods like CD. Fourth and finally, the KLE approximation uses an arbitrary number of basis functions (growing more accurate with each), each of which must be weighted by a coefficient; thus, the problem can take an arbitrary number of inputs (i.e., the coefficients), each of which have similar quantitative impact and qualitative meaning. This allows for an "apples-to-apples" comparison of NMD as applied to problems with different numbers of independent differentiation variables.

This chapter is organized as follows: First, in Section 8.1, each step of the core problem is described in precise detail. Second, in Section 8.2, experiments are described to measure accuracy and cost of several possible applications of the NMD method. Results are then presented in Section 8.3 and Section 8.4.

8.1 Detailed Problem Description

The two-dimensional diffusion problem is modeled as:

$$\begin{aligned}\frac{\partial u}{\partial t} &= \nabla \cdot \left(k \nabla u\right) \\ &= \nabla \cdot \left(k \left(\frac{\partial u}{\partial x_1} + \frac{\partial u}{\partial x_2}\right)\right) \\ &= \left(\frac{\partial k}{\partial x_1}\frac{\partial u}{\partial x_1} + \frac{\partial k}{\partial x_2}\frac{\partial u}{\partial x_2}\right) + k\left(\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial {x_2}^2}\right)\end{aligned} \tag{8.1}$$

where many boundary conditions are possible. This work uses a stochastic variant of the diffusivity problem, which takes as input a vector $\mathbf{y} \in \mathbb{R}^{N_y}$ that contains samples of independent and identically distributed (i.i.d.) random variables and ultimately outputs a quantity of interest (QoI) $f(\mathbf{y})$ of an equilibrium solution. Before any AD tool is applied, computation of the original problem proceeds as follows: First, it calculates a deterministic solution for a spatially uniform diffusivity parameter, which is then used to define boundary conditions (BCs) for the stochastic problem. Second, it uses Galerkin projection to approximate the eigenfunctions of a covariance kernel; together with the coefficients $\mathbf{y}$, this is sufficient to sample a lognormal diffusivity field. Next, the diffusivity is used in a discretization of the diffusion equation, resulting in a linear system that is solved to find an equilibrium concentration field. Finally, the quantity of interest is the mean of the squared difference between the stochastic and deterministic solutions. For convenience, the relevant variables and functions are listed here:[1]

N_y	Number of input (independent) variables
$N_u = N_{u1}^2$	Number of mesh nodes (total and per direction)
$N_d = N_{d1}^2 = (N_{u1} - 2)^2$	Number of degrees of freedom (state variables)
$N_b = N_{b1}^2$	Number of finite elements for eigenfunction approximation
$\mathbf{x} = [x_1, x_2]^T \in [0,1]^2$	Geometric coordinate vector (the spatial domain)

[1] Notation throughout this subsection adapted largely from [121] and [47].

$t \in [0, \infty]$	Time (the temporal domain)
$\omega \in \Omega$	Random variable (the probabilistic domain)
$\mathbf{y}(\omega) \in \mathbb{R}^{N_y}$	Input vector: standard-normal (i.i.d.) random samples
$g_i(\mathbf{x}_i, \omega) \mid i \in \{1, \dots, N_u\}$	Gaussian random process; value at discretized location $\mathbf{x}_i$
$\hat{g}_i(\mathbf{x}_i, \mathbf{y})$	Truncated Karhunen-Loève expansion (KLE) of g
$C_{gg}(\mathbf{x}_1, \mathbf{x}_2)$	Squared exponential covariance function of g
$\phi_{(i,j)}(\mathbf{x}_i) \mid j \in \{1, \dots, N_y\}$	Eigenfunctions of C_{gg}; value of function j at location i
λ_j	Eigenvalues of C_{gg}
$\psi_n(\mathbf{x}) \mid n \in \{1, \dots, N_b\}$	Bilinear finite element basis for approximation of $\phi(\mathbf{x})$
$k_i(\mathbf{x}_i, \mathbf{y})$	Lognormal diffusivity coefficient
$u_i(\mathbf{x}, t, \mathbf{k})$	Concentration of a species (e.g., thermal energy or a solute)
$\mathbf{g}, \hat{\mathbf{g}}, \boldsymbol{\phi}, \mathbf{k} \in \mathbb{R}^{N_u}$	Field vectors: values at all mesh points
$\mathbf{u} \in \mathbb{R}^{N_d}$	State vector: values at all degrees of freedom
$\mathbf{u}_{eq}(\mathbf{k}) \in \mathbb{R}^{N_d}$	(Stochastic) equilibrium solution vector
$\mathbf{u}_{heat} \in \mathbb{R}^{N_d}$	(Deterministic) equilibrium solution vector for uniform k
$f(\mathbf{u}_{eq}) = f(\mathbf{y})$	Output variable (quantity of interest)

The goal of computation is to solve for system equilibrium at $\frac{\partial u}{\partial t} = 0$, thus making Equation 8.1 an elliptic PDE. The main difficulty in the solution stems from the fact that the diffusivity coefficient k is unevenly distributed across the physical domain, in a way that differs from simulation to simulation; this distribution is modeled by a lognormal random process, calculated as the exponential of the KLE of a Gaussian process g. Denoting the KLE approximation as $\hat{g}$ and indexing into node i of a discretized mesh, k can then be defined as:

$$\hat{g}_i(\mathbf{x}_i, \mathbf{y}) = g_0 + \sum_{j=1}^{N_y} y_j \sqrt{\lambda_j} \phi_{(i,j)}(\mathbf{x}_i) \tag{8.2}$$

$$k_i(\mathbf{x}_i, \mathbf{y}) = k_0 + \exp(\hat{g}_i(\mathbf{x}_i, \mathbf{y})) \tag{8.3}$$

Here, y_j are independent, identically-distributed (i.i.d.) standard normal random variables; these will be considered the independent variables (i.e., inputs) during the system differentiation benchmarks, as they are the only variables that change between different stochastic runs of the program. Next, ϕ_j and λ_j are the eigenfunctions and eigenvalues of the squared exponential covariance function:

$$C_{gg}(\mathbf{x}_1, \mathbf{x}_2) = \sigma^2 e^{-(\|\mathbf{x}_1 - \mathbf{x}_2\|/l_c)^2} \tag{8.4}$$

For the experiments described in this chapter (and somewhat following the similar experiment in [47]), the parameters are set to $\sigma = 0.5$, $l_c = 0.01$, $g_0 = 0.1$, and $k_0 = 0$. Calculation of the above quantities and their use in the KLE is explained in Section 8.1.2 and Section 8.1.3. First, though, it is instructive to consider the case of a deterministic and uniform k, which leads to a helpful choice for the problem's boundary conditions (BCs).

8.1.1 A Deterministic Known Solution: The Heat Equation

Observe that if k is constant in space, the problem reduces to the familiar (deterministic) heat equation $\frac{\partial u}{\partial t} = 0 = \nabla^2 u$ which, for certain boundary conditions, has an analytic equilibrium solution u_{heat}. One particularly simple way to find such a solution is to define a function that is linear in both directions, such that both second derivatives are zero. For example:

$$\begin{aligned} u_{heat}(\mathbf{x}) &= \left(m_1 x_1 + b_1\right)\left(m_2 x_2 + b_2\right) \\ &= m_1 m_2 x_1 x_2 + m_1 b_2 x_1 + m_2 b_1 x_2 + b_1 b_2 \end{aligned} \tag{8.5}$$

If the boundaries are held constant at values that match this function, the interior of the domain will converge to the same function. Note that Equation 8.5 has the analytic derivatives

$$
\begin{aligned}
\frac{\partial u_{heat}}{\partial m_1} &= m_2 x_1 x_2 + b_2 x_1 \\
\frac{\partial u_{heat}}{\partial m_2} &= m_1 x_1 x_2 + b_1 x_2 \\
\frac{\partial u_{heat}}{\partial b_1} &= m_2 x_2 + b_2 \\
\frac{\partial u_{heat}}{\partial b_2} &= m_1 x_1 + b_1 \\
\frac{\partial^2 u_{heat}}{\partial m_1 \partial m_2} &= x_1 x_2 \\
\frac{\partial^2 u_{heat}}{\partial m_1 \partial b_2} &= x_1 \\
\frac{\partial^2 u_{heat}}{\partial m_2 \partial b_1} &= x_2 \\
\frac{\partial^2 u_{heat}}{\partial b_1 \partial b_2} &= 1
\end{aligned}
$$

In this work, $m_1 = b_1 = 500$ and $m_2 = b_2 = 1000$. Defining the original program's Dirichlet BCs to be equal to Equation 8.5 thus provides a straightforward way to verify both the original program and any derivative method applied to it. However, when moving to the stochastic problem, $k(\mathbf{x}, \mathbf{y})$ becomes a field that varies across $\mathbf{x}$; typically, analytic solutions are no longer available – a discretized, purely numerical approximation is required. One of the most straightforward approaches to studying the discretized solution vector $\mathbf{u}_{eq}$ and any function $f(\mathbf{u}_{eq})$ is to simply draw many samples of $\mathbf{k}$ and repeatedly solve the system. However, the sampling process itself is a nontrivial task. Before entering the differentiatable section of code, it is necessary to build a Karhunen-Loève expansion of the process, which in turns requires calculation of eigenfunctions of the chosen covariance kernel. These steps are described in Section 8.1.2. This expansion can then be used to generate samples of the process, as described in Section 8.1.3. Only after an instance of $\mathbf{k}$ has been realized can Equation 8.1 be discretized and solved, as described in Section 8.1.4.

8.1.2 Galerkin Projection and Karhunen-Loève Expansion

Any given random process $p(\mathbf{x}, \omega)$ can expressed with a generalized Fourier expansion (GFE) – that is, a decomposition into infinitely many weighted basis functions:

$$p(\mathbf{x}, \omega) = \sum_{0}^{\infty} y_j(\omega)\phi_j(\mathbf{x}) \tag{8.6}$$

Any Fourier expansion is identically equal to the process itself, in the limit of infinitely many basis functions; however, any numeric approximation must truncate the expansion to a finite number of terms N_y, thereby throwing away a potentially significant amount of information. The best approach to this problem is generally considered to be the Karhunen-Loève expansion (KLE), which has been shown to minimize total mean squared error for a given truncation; it is defined as the GFE whose basis functions are the eigenfunctions of the process's covariance function $C_{pp}(\mathbf{x}_1, \mathbf{x}_2)$. That is, each eigenfunction ϕ_j is a solution to the operator equation

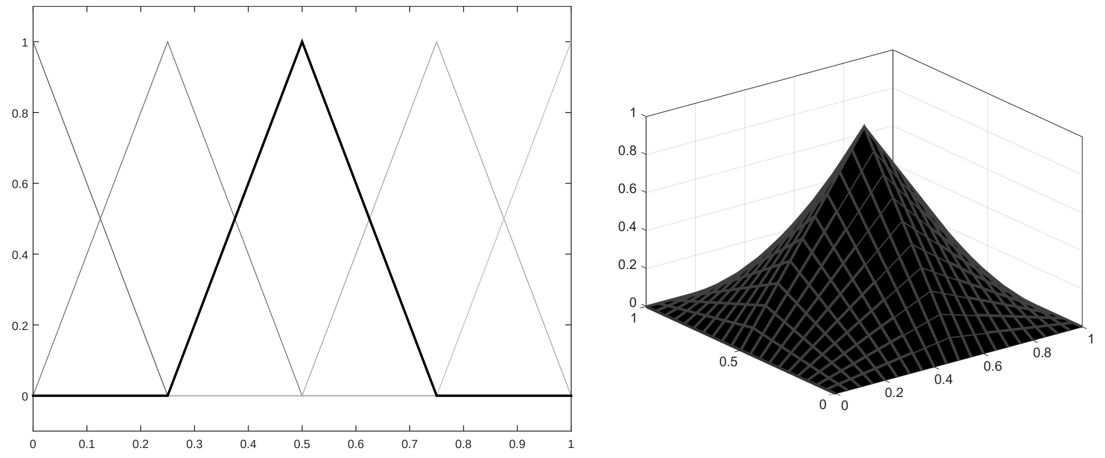
$$\int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)\phi_j(\mathbf{x}_2)d\mathbf{x}_2 = \lambda_j\phi_j(\mathbf{x}_1) \tag{8.7}$$

In plain English, $C_{pp}(\mathbf{x}_1, \mathbf{x}_2)$ determines how much a field measured at $\mathbf{x}_2$ will change if the field is perturbed at $\mathbf{x}_1$. If the function $\phi_j(\mathbf{x})$ is used to perturb an entire field over all $\mathbf{x}$, the net impact on any given point $\mathbf{x}_1$ is thus determined by the integral of the covariance function over the whole spatial domain. If that impact is simply a linear scaling of $\phi_j(\mathbf{x}_1)$ at that point in isolation, then $\phi_j(\mathbf{x})$ is an eigenfunction of C_{pp} and the eigenvalue λ_j measures the strength of the relationship. Because the relationship also works in reverse, it can be seen that the eigenfunctions with the largest λ_j have the greatest impact on the field as a whole. This gives a natural way to approximate the process defined by C_{pp}: stack the most important eigenfunctions on top of each other, until their collective behavior captures that of of the true process.

Unfortunately, solutions of Equation 8.7 are rarely available analytically – they must be found via yet another approximation. In this case, the approximate eigenfunctions are found via the method of Galerkin projection, which is used to convert the continuous operator problem into one of solving for another, separate set of weighted basis functions:

$$\phi_j(\mathbf{x}) \approx \hat{\phi}_j(\mathbf{x}) = \sum_{n=1}^{N_b} z_{j,n}\psi_n(\mathbf{x}) \tag{8.8}$$

In this case, the functions $\psi_n(\mathbf{x})$ are chosen by the user; for this problem, they are simply (bi)linear finite elements, as shown in Figure 8.1.Note that bilinear elements are the product of two one-dimensional linear elements, with one running along each cardinal direction, much as the boundary condition solution chosen in Equation 8.5 is the product of two linear solutions.



(a) (b)

Figure 8.1: (a): A sequence of five linear finite elements, with the central element highlighted. The value of each $\psi_n(\mathbf{x})$ is linear inside the element and zero everywhere else. (b): The central element of a 3×3 grid of bilinear elements. This element is linear along cardinal directions but otherwise quadratic. Note that in both (a) and (b), the sum of all elements is everywhere equal to 1, so that summing across all weighted elements yields a linear interpolation of $\phi(\mathbf{x})$ between adjacent known values. These are very small grids for demonstration, but in real applications hundreds of elements (or more) are used in each direction.

The Galerkin method consists of, first, approximating the residual r of Equation 8.7 by substituting in Equation 8.8:

$$
\begin{aligned}
r(\mathbf{x}_1) &\equiv \int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)\phi_j(\mathbf{x}_2)d\mathbf{x}_2 - \lambda_j \phi_j(\mathbf{x}_1) && (8.9)\\
&\approx \int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)\sum_{n=1}^{N_b} z_{j,n}\psi_n(\mathbf{x}_2)d\mathbf{x}_2 - \lambda_j \sum_{n=1}^{N_b} z_{j,n}\psi_n(\mathbf{x}_1) && (8.10)\\
&= \sum_{n=1}^{N_b}\left(\int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)z_{j,n}\psi_n(\mathbf{x}_2)d\mathbf{x}_2 - \lambda_j z_{j,n}\psi_n(\mathbf{x}_1)\right) && (8.11)
\end{aligned}
$$

The method then sets the residual to be orthogonal to each $\psi_m(\mathbf{x})$ – that is, solves for a set of coefficients $z_{j,n}$ such that each finite element's approximation of the residual is equal to zero. This leads to one residual equation r_m for each element:

$$
\begin{aligned}
r_m = 0 &= \int r(\mathbf{x}_1)\psi_m(\mathbf{x}_1)d\mathbf{x}_1 \mid m \in \{0, \dots, N_b\} && (8.12)\\
&= \int \left(\sum_{n=1}^{N_b}\left(\int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)z_{j,n}\psi_n(\mathbf{x}_2)d\mathbf{x}_2 - \lambda_j z_{j,n}\psi_n(\mathbf{x}_1)\right)\right)\psi_m(\mathbf{x}_1)d\mathbf{x}_1 && (8.13)\\
&= \sum_{n=1}^{N_b}\left(\int\int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)\psi_n(\mathbf{x}_2)\psi_m(\mathbf{x}_1)d\mathbf{x}_2 d\mathbf{x}_1\right) z_{j,n} && (8.14)\\
&\quad - \sum_{n=1}^{N_b}\left(\int \psi_n(\mathbf{x}_1)\psi_m(\mathbf{x}_1)d\mathbf{x}_1\right) z_{j,n}\lambda_j
\end{aligned}
$$

This can be expressed as a generalized eigen-problem of N_b linear equations:

$$\mathbf{CZ} - \mathbf{BZ\Lambda} = \mathbf{0} \qquad (8.15)$$

where

$$
\begin{aligned}
&\mathbf{C} \in \mathbb{R}^{N_b \times N_b} && (8.16)\\
&C_{i,j} = \int\int C_{pp}(\mathbf{x}_1, \mathbf{x}_2)\psi_n(\mathbf{x}_2)\psi_m(\mathbf{x}_1)d\mathbf{x}_2 d\mathbf{x}_1 && (8.17)\\
&\mathbf{B} \in \mathbb{R}^{N_b \times N_b} && (8.18)\\
&B_{i,j} = \int \psi_n(\mathbf{x}_1)\psi_m(\mathbf{x}_1)d\mathbf{x}_1 && (8.19)
\end{aligned}
$$

$$\mathbf{\Lambda} \in \mathbb{R}^{N_y \times N_y} \tag{8.20}$$

$$\Lambda_{i,j} = \delta_{i,j}\lambda_i \tag{8.21}$$

$$\mathbf{Z} \in \mathbb{R}^{N_b \times N_y} \tag{8.22}$$

$$Z_{i,j} = z_{i,j} \tag{8.23}$$

After integrating to find the entries of $\mathbf{C}$ and $\mathbf{B}$ (likely via numerical quadrature), solving this problem for $\mathbf{Z}$ and the diagonal matrix $\mathbf{\Lambda}$ yields all the coefficients $z_{j,n}$ from Equation 8.8 along with the eigenvalues λ_j; the eigenfunctions can then be approximated at any point by inserting any chosen value of $\mathbf{x}$ into that equation, even if such a point was not used to build the approximation – the finite elements can be used to approximate a value of $\phi_j(\mathbf{x})$ at entirely arbitrary locations. Thus, the mesh used to model $\phi(\mathbf{x})$ does not strictly need to match the one used to model $p(\mathbf{x})$; this can be advantageous in cases where computing ϕ becomes more expensive than the actual intended use of p. Note, though, that the matrix $\mathbf{C}$ cannot have more than N_b eigenpairs, which places an upper limit on $N_y \leq N_b$.

Once the vectors $\boldsymbol{\phi}(\mathbf{x}_i) \in \mathbb{R}^{N_u}$ for each eigenfunction have been calculated at all discretized $\mathbf{x}_i$, random samples of the discretized field $\mathbf{p}$ – originally a continuous problem with infinite degrees of freedom – can be very quickly computed by inserting a finite number of random scalar samples $\mathbf{y}$ into the truncated KLE:

$$p(\mathbf{x}, \omega) \approx \hat{p}(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{N_y} y_j \lambda_j \phi_j(\mathbf{x}) \tag{8.24}$$

This can be made even simpler by concatenating the column-vectors into a matrix $\mathbf{\Phi}$ and re-using the matrix $\mathbf{\Lambda}$ from above:

$$\mathbf{\Phi} = \left[\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \dots, \boldsymbol{\phi}_{N_y}\right] \in \mathbb{R}^{N_u \times N_y} \tag{8.25}$$

$$\hat{\mathbf{p}}(\mathbf{y}) = \mathbf{\Phi}\mathbf{\Lambda}\mathbf{y} \tag{8.26}$$

$$p_i \approx \hat{p}(\mathbf{x}_i, \mathbf{y}) \tag{8.27}$$

Note that every step up until the application of $\mathbf{y}$ is purely deterministic; thus, $\mathbf{\Phi}$ and $\mathbf{\Lambda}$ only need to be built once, after which an arbitrary number of random samples of $\mathbf{p}$ can be drawn.

8.1.3 Sampling the Diffusivity Field

For the specific problem considered here, the goal is to compute the lognormal field $k(\mathbf{x}, \omega) \approx k(\mathbf{x}, \mathbf{y})$, which is defined to be the exponential of a Gaussian process – i.e., the exponential e^g of a process $g(\mathbf{x}, \mathbf{y})$ wherein any finite subset of points $\{g(\mathbf{x}_i)\} \mid i = \{1, 2, \dots\}$ will have a multivariate normal distribution. Thus, to compute discretized samples of the entire field $\mathbf{k}(\mathbf{y})$, one sets $p = g$, chooses some two-dimensional covariance function $C_{gg}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = C_{gg}([\mathrm{x}_1{}^{(a)}, \mathrm{x}_2{}^{(a)}], [\mathrm{x}_1{}^{(b)}, \mathrm{x}_2{}^{(b)}])$, and proceeds as above.

When considering options for C_{gg}, it is important to note that the integrals in Equation 8.14 are taken over the entire two-dimensional geometric domain; thus, entries of the matrix $\mathbf{C}$ theoretically require quadruple integrals. Even using efficient sparse numerical quadrature, this can be impractically expensive. Fortunately, under certain circumstances, the product of two one-dimensional eigenfunctions (again, much like the product of two finite elements or two boundary conditions) can itself be an eigenfunction of a two-dimensional process. This is only possible when the covariance function is multiplicativly separable, such that C_{gg} is a product of one-dimensional functions. For this reason, the covariance for this problem is chosen to be the squared exponential, which is a function of the distance d between two points:

$$d = \|\mathbf{x}^{(a)} - \mathbf{x}^{(b)}\| = \sqrt{\sum (\mathrm{x}_i{}^{(a)} - \mathrm{x}_i{}^{(b)})^2} \tag{8.28}$$

$$C_{gg}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = C_{gg}(d) = \sigma^2 e^{-(d/l_c)^2} \tag{8.29}$$

$$= \sigma^2 e^{-((\mathrm{x}_1{}^{(a)} - \mathrm{x}_1{}^{(b)})/l_c)^2 - ((\mathrm{x}_2{}^{(a)} - \mathrm{x}_2{}^{(b)})/l_c)^2} \tag{8.30}$$

$$= \sigma^2 e^{-((\mathrm{x}_1{}^{(a)} - \mathrm{x}_1{}^{(b)})/l_c)^2} e^{-((\mathrm{x}_2{}^{(a)} - \mathrm{x}_2{}^{(b)})/l_c)^2} \tag{8.31}$$

$$= C_{gg}(\mathrm{x}_1{}^{(a)}, \mathrm{x}_1{}^{(b)}) C_{gg}(\mathrm{x}_2{}^{(a)}, \mathrm{x}_2{}^{(b)}) \tag{8.32}$$

where σ^2 is the variance and l_c is the correlation length (a scaling parameter); for this problem, $\sigma = 0.5$ and $l_c = 0.01$. Given a one-dimensional C_{gg} of this form, the program first finds the eigenfunctions $\phi_i(x_1)$ and eigenvalues λ_i for the one-dimensional problem; takes the outer product of the vector of eigenvalues $\boldsymbol{\lambda}\boldsymbol{\lambda}^T$; finds the largest eigenvalues $\lambda_{ij} = \lambda_i \lambda_j$ of the new process; and finally pulls out the corresponding two-dimensional eigenfunctions $\phi_{ij}(\mathbf{x}) = \phi_i(x_1)\phi_j(x_2)$.

As mentioned in the previous section, the one-dimensional linear finite elements $\psi(x_1)$ used to approximate $\phi(x_1)$ might be placed along a lower-resolution coordinate mesh than that for $\phi(x_1)$ itself; however, the separable nature of this C_{gg} makes that unnecessary. Thus, for the sake of using identical meshes and identical bilinear elements, the problem discretizes the field with a square grid, i.e., with $N_{b1} = N_{u1}$ nodes uniformly spaced in each direction such that $N_b = N_u = {N_{u1}}^2$. Given this configuration, example eigenfunctions and samples of k are shown in Figure 8.2 and Figure 8.3, respectively.

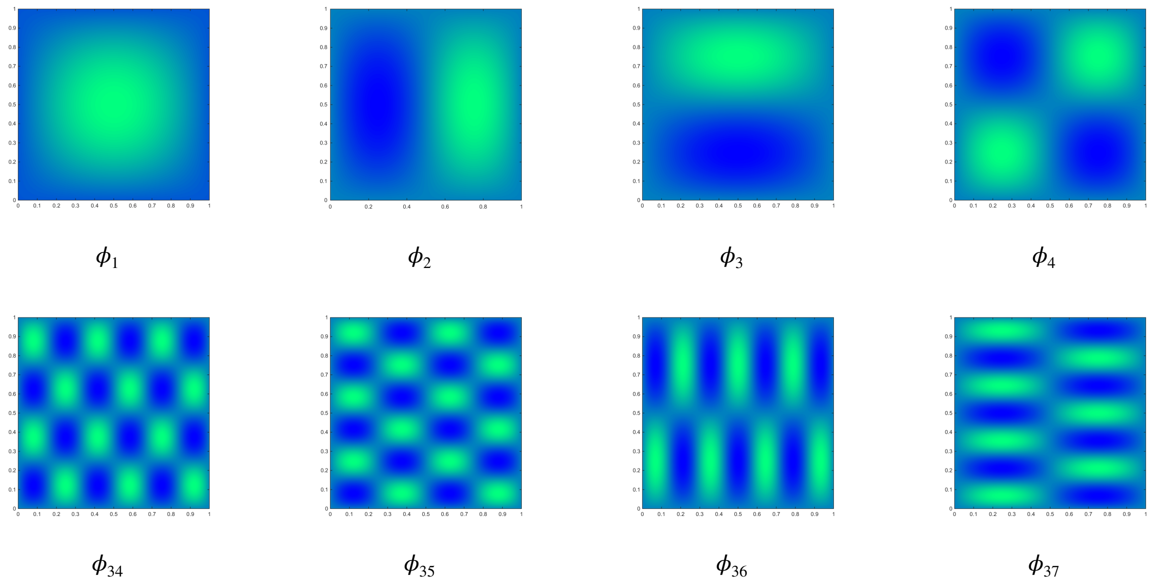


Figure 8.2: Selected eigenfunctions (#1-4 and #34-37) of the Gaussian covariance kernel. Early eigenfunctions capture broad behavior; successive eigenfunctions focus on increasingly fine-grained features.

8.1.4 PDE Discretization, Solution, and Quantity of Interest

Once k is known, the next step is to discretize Equation 8.1 in order to solve for equilibrium at $\frac{\partial u}{\partial t} = 0$ – that is, discretize

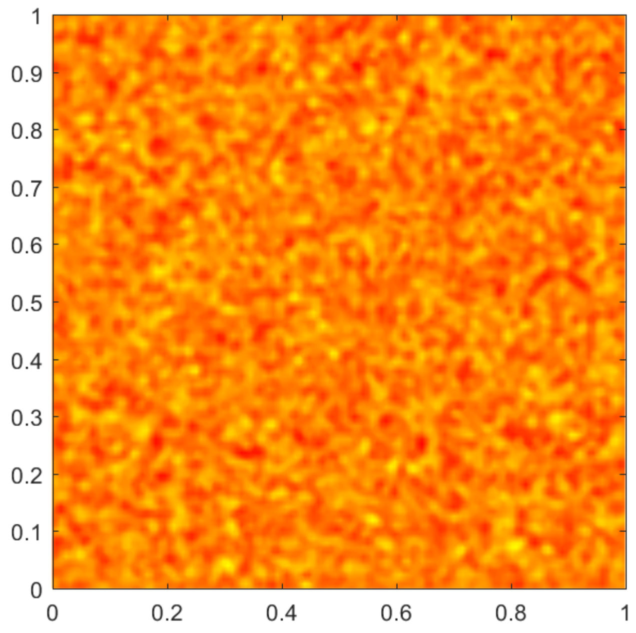


Figure 8.3: Sample of the diffusion k-field for $N_u = 128^2 = 16384$, $N_y = 10000$, $\sigma = 0.5$ and $l_c = 0.01$.

$$\left(\frac{\partial k}{\partial x_1}\frac{\partial u}{\partial x_1} + \frac{\partial k}{\partial x_2}\frac{\partial u}{\partial x_2}\right) + k\left(\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial {x_2}^2}\right) = 0 \tag{8.33}$$

and solve for $u(\mathbf{x}) = u_{eq}(\mathbf{x})$ at every discretized value of $\mathbf{x}$. As stated above, the computational mesh is a uniformly spaced square grid with N_{u1} nodes in each direction; denote the spacing of this grid as $\Delta x = \frac{1}{N_{u1}-1}$ and index each node as $\mathbf{x}_{n,m} = [\mathrm{x}_1^{(\mathrm{n})}, \mathrm{x}_2^{(\mathrm{m})}]$. The adjacent points are then $[\mathrm{x}_1^{(\mathrm{n})} + \Delta x, \mathrm{x}_2^{(\mathrm{m})}] = \mathbf{x}_{n+1,m}$, etc.

The simplest centered finite difference[2] formulas are sufficient to discretize Equation 8.33, and can now be denoted as:

[2] Note, this is one of the few times in this work where a "finite difference" does *not* refer to a derivative approximation across an entire system.

$$
\begin{aligned}
\frac{\partial k}{\partial x_1} &\approx \frac{k(x+\Delta x, y) - k(x-\Delta x, y)}{2\Delta x} \\
&= \frac{k_{n+1,m} - k_{n-1,m}}{2\Delta x} \\
\frac{\partial u}{\partial x_1} &\approx \frac{u(x+\Delta x, y) - u(x-\Delta x, y)}{2\Delta x} \\
&= \frac{u_{n+1,m} - u_{n-1,m}}{2\Delta x} \\
\frac{\partial^2 u}{\partial {x_1}^2} &\approx \frac{u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)}{\Delta x^2} \\
&= \frac{u_{n+1,m} - 2u_{n,m} + u_{n-1,m}}{\Delta x^2}
\end{aligned}
$$

and similarly for derivatives with respect to x_2. Inserting these formulas into Equation 8.33 and rearranging yields:

$$
\begin{aligned}
&\frac{1}{\Delta x^2}\left(k_{n,m} - \frac{1}{4}\left(k_{n+1,m} - k_{n-1,m}\right)\right) u_{n-1,m} \\
+\ &\frac{1}{\Delta x^2}\left(k_{n,m} - \frac{1}{4}\left(k_{n,m+1} - k_{n,m-1}\right)\right) u_{n,m-1} \\
+\ &\qquad\qquad\qquad -\frac{4}{\Delta x^2} k_{n,m}\ u_{n,m} \\
+\ &\frac{1}{\Delta x^2}\left(k_{n,m} + \frac{1}{4}\left(k_{n+1,m} - k_{n-1,m}\right)\right) u_{n+1,m} \\
+\ &\frac{1}{\Delta x^2}\left(k_{n,m} + \frac{1}{4}\left(k_{n,m+1} - k_{n,m-1}\right)\right) u_{n,m+1} \ = \ 0
\end{aligned}
\tag{8.34}
$$

Note that holding the mesh boundary values fixed reduces the number of degrees of freedom (DoFs) – thus, while k must be provided at each of $N_u = N_{u1}^2$ mesh nodes, u is only found at $N_d = N_{d1}^2 = (N_{u1} - 2)^2$ DoF points contained in the sub-mesh that excludes the edge nodes, indexed with $n, m \in \{2, \dots, N_u - 1\}$. To keep this context clear, denote vectors of mesh values with a tilde, such that the k-field becomes $\tilde{\mathbf{k}} \in \mathbb{R}^{N_u}$ while the u-field is simply $\mathbf{u} \in \mathbb{R}^{N_d}$. Further denote the vector of boundary values as $\tilde{\mathbf{v}} \in \mathbb{R}^{N_u}$, which is defined to be zero at every DoF node. Note that any given double-indexed $u_{n,m}$ will always be in the DoF sub-mesh (and thus represented in $\mathbf{u}$) but adjacent values $u_{n+1,m}$, etc., may not. Use of a single index ($\tilde{k}_i$, u_i, etc.) denotes a single entry of a vector, where the range of the index can be inferred from context.

Using this notation, if $3 \leq n, m \leq N - 2$ – i.e., if every neighboring value of u is unknown – Equation 8.34 immediately yields five coefficients that can be placed into row i of a matrix $\mathbf{A}(\tilde{\mathbf{k}}) \in \mathbb{R}^{N_d \times N_d}$, where $i = (m - 1) + N_{d1}(n - 2)$. In the remaining cases where one or more of the u-values are held fixed by the Dirichlet boundary conditions, the product of the coefficient and the value must be added to a vector $\mathbf{b}(\tilde{\mathbf{k}}, \tilde{\mathbf{v}}) \in \mathbb{R}^{N_d}$, such that $\mathbf{Au} + \mathbf{b} = \mathbf{0}$. Once $\mathbf{A}$ and $\mathbf{b}$ are constructed in this way, the equilibrium is found via linear solution as $\mathbf{u} = \mathbf{u}_{eq} = -\mathbf{A}^{-1}\mathbf{b}$.

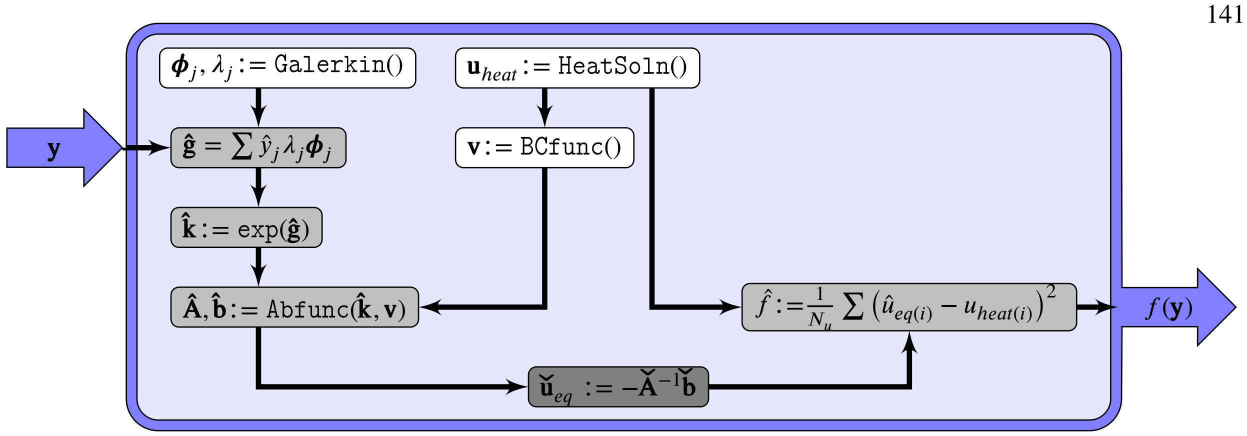
As a rule, most engineering applications only solve for such an equilibrium in order to extract some quantity of interest (QoI) f from the converged solution field. Here, $f(\mathbf{u}_{eq})$ is arbitrarily chosen as the mean squared difference between $\mathbf{u}_{eq}$ and the constant-k solution $\mathbf{u}_{heat}$ chosen in Section 8.1.1 – that is,

$$f(\mathbf{u}_{eq}) = \frac{1}{N_d} \sum_{i=1}^{N_d} \left(u_{eq(i)} - u_{heat(i)}\right)^2 \tag{8.35}$$

Even given this specific discretization and QoI, there are multiple ways to implement this problem. For rigor and additional verification of the analytic model, the experiments described in this chapter use two different programming approaches (both in MATLAB) to build $\mathbf{A}$ and $\mathbf{b}$ and to compute f. The first formulation relies heavily on scalar calculations: each diffusivity value k_i is found via scalar operations and Equation 8.24; likewise, when computing each row of $\mathbf{A}$, each entry is found one at a time as given in Equation 8.34. The only matrix operations are found in the system solver, after which yet more scalar operations are used to compute the output via Equation 8.35. The data-flow of this first program is shown in Figure 8.4, which adopts the same tilde notation for all whole-mesh vectors.

Nilpotent Matrix Differentiation might still be helpful for this problem formulation, even if it were only applied to the solution step – using the manual matrix-inverse differentiation formula from Equation 3.71 is trivial for a single derivative, but repeated application might become laborious for higher derivatives, especially when the order or number of independent variables is expected to change across program runs. However, this is not how NMD is truly expected to be used – its standout feature is the way derivatives propagate across *many* matrix routines, with most[3] new code required only before the first routine and right after the last.

[3] Exceptions to the rule generally deal with treatment of vectors or other limitations discussed in Section 6.2, but are very simple to apply.



$\boldsymbol{\phi}_j, \lambda_j :=$ Galerkin()
$\mathbf{u}_{heat} :=$ HeatSoln()
$\mathbf{y}$
$\hat{\mathbf{g}} = \sum \hat{y}_j \lambda_j \boldsymbol{\phi}_j$
$\mathbf{v} :=$ BCfunc()
$\hat{\mathbf{k}} :=$ exp($\hat{\mathbf{g}}$)
$\hat{\mathbf{A}}, \hat{\mathbf{b}} :=$ Abfunc($\hat{\mathbf{k}}, \mathbf{v}$)
$\hat{f} := \frac{1}{N_u} \sum \left(\hat{u}_{eq(i)} - u_{heat(i)}\right)^2$
$f(\mathbf{y})$
$\check{\mathbf{u}}_{eq} := -\check{\mathbf{A}}^{-1}\check{\mathbf{b}}$

Figure 8.4: Data-flow of the first, scalar-focused implementation of the diffusion problem. White functions represent values that remain constant between function runs; gray represents user-written scalar code with varying inputs and outputs; dark gray represents code that calls inaccessible matrix functions; and blue represents the program API.

For this reason, a second formulation of the program is also used as a more rigorous test for NMD. In this implementation, every operation is replaced by a matrix variant. First, a matrix-matrix-vector product like that in Equation 8.26 is used to calculate $\tilde{\mathbf{g}}(\mathbf{y})$:

$$\tilde{\mathbf{g}}(\mathbf{y}) = \boldsymbol{\Phi}\boldsymbol{\Lambda}\mathbf{y} \tag{8.36}$$

In the case of many eigenfunctions, this is significantly faster than the scalar variant. Next, rather than operate on one element at a time, the vector $\tilde{\mathbf{g}}$ is converted to a diagonal matrix $\tilde{\mathbf{G}}$, which allows the calculation of a similarly diagonal $\tilde{\mathbf{K}} = \texttt{expm}(\tilde{\mathbf{G}})$, where `expm` is the matrix exponential function; this has an analogous definition to the scalar exponential:

$$\texttt{expm}(\tilde{\mathbf{G}}) \equiv e^{\tilde{\mathbf{G}}} = \sum_{n=0}^{\infty} \frac{\tilde{\mathbf{G}}^n}{n!} \tag{8.37}$$

For a diagonal matrix, this was experimentally found to converge to machine precision after about $n \approx 10$ terms. A sparse implementation of `expm` is thus trivially inexpensive to run compared to the cost of calculating $\tilde{\mathbf{g}}(\mathbf{y})$.

Note that, as Equation 8.34 requires k-values even for the boundary nodes, $\tilde{\mathbf{K}}$ cannot simply be "stripped down" by removing the boundaries. Instead, in order to construct $\mathbf{A} \in \mathbb{R}^{N_d \times N_d}$ and $\mathbf{b} \in \mathbb{R}^{N_d}$, temporary matrices $\tilde{\mathbf{A}} \in$

$\mathbb{R}^{N_u \times N_u}$ and $\tilde{\mathbf{b}} \in \mathbb{R}^{N_u}$ must be built first, then mapped down to the smaller index set. The off-diagonals of these larger matrices can be calculated using a set of four "shift" matrices:

$$\tilde{\mathbf{S}}_{(\Delta n, \Delta m)} \in \left\{ \tilde{\mathbf{S}}_{(-1,0)}, \tilde{\mathbf{S}}_{(0,-1)}, \tilde{\mathbf{S}}_{(1,0)}, \tilde{\mathbf{S}}_{(0,1)} \right\} \in \mathbb{R}^{N_u \times N_u} \tag{8.38}$$

These are each sparse matrices with a single diagonal of all ones, offset from the main diagonal such that matrix multiplication will "shift" a value from its own index to that of a neighbor in the indicated direction. That is:

$$\tilde{K}_{ii} = \left[\tilde{\mathbf{K}}\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}\right]_{ij} = \left[\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}{}^{T}\tilde{\mathbf{K}}\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}\right]_{jj} \tag{8.39}$$

$$\tilde{K}_{jj} = \left[\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}\tilde{\mathbf{K}}\right]_{ij} = \left[\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}\tilde{\mathbf{K}}\tilde{\mathbf{S}}_{(\Delta n, \Delta m)}{}^{T}\right]_{ii} \tag{8.40}$$

$$= \tilde{K}_{(\Delta n, \Delta m)(ii)} \tag{8.41}$$

$$i = m + N_{u1}(n-1) \tag{8.42}$$

$$j = (m + \Delta m) + N_{u1}((n + \Delta n) - 1) \tag{8.43}$$

Boundary values that have no neighbor will be "shifted off" the matrix entirely, i.e., set to zero. Note that $\tilde{\mathbf{S}}_{(1,0)} = \tilde{\mathbf{S}}_{(-1,0)}{}^{T}$ and $\tilde{\mathbf{S}}_{(0,1)} = \tilde{\mathbf{S}}_{(0,-1)}{}^{T}$ – that is, to shift to the opposite neighbor, simply transpose the shift matrix. Separate indices are used here for ease of notation.

Only once $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{b}}$ are built can the extra information be stripped out. This is done using a map matrix $\mathbf{M} \in \mathbb{R}^{N_u \times N_d}$, specially built so that $\mathbf{A} = \mathbf{M}^T\tilde{\mathbf{A}}\mathbf{M}$ and $\mathbf{b} = \mathbf{M}^T\tilde{\mathbf{b}}$. This matrix is also sparse, with a single 1 in each column (and thus having some empty rows); it is built during program initialization by finding the mesh index i corresponding to each DoF index j, then setting $M_{ij} = 1$. Given these matrices and the BC vector $\tilde{\mathbf{v}}$, the off-diagonal coefficients from Equation 8.34 can be built with the following steps:

$\forall \Delta n, \Delta m :$

$$\tilde{\mathbf{K}}_{(\Delta n,\Delta m)} = \tilde{\mathbf{S}}_{(\Delta n,\Delta m)} \tilde{\mathbf{K}} \tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^{T} \tag{8.44}$$

$$\tilde{\mathbf{C}}_{(\Delta n,\Delta m)} = \frac{1}{\Delta x^2} \left(\tilde{\mathbf{K}} + \frac{1}{4} \left(\tilde{\mathbf{K}}_{(\Delta n,\Delta m)} - \tilde{\mathbf{K}}_{(-\Delta n,-\Delta m)} \right) \right) \tag{8.45}$$

$$\tilde{\mathbf{A}}_{(\Delta n,\Delta m)} = \tilde{\mathbf{C}}_{(\Delta n,\Delta m)} \tilde{\mathbf{S}}_{(\Delta n,\Delta m)} \tag{8.46}$$

$$\tilde{\mathbf{b}}_{(\Delta n,\Delta m)} = \tilde{\mathbf{A}}_{(\Delta n,\Delta m)} \mathbf{v} \tag{8.47}$$

These steps calculate the coefficients corresponding to each neighbor on the full mesh; once they are found, they must be summed and mapped to the DoF sub-mesh:

$$\mathbf{A} = \mathbf{M}^T \left(-\frac{4}{\Delta x^2} \tilde{\mathbf{K}} + \sum_{\Delta n,\Delta m} \tilde{\mathbf{A}}_{(\Delta n,\Delta m)} \right) \mathbf{M} \tag{8.48}$$

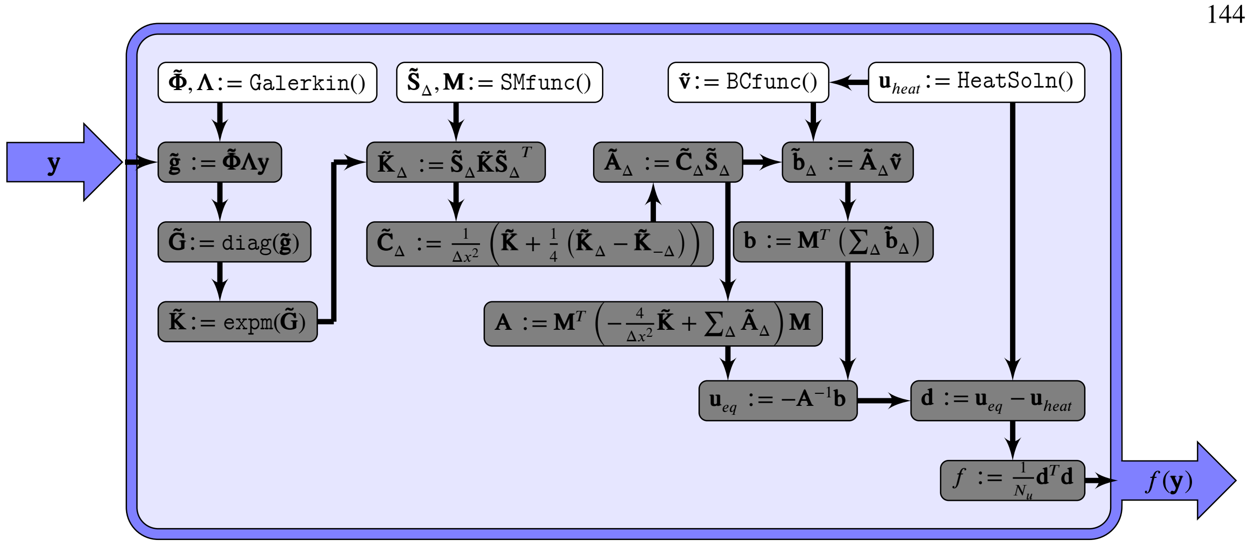
$$\mathbf{b} = \mathbf{M}^T \left(\sum_{\Delta n,\Delta m} \tilde{\mathbf{b}}_{(\Delta n,\Delta m)} \right) \tag{8.49}$$

Finally, once $\mathbf{A}$ and $\mathbf{b}$ have been used to solve for $\mathbf{u}_{eq}$, the quantity of interest from Equation 8.35 is also expressed entirely with vector operations as:

$$\mathbf{d} = \mathbf{u}_{eq} - \mathbf{u}_{heat} \tag{8.50}$$

$$f(\mathbf{u}_{eq}) = \frac{1}{N_u} \mathbf{d}^T \mathbf{d} \tag{8.51}$$

The data-flow of this second program is shown in Figure 8.5. This is a larger diagram that Figure 8.4 because the inner workings of `Abfunc()`, which were hidden there, are broken out here; in terms of implementation, it is no more complex. Note that for the deterministic version of the problem, the first three blocks of the matrix problem (i.e., those for $\mathbf{g}$, $\mathbf{G}$, and $\mathbf{K}$) are bypassed so that a diagonal matrix $\mathbf{K} = k_0 \mathbf{I}$ can be passed to the discretization process;



$\tilde{\mathbf{\Phi}}, \mathbf{\Lambda} :=$ Galerkin()
$\tilde{\mathbf{S}}_\Delta, \mathbf{M} :=$ SMfunc()
$\tilde{\mathbf{v}} :=$ BCfunc()
$\mathbf{u}_{heat} :=$ HeatSoln()
$\mathbf{y}$
$\tilde{\mathbf{g}} := \tilde{\mathbf{\Phi}}\mathbf{\Lambda}\mathbf{y}$
$\tilde{\mathbf{K}}_\Delta := \tilde{\mathbf{S}}_\Delta \tilde{\mathbf{K}} \tilde{\mathbf{S}}_\Delta^T$
$\tilde{\mathbf{A}}_\Delta := \tilde{\mathbf{C}}_\Delta \tilde{\mathbf{S}}_\Delta$
$\tilde{\mathbf{b}}_\Delta := \tilde{\mathbf{A}}_\Delta \tilde{\mathbf{v}}$
$\tilde{\mathbf{G}} :=$ diag($\tilde{\mathbf{g}}$)
$\tilde{\mathbf{C}}_\Delta := \frac{1}{\Delta x^2}\left(\tilde{\mathbf{K}} + \frac{1}{4}\left(\tilde{\mathbf{K}}_\Delta - \tilde{\mathbf{K}}_{-\Delta}\right)\right)$
$\mathbf{b} := \mathbf{M}^T\left(\sum_\Delta \tilde{\mathbf{b}}_\Delta\right)$
$\tilde{\mathbf{K}} :=$ expm($\tilde{\mathbf{G}}$)
$\mathbf{A} := \mathbf{M}^T\left(-\frac{4}{\Delta x^2}\tilde{\mathbf{K}} + \sum_\Delta \tilde{\mathbf{A}}_\Delta\right)\mathbf{M}$
$\mathbf{u}_{eq} := -\mathbf{A}^{-1}\mathbf{b}$
$\mathbf{d} := \mathbf{u}_{eq} - \mathbf{u}_{heat}$
$f := \frac{1}{N_u}\mathbf{d}^T\mathbf{d}$
$f(\mathbf{y})$

Figure 8.5: Data-flow of the second, matrix-only implementation of the diffusion problem. White functions represent values that remain constant between function runs, dark gray represents code that calls inaccessible matrix functions, and blue represents the program API. Every operation on a variable with a Δ subscript is applied to each of $\Delta = (\Delta n, \Delta m) \in \{(1,0), (-1,0), (0,1), (0,-1)\}$.

otherwise, the process remains the same.

8.1.5 Undifferentiated Program Results

Once again, this problem was chosen simply to benchmark differentiation algorithms; to some extent, the undifferentiated (real-valued) outputs are beside the point. Nonetheless, it is useful to review the system behavior to clarify precisely what the base program is meant to output.

A sample of the k-field was shown in Figure 8.3. Given a diffusivity sample such as that one, the diffusive system reaches a equilibrium such as the one shown in Figure 8.6; this equilibrium represents the steady-state concentration of some arbitrary solute after it has diffused across the spatial domain. Note in the figure that the deterministic behavior imposed by the boundary conditions $u_{heat}(\mathbf{x}) = \left(m_1 x_1 + b_1\right)\left(m_2 x_2 + b_2\right)$ for $m_1 = b_1 = 500$ and $m_2 = b_2 = 1000$ dominates the solution; however, the true quantity of interest is the stochastic deviation from deterministic expectations.

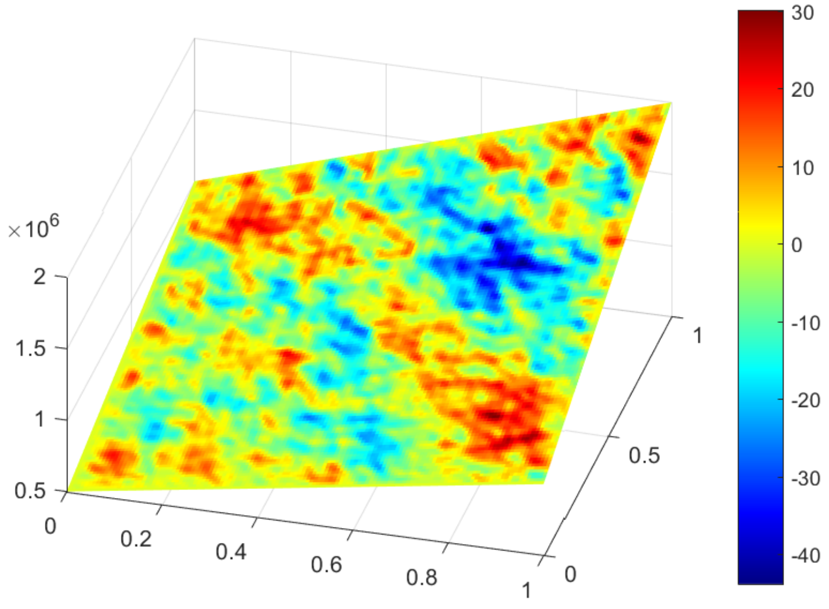


Figure 8.6: Example diffusion equilibrium solution with $N_y = 10000$ eigenfunctions. Elevation represents deterministic solution; color represents delta between stochastic and deterministic solutions.

Recall that the diffusion equation is:

$$\frac{\partial u}{\partial t} = \left(\frac{\partial k}{\partial x_1} \frac{\partial u}{\partial x_1} + \frac{\partial k}{\partial x_2} \frac{\partial u}{\partial x_2} \right) + k \left(\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial {x_2}^2} \right)$$

Further recall that the boundary conditions were explicitly constructed to set both spatial second derivatives to zero; therefore, considering that stochastic behavior constitutes so little of the solution's total magnitude, one can expect the second term of the equation to be negligible – the deviation should be dominated by first-order behavior. Given this, examine the two remaining terms knowing that $^{\partial k}/_{\partial x_1}$ and $^{\partial k}/_{\partial x_2}$ cannot be counted on to cancel. To maintain the equilibrium $^{\partial u}/_{\partial t} = 0$, if $^{\partial k}/_{\partial x_1}$ is large then $^{\partial u}/_{\partial x_1}$ will likely be small, and likewise for $^{\partial k}/_{\partial x_2}$ and $^{\partial u}/_{\partial x_2}$. However, a small spatial derivative of the solute implies that the field is not conforming to the steep underlying deterministic behavior – and so, in any location with *small* k-derivatives, u must "catch up". This expectation is confirmed in Figure 8.7, which demonstrates a close relationship between the two spatial gradient magnitudes.[4]

[4] Note that these gradients were calculated by finite differences across the mesh – unfortunately, discretization of a PDE makes the true spatial derivatives unavailable to NMD or any other AD method.

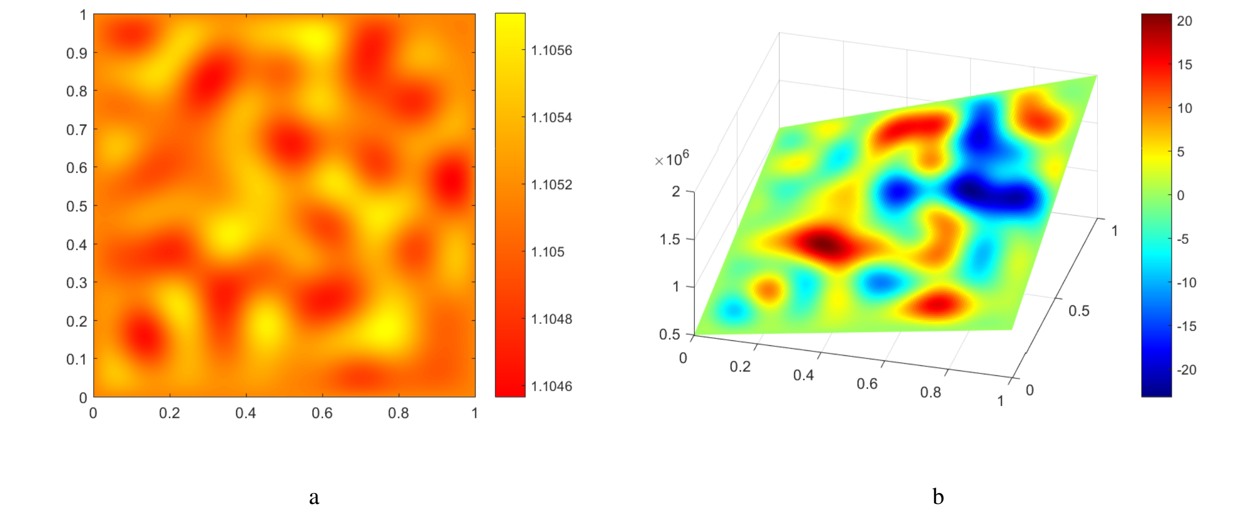
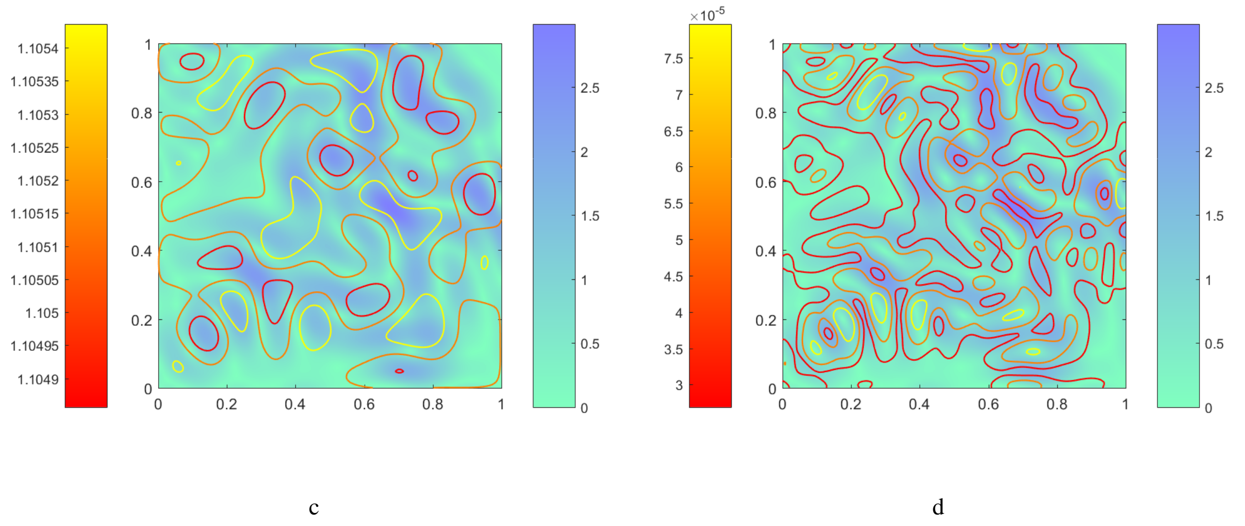



Figure 8.7: (a) Sample of k with $N_y = 100$ eigenfunctions. (b) The corresponding stochastic equilibrium solution $\mathbf{u}_{eq}$ and its deviation $\mathbf{d} = \mathbf{u}_{eq} - \mathbf{u}_{heat}$ from the deterministic solution. (See Figure 8.6 caption). (c) Magnitude of the same deviation's spatial gradient vector $||\nabla_x \mathbf{d}||$ (green/blue) overlaid with contours of k (red/orange). (d) $||\nabla_x \mathbf{d}||$ overlaid with contours of $||\nabla_x \mathbf{k}||$. In general and as expected (see text), the field deviation is steepest at the local minima and maxima of k, i.e, the minima of $||\nabla_x \mathbf{k}||$.

Importantly, both versions of the program (summarized in Figure 8.4 and Figure 8.5) produce identical outputs $f(\mathbf{y})$ for identical input vectors $\mathbf{y}$, and when differentiated (as will be shown in Figure 8.9 and Figure 8.10) produce identical derivatives of f with respect to $\mathbf{y}$. With this verification in hand, NMD benchmarks will only be reported for the second problem, i.e., the one composed purely of matrix operations.

8.1.6 Adjoint Formulation

As already mentioned repeatedly, one of the most promising applications of Nilpotent Matrix Differentiation is the higher-order differentiation of the adjoint method, as that approach is itself based on matrix operations – namely, the manipulation of several key Jacobians. To give a proof-of-concept of this capability, the adjoint method must first be applied to the diffusion code. Specifically, the discrete variant discussed in Section 3.1.2 is implemented almost exactly as described there.

To compactly rephrase the method using variables from the (forward) diffusion problem, denote the partial derivative of any scalar z with respect to vector $\mathbf{w}$ as the gradient row vector $\mathbf{z}_{\mathbf{w}}$; likewise, denote the derivative of any vector $\mathbf{z}$ as the Jacobian matrix $\mathbf{Z}_{\mathbf{w}}$. Then, given input $\mathbf{y}$, solution $\mathbf{u}_{eq}(\mathbf{y})$, residual $\mathbf{r}(\mathbf{y}, \mathbf{u}_{eq}) = 0$, and output $f(\mathbf{u}_{eq})$, the bulk of the method's computational effort goes into solving the adjoint equation:

$$\mathbf{R}_{\mathbf{u}}^T \boldsymbol{\lambda} = -\mathbf{f}_{\mathbf{u}}^T \tag{8.52}$$

Solving this system yields the adjoint vector $\boldsymbol{\lambda}$; to compute the gradient ${}^{\mathrm{d}f}/_{\mathrm{d}\mathbf{y}}$, this can be inserted into the formula:

$$\frac{\mathrm{d}f}{\mathrm{d}\mathbf{y}} = \boldsymbol{\lambda}^T \mathbf{R}_{\mathbf{y}} + \mathbf{f}_{\mathbf{y}} \tag{8.53}$$

For the diffusion problem, the output f is given by Equation 8.50, above, and the residual is:

$$\mathbf{r} = \mathbf{A}(\mathbf{y})\mathbf{u}_{eq} + \mathbf{b}(\mathbf{y}) = 0 \tag{8.54}$$

The key derivatives required by the adjoint method are then:

$$\mathbf{f_u} = \frac{2}{N_u}(\mathbf{u}_{eq} - \mathbf{u}_{heat}) \tag{8.55}$$

$$\mathbf{f_y} = 0 \tag{8.56}$$

$$\mathbf{R_u} = \mathbf{A} \tag{8.57}$$

$$\left[\mathbf{R_y}\right]_{(\cdot,j)} = \frac{\partial \mathbf{r}}{\partial y_j} \tag{8.58}$$

$$= \frac{\partial \mathbf{A}}{\partial y_j}\mathbf{u}_{eq} + \frac{\partial \mathbf{b}}{\partial x_j} \tag{8.59}$$

The first three derivatives are trivial, but $\mathbf{R_y} \in \mathbb{R}^{N_d \times N_y}$ – the Jacobian of the residual with respect to the input – must be defined column-wise. The overwhelming majority of the new implementation work goes into finding this quantity efficiently. Referring to the original discretization Equation 8.34, it can be seen that **A** and **b** only depend on **y** through linear use of the diffusivity vector $\tilde{\mathbf{k}}$, which makes the first step of analytic differentiation straightforward:

$$\begin{aligned}
\frac{\partial r_i}{\partial y_j} = \frac{\partial r_{n,m}}{\partial y_j} = & \frac{1}{\Delta x^2}\left(\frac{\partial k_{n,m}}{\partial y_j} - \frac{1}{4}\left(\frac{\partial k_{n+1,m}}{\partial y_j} - \frac{\partial k_{n-1,m}}{\partial y_j}\right)\right) u_{n-1,m} \\
+ & \frac{1}{\Delta x^2}\left(\frac{\partial k_{n,m}}{\partial y_j} - \frac{1}{4}\left(\frac{\partial k_{n,m+1}}{\partial y_j} - \frac{\partial k_{n,m-1}}{\partial y_j}\right)\right) u_{n,m-1} \\
- & \frac{4}{\Delta x^2}\frac{\partial k_{n,m}}{\partial y_j} u_{n,m} \\
+ & \frac{1}{\Delta x^2}\left(\frac{\partial k_{n,m}}{\partial y_j} + \frac{1}{4}\left(\frac{\partial k_{n+1,m}}{\partial y_j} - \frac{\partial k_{n-1,m}}{\partial y_j}\right)\right) u_{n+1,m} \\
+ & \frac{1}{\Delta x^2}\left(\frac{\partial k_{n,m}}{\partial y_j} + \frac{1}{4}\left(\frac{\partial k_{n,m+1}}{\partial y_j} - \frac{\partial k_{n,m-1}}{\partial y_j}\right)\right) u_{n,m+1}
\end{aligned} \tag{8.60}$$

The derivatives of k are in turn defined by

$$\tilde{K}_{ii} = \tilde{k}_i = k_0 + \exp(\tilde{g}_i) \tag{8.61}$$

$$\tilde{g}_i = g_0 + \sum_{k=1}^{N_y} y_k \lambda_k \tilde{\phi}_{ik} \tag{8.62}$$

$$\frac{\partial \tilde{g}_i}{\partial y_j} = \lambda_j \tilde{\phi}_{ij} \tag{8.63}$$

$$\frac{\partial \tilde{K}_{ii}}{\partial y_j} = \frac{\partial \tilde{k}_i}{\partial y_j} = \lambda_j \tilde{\phi}_{ij} \exp(\tilde{g}_i) \tag{8.64}$$

Thus, $^{\partial \mathbf{A}}/_{\partial y_j}$ and $^{\partial \mathbf{b}}/_{\partial y_j}$ could be built by replacing $\tilde{\mathbf{k}}$ in Figure 8.4 and $\tilde{\mathbf{K}}$ in Figure 8.5 with $^{\partial \tilde{\mathbf{k}}}/_{\partial y_j}$ and $^{\partial \tilde{\mathbf{K}}}/_{\partial y_j}$, respectively. This yields the most trivial way to build $\mathbf{R_y}$: construct $^{\partial \mathbf{A}}/_{\partial y_j}$ and $^{\partial \mathbf{b}}/_{\partial y_j}$ for each y_j, then repeatedly apply Equation 8.59 to find each column; this is, however, quite inefficient. A better way is reuse the diagonal matrix $\tilde{\mathbf{G}}$ from the matrix-focused implementation of the problem (but not the diagonal $\tilde{\mathbf{K}}$) and build two new (dense) Jacobians $\tilde{\mathbf{G}}_\mathbf{y}, \tilde{\mathbf{K}}_\mathbf{y} \in \mathbb{R}^{N_u \times N_y}$:

$$\tilde{\mathbf{g}} = \tilde{\mathbf{\Phi}} \mathbf{\Lambda} \mathbf{y} \tag{8.65}$$

$$\frac{\partial \tilde{\mathbf{g}}}{\partial y_j} = \lambda_j \phi_{(\cdot,j)} \tag{8.66}$$

$$\tilde{\mathbf{G}}_\mathbf{y} = \tilde{\mathbf{\Phi}} \mathbf{\Lambda} \tag{8.67}$$

$$\tilde{\mathbf{K}}_\mathbf{y} = \texttt{expm}(\tilde{\mathbf{G}}) \tilde{\mathbf{G}}_\mathbf{y} \tag{8.68}$$

Note that Equation 8.60 can be used to define values of $\mathbf{R_y} \in \mathbb{R}^{N_d \times N_y}$ at every DoF node, but must index into the boundaries to do so. Thus, the goal is to first build a whole-mesh matrix $\tilde{\mathbf{R}}_\mathbf{y} \in \mathbb{R}^{N_u \times N_y}$, then trim it with the map matrix $\mathbf{M}$. This procedure likewise needs access to a vector $\tilde{\mathbf{u}} \in \mathbb{R}^{N_u}$ of species concentrations across the whole mesh; this is built simply by inserting the values of $\mathbf{u}_{eq} \in \mathbb{R}^{N_d}$ into the corresponding locations in the BC vector $\tilde{\mathbf{v}} \in \mathbb{R}^{N_u}$ (i.e., locations that were set to zero in the original $\tilde{\mathbf{v}}$).

In the original matrix-focused forward problem, $\tilde{\mathbf{k}}$ was turned into a diagonal matrix $\tilde{\mathbf{K}}$, a function of which was ultimately multiplied with the column vector $\tilde{\mathbf{v}}$. In this case, as $\tilde{\mathbf{K}}_\mathbf{y}$ is already a matrix, it cannot be diagonalized; instead, $\tilde{\mathbf{u}}$ becomes the diagonal matrix $\tilde{\mathbf{U}}$. Each of the four neighbor terms from Equation 8.60 require an offset u and

k, which can be calculated with the same shift matrices from earlier:

$$\tilde{u}_i = u_{n,m} \tag{8.69}$$

$$\tilde{u}_q = u_{(n+\Delta n, m+\Delta m)} = \tilde{\mathbf{U}}_{qq} \tag{8.70}$$

$$= \tilde{\mathbf{U}}_{(\Delta n,\Delta m)(i,i)} \tag{8.71}$$

$$= \left[\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{U}}\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right]_{ii} \tag{8.72}$$

$$\frac{\partial \tilde{k}_i}{\partial y_j} = \frac{\partial k_{n,m}}{\partial y_j} = \tilde{\mathbf{K}}_{\mathbf{y}(i,j)} \tag{8.73}$$

$$\frac{\partial \tilde{k}_q}{\partial y_j} = \frac{\partial k_{(n+\Delta n,m+\Delta m)}}{\partial y_j} \tag{8.74}$$

$$= \tilde{\mathbf{K}}_{\mathbf{y}(\Delta n,\Delta m)(i,i)} \tag{8.75}$$

$$= \left[\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{K}}_{\mathbf{y}}\right]_{(i,j)} \tag{8.76}$$

Knowing these values, $\tilde{\mathbf{R}}_{\mathbf{y}}$ can be built as the sum

$$\tilde{\mathbf{R}}_{\mathbf{y}} = \frac{1}{\Delta x^2}\left(-4\tilde{\mathbf{U}}\tilde{\mathbf{K}}_{\mathbf{y}} + \sum_{(\Delta n,\Delta m)}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{U}}\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right)\left(\tilde{\mathbf{K}}_{\mathbf{y}} + \frac{1}{4}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{K}}_{\mathbf{y}} - \tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\tilde{\mathbf{K}}_{\mathbf{y}}\right)\right)\right) \tag{8.77}$$

$$= \frac{1}{\Delta x^2}\left(-4\tilde{\mathbf{U}} + \sum_{(\Delta n,\Delta m)}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{U}}\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right)\left(\mathbf{I} + \frac{1}{4}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)} - \tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right)\right)\right)\tilde{\mathbf{K}}_{\mathbf{y}} \tag{8.78}$$

Thus, the target matrix $\mathbf{R}_{\mathbf{y}}$ can then be built with a procedure similar to Equations 8.44 through 8.49 and Figure 8.5, ultimately resulting in the formula:

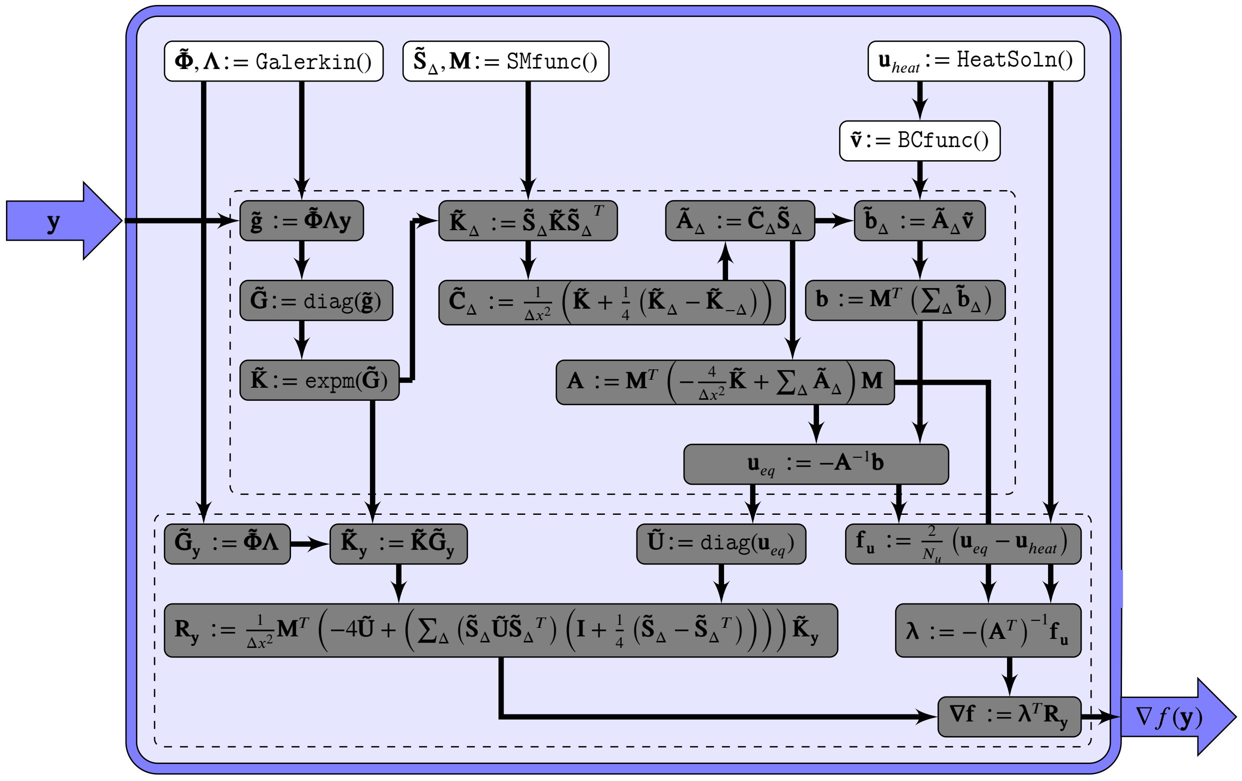
$$\mathbf{R}_{\mathbf{y}} = \mathbf{M}^T\tilde{\mathbf{R}}_{\mathbf{y}} \tag{8.79}$$

$$= \frac{1}{\Delta x^2}\mathbf{M}^T\left(-4\tilde{\mathbf{U}} + \sum_{(\Delta n,\Delta m)}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}\tilde{\mathbf{U}}\tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right)\left(\mathbf{I} + \frac{1}{4}\left(\tilde{\mathbf{S}}_{(\Delta n,\Delta m)} - \tilde{\mathbf{S}}_{(\Delta n,\Delta m)}{}^T\right)\right)\right)\tilde{\mathbf{K}}_{\mathbf{y}} \tag{8.80}$$

Note that every matrix in Equation 8.80 except $\tilde{\mathbf{K}}_{\mathbf{y}}$ is sparse with a single (off-)diagonal; only a single dense $N_d \times N_y$ multiplication is required. Add this to the equally-sized operations used to find $\tilde{\mathbf{g}}$ (Equation 8.65) and $\tilde{\mathbf{K}}_{\mathbf{y}}$

(Equation 8.68), and the entire adjoint method – excluding the forward sweep used to find $\mathbf{u}_{eq}$ – requires three $N_d \times N_y$ dense matrix multiplications and one linear solution of a $N_u \times N_u$ but highly sparse system. For values of N_y used in practice[5] (on the order of $N_y \sim 10^1 - 10^3$, compared to $N_u \sim 128^2 - 256^2 \sim 10^4 - 10^5$) the cost of the solve is several times that of the matrix creation.

The data-flow of the adjoint code is shown in Figure 8.8.



$\tilde{\mathbf{\Phi}}, \mathbf{\Lambda} := \texttt{Galerkin()}$
$\tilde{\mathbf{S}}_\Delta, \mathbf{M} := \texttt{SMfunc()}$
$\mathbf{u}_{heat} := \texttt{HeatSoln()}$
$\tilde{\mathbf{v}} := \texttt{BCfunc()}$
$\mathbf{y}$
$\tilde{\mathbf{g}} := \tilde{\mathbf{\Phi}}\mathbf{\Lambda}\mathbf{y}$
$\tilde{\mathbf{K}}_\Delta := \tilde{\mathbf{S}}_\Delta \tilde{\mathbf{K}} \tilde{\mathbf{S}}_\Delta{}^T$
$\tilde{\mathbf{A}}_\Delta := \tilde{\mathbf{C}}_\Delta \tilde{\mathbf{S}}_\Delta$
$\tilde{\mathbf{b}}_\Delta := \tilde{\mathbf{A}}_\Delta \tilde{\mathbf{v}}$
$\tilde{\mathbf{G}} := \texttt{diag}(\tilde{\mathbf{g}})$
$\tilde{\mathbf{C}}_\Delta := \frac{1}{\Delta x^2}\left(\tilde{\mathbf{K}} + \frac{1}{4}\left(\tilde{\mathbf{K}}_\Delta - \tilde{\mathbf{K}}_{-\Delta}\right)\right)$
$\mathbf{b} := \mathbf{M}^T\left(\sum_\Delta \tilde{\mathbf{b}}_\Delta\right)$
$\tilde{\mathbf{K}} := \texttt{expm}(\tilde{\mathbf{G}})$
$\mathbf{A} := \mathbf{M}^T\left(-\frac{4}{\Delta x^2}\tilde{\mathbf{K}} + \sum_\Delta \tilde{\mathbf{A}}_\Delta\right)\mathbf{M}$
$\mathbf{u}_{eq} := -\mathbf{A}^{-1}\mathbf{b}$
$\tilde{\mathbf{G}}_\mathbf{y} := \tilde{\mathbf{\Phi}}\mathbf{\Lambda}$
$\tilde{\mathbf{K}}_\mathbf{y} := \tilde{\mathbf{K}}\tilde{\mathbf{G}}_\mathbf{y}$
$\tilde{\mathbf{U}} := \texttt{diag}(\mathbf{u}_{eq})$
$\mathbf{f}_\mathbf{u} := \frac{2}{N_u}\left(\mathbf{u}_{eq} - \mathbf{u}_{heat}\right)$
$\mathbf{R}_\mathbf{y} := \frac{1}{\Delta x^2}\mathbf{M}^T\left(-4\tilde{\mathbf{U}} + \left(\sum_\Delta \left(\tilde{\mathbf{S}}_\Delta \tilde{\mathbf{U}} \tilde{\mathbf{S}}_\Delta{}^T\right)\left(\mathbf{I} + \frac{1}{4}\left(\tilde{\mathbf{S}}_\Delta - \tilde{\mathbf{S}}_\Delta{}^T\right)\right)\right)\right)\tilde{\mathbf{K}}_\mathbf{y}$
$\boldsymbol{\lambda} := -\left(\mathbf{A}^T\right)^{-1}\mathbf{f}_\mathbf{u}$
$\nabla \mathbf{f} := \boldsymbol{\lambda}^T \mathbf{R}_\mathbf{y}$
$\nabla f(\mathbf{y})$

Figure 8.8: Data-flow of the diffusion adjoint problem. White functions represent values that remain constant between function runs, dark gray represents code that calls inaccessible matrix functions, and blue represents the program API. Every operation on a variable with a Δ subscript is applied to each of $\Delta = (\Delta n, \Delta m) \in \{(1,0), (-1,0), (0,1), (0,-1)\}$. Functions and data inside the first dotted box are from the forward sweep of the problem; those inside the second box are from new adjoint code.

[5] Recall that when $\tilde{\mathbf{\Phi}}$ and $\tilde{\mathbf{g}}$ were built they used a finite element basis with $N_b = N_u$ elements, which placed an absolute upper limit of $N_y \le N_u$ on the number of computable eigenfunctions.

8.2 Experiments in Nilpotent and Complex Differentiation

Nilpotent Matrix Differentiation was implemented in MATLAB and applied to both versions of the diffusion program. In the version with many scalar operations, a specially-coded, vectorized and naively inductive AD technique (see Section 3.3.2) was used for every operation outside the solve. This was coded by hand because arbitrary-order AD methods are not available for MATLAB, and took the form it did because an efficient implementation would be a very involved task beyond the scope of this work. A high-level schematic of the modified process is shown in Figure 8.9; as this version was intended solely for verification of the second, matrix-only version, the precise details are not relevant. It is only important to know that the calculated values of the two versions matched exactly, to within the precision of the solver. Together with finite difference approximations and comparisons to known deterministic values (see Section 8.3), it is safe to say that the matrix program correctly implements the model in all relevant ways.

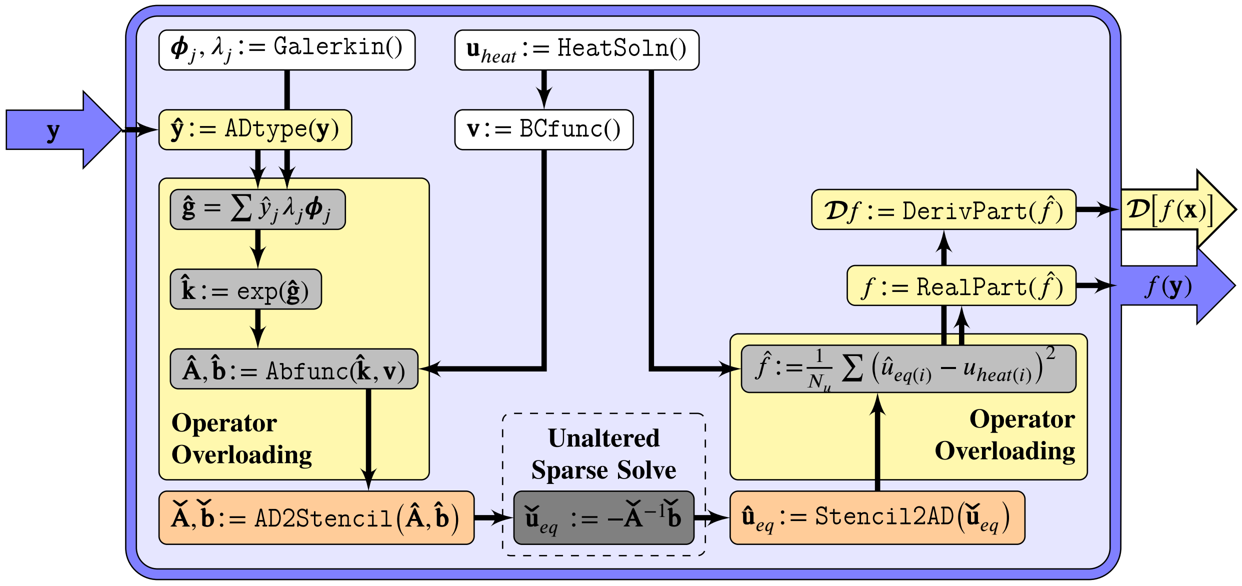
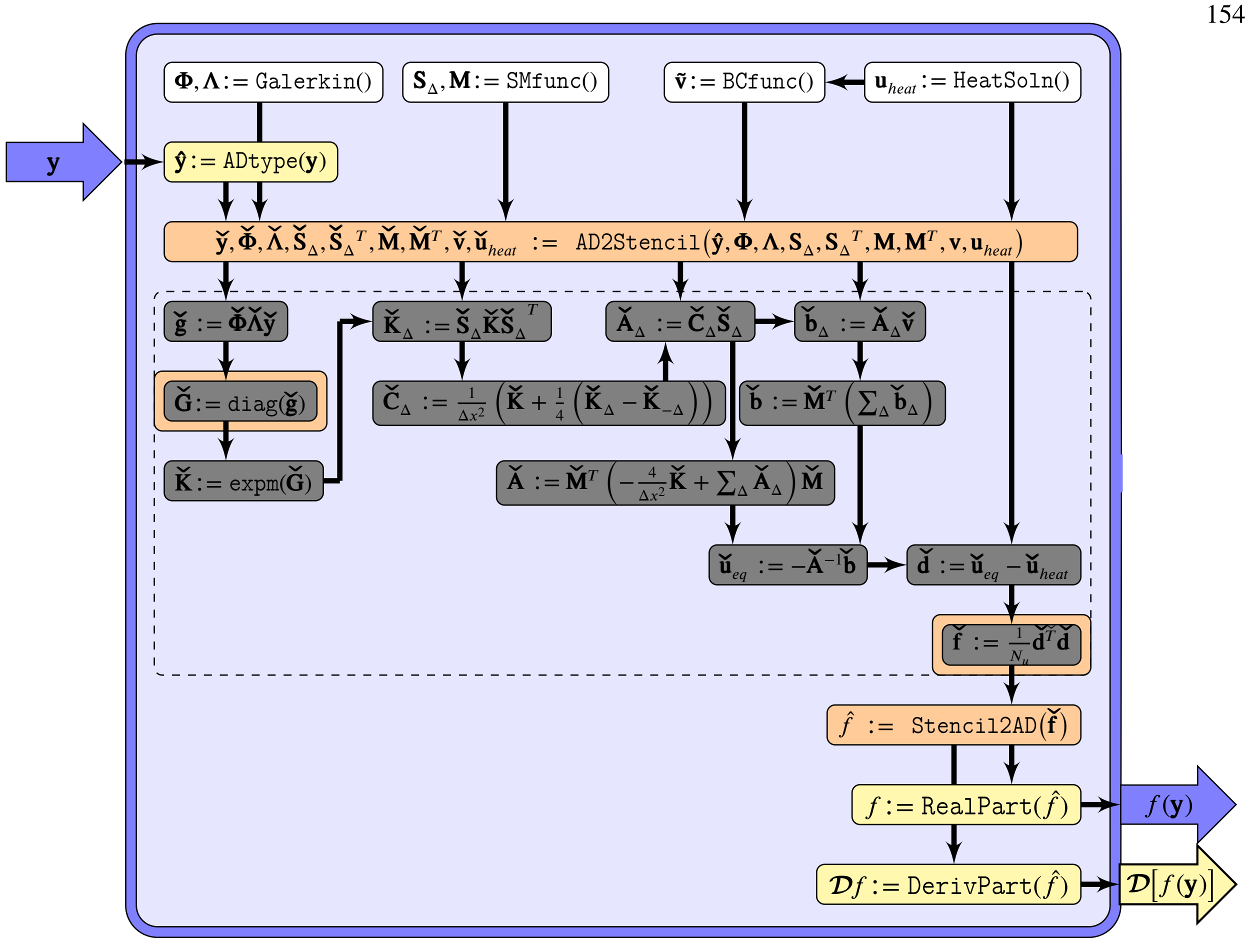

$\boldsymbol{\phi}_j, \lambda_j :=$ Galerkin()
$\mathbf{u}_{heat} :=$ HeatSoln()
$\mathbf{y}$
$\hat{\mathbf{y}} :=$ ADtype($\mathbf{y}$)
$\mathbf{v} :=$ BCfunc()
$\hat{\mathbf{g}} = \sum \hat{y}_j \lambda_j \boldsymbol{\phi}_j$
$\mathcal{D}f :=$ DerivPart($\hat{f}$)
$\mathcal{D}[f(\mathbf{x})]$
$\hat{\mathbf{k}} :=$ exp($\hat{\mathbf{g}}$)
$f :=$ RealPart($\hat{f}$)
$f(\mathbf{y})$
$\hat{\mathbf{A}}, \hat{\mathbf{b}} :=$ Abfunc($\hat{\mathbf{k}}, \mathbf{v}$)
$\hat{f} := \frac{1}{N_u} \sum \left(\hat{u}_{eq(i)} - u_{heat(i)}\right)^2$
Operator Overloading
Unaltered Sparse Solve
Operator Overloading
$\check{\mathbf{A}}, \check{\mathbf{b}} :=$ AD2Stencil($\hat{\mathbf{A}}, \hat{\mathbf{b}}$)
$\check{\mathbf{u}}_{eq} := -\check{\mathbf{A}}^{-1}\check{\mathbf{b}}$
$\hat{\mathbf{u}}_{eq} :=$ Stencil2AD($\check{\mathbf{u}}_{eq}$)

Figure 8.9: Data-flow of the first, scalar-focused implementation of the diffusion problem, after AD and NMD are applied. (See caption under Figure 8.5.) Yellow represents new or changed code for scalar AD; orange represents new NMD code. The majority of the code (but not run-time cost) is the responsibility of scalar AD; NMD only differentiates the solve. Both the output and the derivatives of this code match the matrix-focused implementation, verifying those results.

The summary schematic for the NMD-modified matrix program is shown in Figure 8.10. As can be seen there, the majority of the code modification comes at program initialization and immediately before program output. To use the same $\mathrm{AD2Stencil}()$ and $\mathrm{Stencil2AD}()$ functions as the scalar program (noting that most real-world applications will have at least *some* scalar code, and thus make some use of a standard AD method), inputs are turned into an operator-overloaded AD type purely for initialization, then immediately converted to a stencil column vector $\check{\mathbf{y}}$; likewise, the final output f is unpacked into an AD variable purely as a convenient container, which then presents resulting data in the desired format. Considering that no intermediate derivatives are needed, this approach is not the only way – one could alternatively use a hypothetical $\mathrm{InitializeStencil}()$ function to set relevant stencil entries to one, without ever invoking a scalar AD tool. Indeed, the conversion of the deterministic matrices (white blocks in the figure) into NMD stencils requires no derivative information, and thus follows the same initialization procedure for real-valued and AD-overloaded types alike. Likewise, one could use a $\mathrm{UnpackStencil}()$ function to organize the final derivative information into whatever format desired.



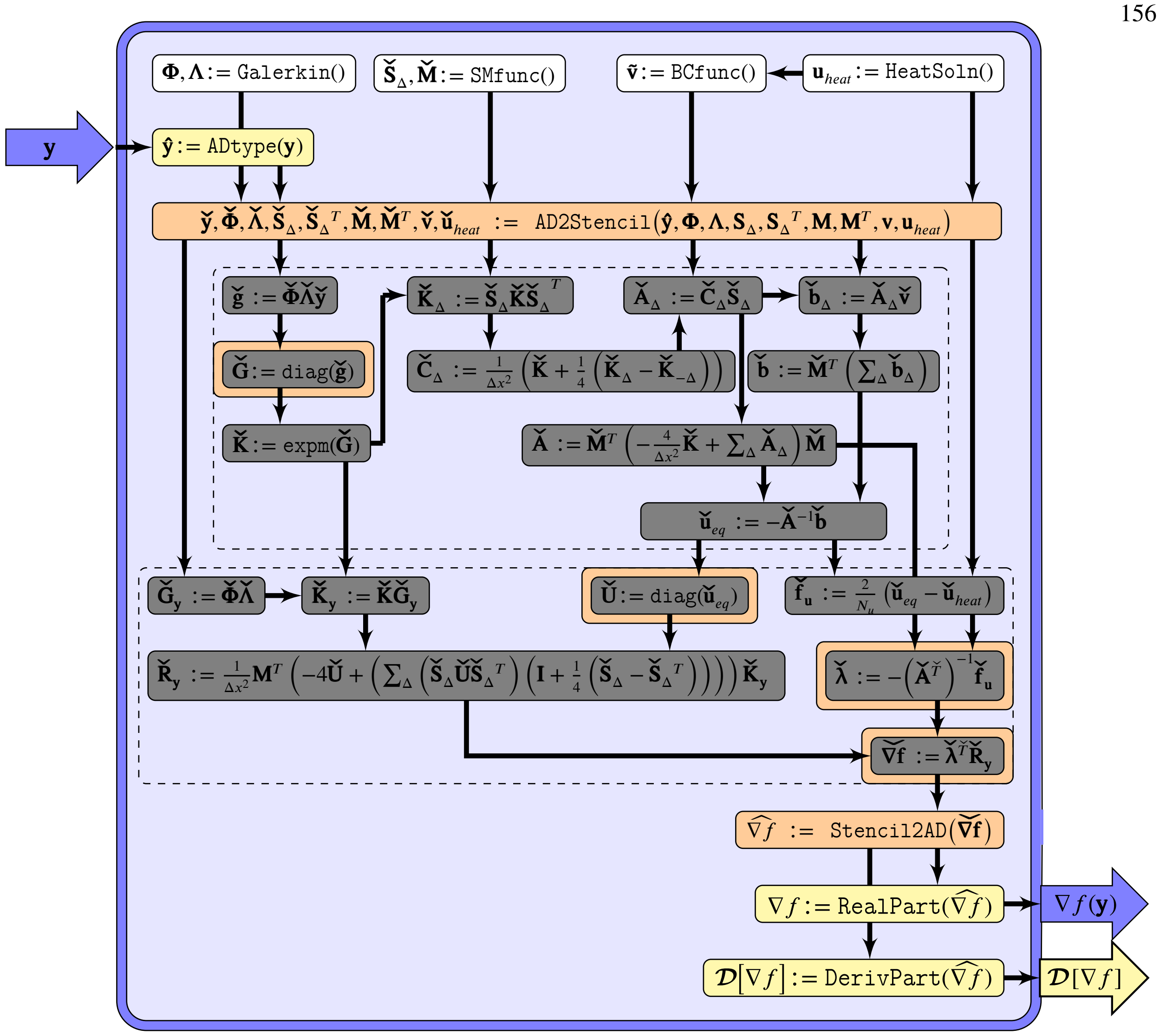
$\mathbf{\Phi},\mathbf{\Lambda} :=$ Galerkin()
$\mathbf{S}_\Delta,\mathbf{M} :=$ SMfunc()
$\check{\mathbf{v}} :=$ BCfunc()
$\mathbf{u}_{heat} :=$ HeatSoln()
$\mathbf{y}$
$\hat{\mathbf{y}} :=$ ADtype($\mathbf{y}$)
$\check{\mathbf{y}},\check{\mathbf{\Phi}},\check{\mathbf{\Lambda}},\check{\mathbf{S}}_\Delta,\check{\mathbf{S}}_\Delta{}^T,\check{\mathbf{M}},\check{\mathbf{M}}^T,\check{\mathbf{v}},\check{\mathbf{u}}_{heat} :=$ AD2Stencil$(\hat{\mathbf{y}},\mathbf{\Phi},\mathbf{\Lambda},\mathbf{S}_\Delta,\mathbf{S}_\Delta{}^T,\mathbf{M},\mathbf{M}^T,\mathbf{v},\mathbf{u}_{heat})$
$\check{\mathbf{g}} := \check{\mathbf{\Phi}}\check{\mathbf{\Lambda}}\check{\mathbf{y}}$
$\check{\mathbf{K}}_\Delta := \check{\mathbf{S}}_\Delta\check{\mathbf{K}}\check{\mathbf{S}}_\Delta{}^T$
$\check{\mathbf{A}}_\Delta := \check{\mathbf{C}}_\Delta\check{\mathbf{S}}_\Delta$
$\check{\mathbf{b}}_\Delta := \check{\mathbf{A}}_\Delta\check{\mathbf{v}}$
$\check{\mathbf{G}} :=$ diag($\check{\mathbf{g}}$)
$\check{\mathbf{C}}_\Delta := \frac{1}{\Delta x^2}\left(\check{\mathbf{K}} + \frac{1}{4}\left(\check{\mathbf{K}}_\Delta - \check{\mathbf{K}}_{-\Delta}\right)\right)$
$\check{\mathbf{b}} := \check{\mathbf{M}}^T\left(\sum_\Delta \check{\mathbf{b}}_\Delta\right)$
$\check{\mathbf{K}} :=$ expm($\check{\mathbf{G}}$)
$\check{\mathbf{A}} := \check{\mathbf{M}}^T\left(-\frac{4}{\Delta x^2}\check{\mathbf{K}} + \sum_\Delta \check{\mathbf{A}}_\Delta\right)\check{\mathbf{M}}$
$\check{\mathbf{u}}_{eq} := -\check{\mathbf{A}}^{-1}\check{\mathbf{b}}$
$\check{\mathbf{d}} := \check{\mathbf{u}}_{eq} - \check{\mathbf{u}}_{heat}$
$\check{\mathbf{f}} := \frac{1}{N_u}\check{\mathbf{d}}^T\check{\mathbf{d}}$
$\hat{f} :=$ Stencil2AD($\check{\mathbf{f}}$)
$f :=$ RealPart($\hat{f}$)
$f(\mathbf{y})$
$\mathcal{D}f :=$ DerivPart($\hat{f}$)
$\mathcal{D}[f(\mathbf{y})]$

Figure 8.10: Data-flow of the second, matrix-only implementation of the diffusion problem after NMD has been applied. (See caption under Figure 8.5.) Yellow represents new or changed code for scalar AD; orange represents new NMD code. Almost the entire original program (dashed box) uses unmodified matrix code, with the exception of the diagonalize and dot-product functions (highlighted in orange), which require custom NMD variants.

As shown in the AD2Stencil() block, the matrix transposes $\mathbf{S}_\Delta{}^T$ and $\mathbf{M}^T$ are computed before stencil conversion, just as was mentioned in Section 6.2; this is not strictly necessary for these purely real-valued (undifferentiated) matrices, as blockwise transpose operations are only called for when a stencil is carrying derivative information. In fact, to make the entire code compatible with NMD, only two blockwise function variants are needed: one to diagonalize $\check{\mathbf{g}}$ (which uses a vector-stencil) into $\check{\mathbf{G}}$ (which needs a matrix-stencil), and one to compute the dot product of the deviation vector $\check{\mathbf{d}}$ with itself. Both functions are straightforward and only need to be implemented once, after which they can be applied to any other program. The blockwise diagonalization simply locates each derivative value in $\check{\mathbf{g}}$ and places it in the appropriate location(s) in a new block-diagonal matrix $\check{\mathbf{G}}$. The dot product function must unpack vector-stencils into two matrices – one of many matrix-stencils laid out side by side (a "block row vector") and one of many stacked on top of each other (a “block column vector”). After this, standard matrix multiplication yields a single stencil, meaning that the program result $\breve{\mathbf{f}}$ has been converted from a scalar to a matrix before hand-off to NMD finalization.

Because the program was designed from the start to use the matrix exponential $\mathrm{expm}()$ function, no modification is required there – NMD handles the power series implementation of the function just as well as it does every other operation. If the original program had applied a standard scalar exponential $\exp()$ to each entry in $\mathbf{g}$, NMD could either temporarily unpack the augmented $\breve{\mathbf{g}}$ into a scalar AD vector $\hat{\mathbf{g}}$, or else simply use $\breve{\mathbf{G}}$ on a purely temporary basis before returning to vector form. Note, though, the similarity to the truncated polynomial evaluation of $\exp()$ from Section 4.5: just as a polynomial is represented as a real plus many powers of t, NMD uses a real (identity) basis stencil plus many nilpotent basis stencils. In theory, a NMD-specific variant of expm could thus be developed that stripped out derivative data in order to evaluate the real values of a matrix exponential in isolation, before evaluating a much shorter power series for the derivative data. However, the cost of separating the data would almost certainly outweigh any savings, let alone the extra implementation burden on the user.

Finally, NMD was also applied to the adjoint code; the schematic is shown in Figure 8.11. Note there that this application requires three more custom function calls: another diagonalization for the vector $\breve{\mathbf{u}}$ (which uses the same code used to diagonalize $\breve{\mathbf{g}}$ in the forward problem), a block-transpose of $\breve{\mathbf{A}}$ before solving the adjoint equation, and a conversion of $\breve{\boldsymbol{\lambda}}$ into a block row vector before premultiplying with $\breve{\mathbf{R}}_{\mathbf{y}}$. Note that this last variant means $\widetilde{\nabla \mathbf{f}}$ is itself in block row form; if it were to be passed to other matrix operations, it would first be converted back to a real vector, but as is, it can be passed directly into $\mathrm{Stencil2AD}()$. It is true that a switch from vector-matrix to matrix-matrix multiplication requires a change of code (in BLAS, at least – the MATLAB prototype did not); if this were truly undesirable, a block-transpose could be used to find $\widetilde{\nabla \mathbf{f}}^{\breve{T}} = \left(\breve{\mathbf{R}}_{\mathbf{y}}{}^{\breve{T}} \breve{\boldsymbol{\lambda}}\right)$, which would use a single matrix-vector multiplication and require no conversion to or from block row form.



$\boldsymbol{\Phi},\boldsymbol{\Lambda} :=$ Galerkin()
$\breve{\mathbf{S}}_\Delta, \breve{\mathbf{M}} :=$ SMfunc()
$\breve{\mathbf{v}} :=$ BCfunc()
$\mathbf{u}_{heat} :=$ HeatSoln()
$\mathbf{y}$
$\hat{\mathbf{y}} :=$ ADtype($\mathbf{y}$)
$\breve{\mathbf{y}}, \breve{\boldsymbol{\Phi}}, \breve{\boldsymbol{\Lambda}}, \breve{\mathbf{S}}_\Delta, \breve{\mathbf{S}}_\Delta{}^T, \breve{\mathbf{M}}, \breve{\mathbf{M}}^T, \breve{\mathbf{v}}, \breve{\mathbf{u}}_{heat} :=$ AD2Stencil$(\hat{\mathbf{y}}, \boldsymbol{\Phi}, \boldsymbol{\Lambda}, \mathbf{S}_\Delta, \mathbf{S}_\Delta{}^T, \mathbf{M}, \mathbf{M}^T, \mathbf{v}, \mathbf{u}_{heat})$
$\breve{\mathbf{g}} := \breve{\boldsymbol{\Phi}}\breve{\boldsymbol{\Lambda}}\breve{\mathbf{y}}$
$\breve{\mathbf{K}}_\Delta := \breve{\mathbf{S}}_\Delta \breve{\mathbf{K}} \breve{\mathbf{S}}_\Delta{}^T$
$\breve{\mathbf{A}}_\Delta := \breve{\mathbf{C}}_\Delta \breve{\mathbf{S}}_\Delta$
$\breve{\mathbf{b}}_\Delta := \breve{\mathbf{A}}_\Delta \breve{\mathbf{v}}$
$\breve{\mathbf{G}} :=$ diag$(\breve{\mathbf{g}})$
$\breve{\mathbf{C}}_\Delta := \frac{1}{\Delta x^2}\left(\breve{\mathbf{K}} + \frac{1}{4}\left(\breve{\mathbf{K}}_\Delta - \breve{\mathbf{K}}_{-\Delta}\right)\right)$
$\breve{\mathbf{b}} := \breve{\mathbf{M}}^T\left(\sum_\Delta \breve{\mathbf{b}}_\Delta\right)$
$\breve{\mathbf{K}} :=$ expm$(\breve{\mathbf{G}})$
$\breve{\mathbf{A}} := \breve{\mathbf{M}}^T\left(-\frac{4}{\Delta x^2}\breve{\mathbf{K}} + \sum_\Delta \breve{\mathbf{A}}_\Delta\right)\breve{\mathbf{M}}$
$\breve{\mathbf{u}}_{eq} := -\breve{\mathbf{A}}^{-1}\breve{\mathbf{b}}$
$\breve{\mathbf{G}}_\mathbf{y} := \breve{\boldsymbol{\Phi}}\breve{\boldsymbol{\Lambda}}$
$\breve{\mathbf{K}}_\mathbf{y} := \breve{\mathbf{K}}\breve{\mathbf{G}}_\mathbf{y}$
$\breve{\mathbf{U}} :=$ diag$(\breve{\mathbf{u}}_{eq})$
$\breve{\mathbf{f}}_\mathbf{u} := \frac{2}{N_u}\left(\breve{\mathbf{u}}_{eq} - \breve{\mathbf{u}}_{heat}\right)$
$\breve{\mathbf{R}}_\mathbf{y} := \frac{1}{\Delta x^2}\mathbf{M}^T\left(-4\breve{\mathbf{U}} + \left(\sum_\Delta\left(\breve{\mathbf{S}}_\Delta \breve{\mathbf{U}} \breve{\mathbf{S}}_\Delta{}^T\right)\left(\mathbf{I} + \frac{1}{4}\left(\breve{\mathbf{S}}_\Delta - \breve{\mathbf{S}}_\Delta{}^T\right)\right)\right)\right)\breve{\mathbf{K}}_\mathbf{y}$
$\breve{\boldsymbol{\lambda}} := -\left(\breve{\mathbf{A}}^{\breve{T}}\right)^{-1}\breve{\mathbf{f}}_\mathbf{u}$
$\breve{\nabla}\mathbf{f} := \breve{\boldsymbol{\lambda}}^{\breve{T}}\breve{\mathbf{R}}_\mathbf{y}$
$\widehat{\nabla f} :=$ Stencil2AD$(\breve{\nabla}\mathbf{f})$
$\nabla f :=$ RealPart$(\widehat{\nabla f})$
$\nabla f(\mathbf{y})$
$\mathcal{D}[\nabla f] :=$ DerivPart$(\widehat{\nabla f})$
$\mathcal{D}[\nabla f]$

Figure 8.11: Data-flow of the diffusion adjoint problem after NMD has been applied. (See caption under Figure 8.8.) Yellow represents new or changed code for scalar AD; orange represents new NMD code. Almost every user function can be used as-is; the exceptions (highlighted in orange) include the diagonalization of $\mathbf{g}$ and $\mathbf{u}$, the transpose of $\mathbf{A}$, and the transposed row-multiply of $\boldsymbol{\lambda}^T\mathbf{R}_\mathbf{y}$, all of which require custom NMD variants as discussed in the text.

For the forward problem (including the deterministic variant of the matrix-only implementation), NMD was used both in its simplest form, that of a dual-number stencil, and with a stencil capable of finding all entries of a 2×2 Hessian and the corresponding 2-variable gradient:

$$\mathcal{DM}_{\mathbf{y}} \equiv \begin{bmatrix} 1 & 0 & 0 & 2{}^{\partial}/_{\partial y_2} & 0 & {}^{\partial^2}/_{\partial y_2{}^2} \\ 0 & 1 & 0 & {}^{\partial}/_{\partial y_1} & {}^{\partial}/_{\partial y_2} & {}^{\partial^2}/_{\partial y_1 \partial y_2} \\ 0 & 0 & 1 & 0 & 2{}^{\partial}/_{\partial y_1} & {}^{\partial^2}/_{\partial y_1{}^2} \\ 0 & 0 & 0 & 1 & 0 & {}^{\partial}/_{\partial y_2} \\ 0 & 0 & 0 & 0 & 1 & {}^{\partial}/_{\partial y_1} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.81}$$

To compare additional approaches, first and second derivatives of the forward codes were also found with centered finite differences and a single first derivative was found with CD; both were run with a great many step-sizes, ranging from 10^6 all the way down to 10^{-18} (below machine epsilon). Likewise, NMD, FD and CD were all applied to the adjoint code to find a Hessian-vector product in a random direction $\mathbf{v}$; here, NMD was used both with a single-variable dual stencil and a two-variable first-order stencil to find two products at once:

$$\mathcal{DM}_{\mathbf{y}} \equiv \begin{bmatrix} 1 & 0 & {}^{\partial}/_{\partial y_2} \\ 0 & 1 & {}^{\partial}/_{\partial y_1} \\ 0 & 0 & 1 \end{bmatrix} \tag{8.82}$$

The same stencil was also used in cardinal directions to find the first two columns of the Hessian, thereby providing second derivative values to compare against forward NMD and FD for the stochastic problem. First derivative values of the stochastic problem were compared against the standard adjoint method (even though the real value of NMD produced the same results). All derivatives of the deterministic problem, of course, were compared to the analytically-known values. All accuracy measurements were averaged across ten runs and used $N_y = 100$ eigenfunctions.

In addition to measuring the accuracy of each method, the wall-clock execution time of every major program block was measured across many runs, as was the memory requirement of every major matrix. As the matrices $\boldsymbol{\Phi}$, $\tilde{\mathbf{K}}_{\mathbf{y}}$, and $\mathbf{R}_{\mathbf{y}}$ were all dense in the undifferentiated problem, they were not converted to sparse format (despite NMD stencils themselves being sparse) as this would not constitute a fair test for a method designed to have little implementation effort. All remaining matrices, including $\tilde{\mathbf{S}}$, $\mathbf{M}$, $\tilde{\mathbf{U}}$, $\tilde{\mathbf{K}}$, $\tilde{\mathbf{G}}$, and $\mathbf{A}$, were sparse to begin with and remained so under NMD.

Further, although the base code simply used MATLAB's default backslash operator `x = −A\b` to solve the system, on the back-end this operation called MATLAB's built-in UMFPACK sparse solver system; for asymmetric matrices like **A**, this system applied LU decomposition. The resulting sparse **L** and **U** matrices were invisible to the user and to the profiling tool, but available under MATLAB's sparse diagnostic system. Memory requirements for these matrices are thus provided alongside the others.

8.3 Results: Differentiation Accuracy

The derivative methods were first verified in the forward mode by differentiating the deterministic code and comparing the results to the analytically known values for the heat equation. Given the boundary conditions

$$u_{BC}(\mathbf{x}) = \left(m_1 x_1 + b_1\right)\left(m_2 x_2 + b_2\right) \tag{8.83}$$

where $m_1 = b_1 = 500$ and $m_2 = b_2 = 1000$, first and second derivatives of u at the middle point of the mesh were found with respect to m_1 and m_2; for CD, which is limited to a single first derivative, only ${}^{\partial u}/_{\partial m_1}$ was found. Relative root-mean-square errors (RRMSE) of the results are shown in Figure 8.12, plotted against step size for FD and CD; the RRMSE shown for NMD is from a single evaluation. Note that the figure also includes lines for machine epsilon, the solve residual, and the upper bound on solution error given by the condition number[6] (see the caption).

The important take-away from the figure is the fact that the error in the real quantity of interest (QoI, i.e., u at the midpoint) is identical to the NMD and CD error in the first derivatives and to the NMD error in the second derivatives – those methods are, for all intents and purposes, exact. The most striking feature of the data, though, is the behavior of the finite difference method: extremely large step-sizes h – on the order of 10^4 or higher – actually produced results precise to near machine epsilon and better than the solve residual itself. This is likely due to the nature of the chosen BCs and the corresponding analytic solution – each derivative is linearly dependent on at most a single m-value, so that any slope equation is insensitive to step-size $h = \Delta m$. When the step-size is so large as to dominate the other values, the calculation in fact becomes more accurate. The same behavior leads to similarly accurate values for CD, although the properties of that method prevent error from ever setting in for smaller h – first derivatives have the same precision

[6] Importantly, the condition number of **A** did not change under NMD.

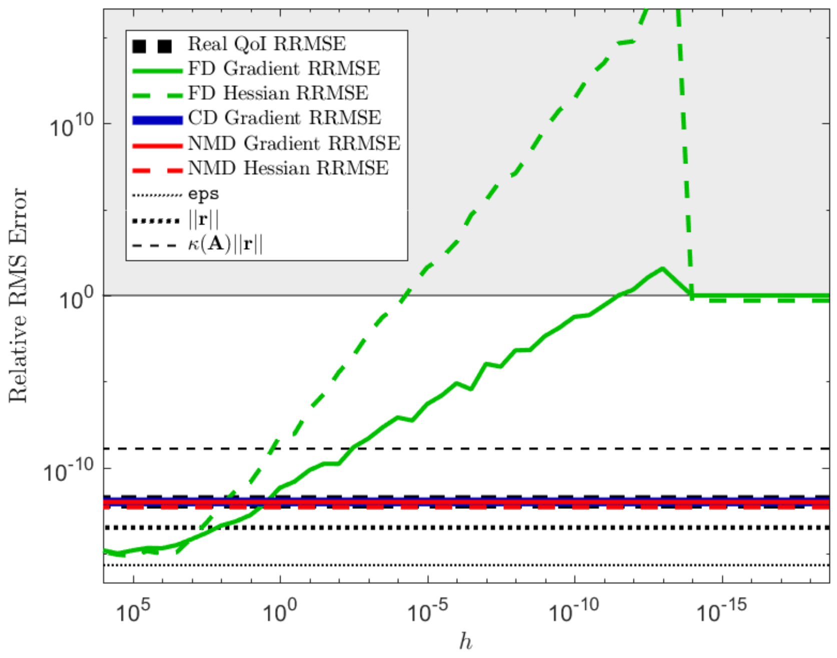


Figure 8.12: Comparison of derivative error vs known values of the deterministic heat equation, plotted against the step-size used for FD and CD. Note that CD and NMD have almost exactly the same error as the real output itself – those lines all overlap. In the case of FD, very large step-sizes produce surprisingly accurate values, due to the unusual circumstances described in the text; however, if the user does not try these abnormal h-values, error can be very high.

For reference, both the relative residual $||\mathbf{r}||$ of the linear solution and the upper bound on solution error given the given condition number $\kappa(\mathbf{A})$ (both found with the l_1 norm) are also provided, along with machine epsilon. These are admittedly imperfect comparisons, as the other values represent RRMSE (relative root-mean-square error) in the derivatives of the quantity of interest, not the linear solution; however, as the system solution is the major bottleneck in the program's accuracy, these values give at least some intuition into the best-case behavior.

for any step-size.

This is most certainly not something one can count on in production codes. More than this, note that if analytic derivatives were not available, most users would not think to try such unintuitively large values of h. In fact, the first guess for a step size is often that which sets the error order of the FD approximation equal to the order of the subtractive cancellation error [61]; for the $\mathcal{O}(h^2)$ centered difference method and (for first derivatives) $\mathcal{O}(\texttt{eps}/h)$ cancellation error, where $\texttt{eps} \approx 10^{-16}$ is machine epsilon, this leads to:

$$\mathcal{O}\left(h^2\right) \sim \mathcal{O}\left(\frac{10^{-16}}{h}\right) \tag{8.84}$$

$$h^3 \sim 10^{-16} \tag{8.85}$$

$$h \sim 5 \times 10^{-6} \tag{8.86}$$

However, as can be seen in the figure, this value of h only yields roughly five significant figures; compare this to the eleven or twelve figures possible with CD or NMD. Meanwhile, the first-guess h for the Hessian would likely be:

$$\mathcal{O}\left(h^2\right) \sim \mathcal{O}\left(\frac{10^{-16}}{h^2}\right) \tag{8.87}$$

$$h^4 \sim 10^{-16} \tag{8.88}$$

$$h \sim 10^{-4} \tag{8.89}$$

which leads to an RRMSE on the order of 10^0 – that is, second-order FD yields no significant figures at all; for the guessed step-size, it is useless for differentiation purposes. For its part, NMD still finds eleven or twelve figures, while CD cannot find second derivatives at all.

Next, for the forward stochastic problem, derivatives of f were found with respect to y_1 and y_2; results are shown in Figure 8.13. As true values are unknown, this plot represents relative difference from the real-valued adjoint method (for first derivatives) and from the NMD-adjoint method (for second derivatives); the close agreement of all values indicates this is a fair comparison. As can be seen, for the stochastic problem forward NMD and CD produce values that differ from the adjoints by roughly the same magnitude. This difference is very slightly more than would be implied by the upper bound imposed by the condition number of $\mathbf{A}$ – note, however, that the condition number reported by MATLAB for sparse matrices uses the l_1 norm[7] , whereas reported errors are the RRMSE of the derivatives of f, and thus cannot be used as a true "apples-to-apples" comparison. Regardless, NMD and CD both provide nine or more

[7] The residual also uses the l_1 norm.

significant figures of accuracy.

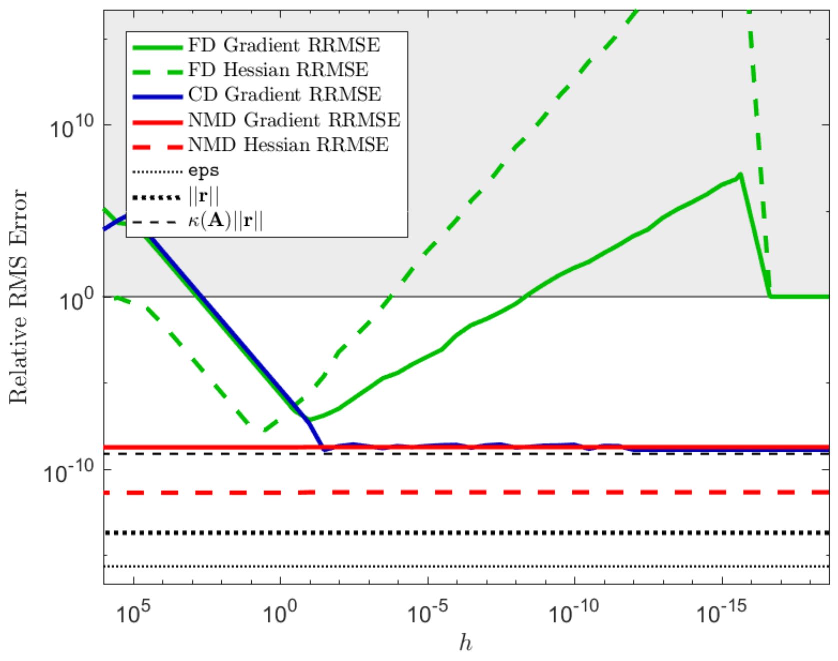


Figure 8.13: Error in stochastic forward-mode methods as compared to adjoint methods. All forward methods are used to find derivatives of the first two y-values. Error in the gradient is as compared to the ordinary adjoint method; error in the Hessian is as compared to a first-order NMD-adjoint hybrid. Other reference values are as described in Figure 8.12.

In this problem, the error behavior for FD and CD is closer to what one might expect, but the ideal h for FD is again surprisingly large – 10^{-1} for the gradient and 10^{1} for the Hessian. (Note that, as each y is a standard normal variable, $h \sim 10$ represents ten standard deviations away from the mean.) In this case, the reason is likely the choice of $f(\mathbf{u})$: because diffusion is by nature a smoothing process, the mean squared deviation is very insensitive to changes in $\mathbf{y}$. Once again, the typical first guesses for h would lead to very inaccurate values – only three significant figures for the gradient and none for the Hessian. Again, without careful parameter searching FD would be useless for calculating second derivatives.

Finally, FD, CD, and NMD were used to find a Hessian-vector product by taking a single first derivative of the adjoint code; results are shown in Figure 8.14. In this case, values were compared to that of NMD, as it is the only method theoretically guaranteed to find the exact derivative; again, close agreement of the results indicates this is a fair comparison. Once again, CD – and, presumably, NMD – found the result to ten or more significant figures, while

FD again had the same anomalous error behavior. The typical first guess of h would yield less than three significant figures.

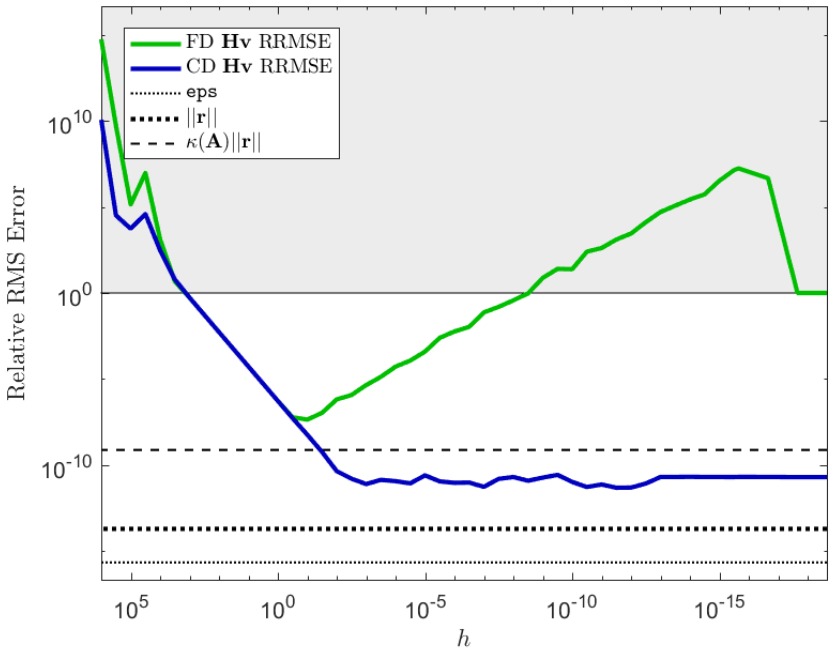


Figure 8.14: Relative difference of stochastic FD and CD Hessian-vector values as compared to NMD. The difference for CD is of the same order as numerical error and as errors from Figure 8.12 and Figure 8.13, implying NMD and CD both have relative errors of the same order.

8.4 Results: Differentiation Costs

Given the experiments described in Section 8.2 and the desired derivatives found in Section 8.3, four variants of both the forward and adjoint problem (i.e., a grand total of eight) were run: a real-valued (i.e., unmodified) variant for finite differencing; a complex-valued variant for CD; and the two stencil-expanded variants for NMD – as described in Section 8.2, a 2×2 and a 6×6 stencil for the forward mode and a 2×2 and a 3×3 stencil for the adjoint. As the diffusion problem has a wide array of input parameters, a combinatoric number of benchmarks are possible. However, the two major factors in the time and memory cost of the underlying program are the number of mesh nodes N_u (i.e., the model fidelity) and the number of eigenfunctions N_y (i.e., the number of inputs). Therefore, differentiation methods were tested against a wide array of both values – first, by holding $N_y = 100$ fixed and varying the number of nodes from $N_u = 16^2 = 256$ up to a maximum of $N_u = 256^2 = 65536$, then by holding N_u fixed at that maximum and varying N_y from $2^2 = 4$ up to $20^2 = 400$. At each parameter setting, all major program steps were timed and the memory requirements of all large matrices were recorded. Plots of these time and memory costs, including detailed captions, can be found in Appendix A.

In terms of the cost versus the number of eigenfunctions N_y (plotted and discussed case by case in Section A.1), two important trends stand out: first, the costs of both the forward code and the adjoint were linearly dependent on this quantity, i.e., $\mathcal{O}(N_y)$; second, the costs of both CD and NMD were a constant multiple of the code to which they were applied, i.e., the cost of applying either method to a given code was $\mathcal{O}(1)$. The first observation is unfortunate, especially for the adjoint code, which one would hope to be independent of the number of inputs; it is due to the fact that the cost of sampling the diffusivity field or building the Jacobians thereof will always be linearly dependent on the number of eigenfunctions or on the number of independent differentiation variables. Note, though, that this was a very large N_y – for comparison, the other published work [47] that built an adjoint of this problem used only $N_y = 30$. For large meshes and N_y in that range, the cost of sampling or building the Jacobians would be lost in the noise, leaving an effectively $\mathcal{O}(1)$ problem. The same can be said of any adjoint method where the cost of the solve significantly outweighs the cost of problem setup.

The second observation is much more pertinent to this work and much more promising: in the both the forward and adjoint modes, the additional cost of applying either CD or NMD was entirely independent of N_y. This was expected for the forward mode, but the adjoint result is crucial: assuming a truly $\mathcal{O}(1)$ gradient method, effectively exact Hessian-vector products can likewise be found via either approach in $\mathcal{O}(1)$ time, independent of the number of inputs N_y; if desired, the full Hessian can thus be found in $\mathcal{O}(N_y)$.

Comparisons across multiple mesh sizes N_u (see Section A.2) – that is, across varying problem difficulties – were more concerning: while the cost of applying almost all functions was independent of N_u, the major exception was the all-important solve. Specifically, the time costs of sparse system solution using NMD grew faster than the underlying problem. After some investigation, this was traced to the LU factorization performed automatically by MATLAB, where memory costs were also shown to grow disproportionately. As alluded to in Section 8.2, when the user invokes the backslash operator `x = -A\b` on a matrix $\mathbf{A}$, MATLAB executes a decision tree to determine how best to solve the system. In the case of asymmetric matrices (like those for both the real- and nilpotent-valued diffusion problem), it builds the lower and upper triangular matrices **L** and **U** via its built-in implementation of the UMFPACK (Unsymmetric MultiFrontal PACKage) [122] library.

As stated in Chapter 6, any operation that respects a stencil's hypercomplex isomorphism will propagate derivatives. The process of fully inverting a matrix, being the inverse of the very multiplication operation that stencils are built to represent, is one such operation – and thus, so is any system solver that only finds a vector without building the full matrix inverse. However, as stated in Section 6.2, algorithms that operate on a single row or column at a time are almost certain to destroy the stencil structure, and thus the isomorphism. LU factorization is one such algorithm.

The process of LU factorization, then, presents a paradox: even though it is used as an intermediate step before system solution, it operates column by column and cannot be expected to respect NMD's block structure. In practice, though, the algorithm can reach the correct result – it simply takes a much longer route to find it. In the process of building the sparse factors, it generates more non-zero scalar values than should be expected from the sparsity pattern of a given stencil; in fact, it can require more than twice the memory one would get even from dense fill-in of the stencil (see Figure A.11). The time required for factorization (which in fact dominates the cost of the backslash operator) of course grows proportionately to the size of **L** and **U**, as does the time required for the solve itself.

Although unfortunate, this behavior is not catastrophic for practical usage of NMD. Note that throughout Section A.2, the costs of the solve and factorization form a straight line on a semilog plot; this implies logarithmic – i.e., sublinear – growth versus N_u, which might considered quite respectable in any other context.

Before closing this section and with it all discussion of the diffusion proof of concept, note that the appendix does not plot the NMD custom variants of the transpose, diagonalize, and dot-product functions, nor does it plot the cost to initialize NMD matrices. The is because the cost of these operations is highly implementation-specific, while the test code is only a very rough prototype. It is sufficient to know that the relative cost of each custom variant was constant versus N_y, while the absolute[8] cost of initial stencil construction was linear in N_y (i.e., just like the cost of sampling the field – if the cost of using the eigenfunctions was constant, NMD initialization would be, as well); likewise, each function was linear with respect to the number of non-zero values (and thus with respect to N_u) on which it operated.

[8] The cost of stencil initialization cannot be expressed as a relative ratio, as there is nothing to compare it against.

Chapter 9

Conclusions

This work has shown that hypercomplex algebras with nilpotent elements can be used as an alternate analytic framework to discuss high-order differentiation of computer code. For purely scalar functions – that is, those that operate on individual real- or complex-valued numbers – nilpotent (and most other non-real) elements have only limited real-world use as a practical tool, as existing codes must be modified to compute the non-real values; for matrix functions, however, new options become available. Given the new analytic framework, isomorphisms can be constructed to represent values from any hypercomplex algebra as purely real-valued matrices that store values in very specific patterns, termed "stencils." If a stencil is used to blockwise replace each element of another, pre-existing matrix, the resulting structure can differentiate functions in identical fashion to numbers from the scalar hypercomplex algebra – i.e., the modified matrix will carry derivatives of the original matrix, despite using only real numbers and only calling unaltered code meant for the original matrix. Once imbued with this information, functions of the new matrix or matrices will compute the chain rule with regard to most operations, including basic arithmetic, power series, and system solvers. Many other operations such as transposition, diagonalization, or the dot product can easily be replaced with variants that likewise propagate derivatives. All such computed values are exact to within the precision of the underlying matrix routines themselves.

The concept of derivative stencils, combined with the analytic framework used to derive and construct them, form a new method called Nilpotent Matrix Differentiation (NMD). Use of this method allows developers of existing linear-algebra-intensive codes to find derivatives of long sequences of matrix operations after only modifying the initial matrices of each sequence; alternatively, even differentiation of short sequences can be made easier when the number or order of derivatives is expected to change between program runs – beyond the trivial redefinition of the problem's stencil, no additional implementation is required to effect such a change. This ability to find derivatives with comparatively little human programming effort makes NMD an automatic differentiation (AD) method, with its own niche among the broad array of existing AD tools.

One particularly useful application of NMD is a hybrid approach with the existing first-order discrete adjoint method, which constructs a series of intermediate Jacobian matrices in order to solve for an entire gradient with a cost that may be independent of the number of program inputs N. Applying a first-order stencil to the Jacobian matrices can compute one or more implicit Hessian-vector products without evaluating the full Hessian, again with cost independent of N. If the vectors in question are basis vectors in cardinal directions, the entire Hessian can be computed one column at a time with a cost linearly dependent on N, which makes NMD competitive with existing second-order adjoint approaches that require far more implementation effort. Further, the application of second- or higher-order stencils can find higher derivatives with similarly reduced cost.

As a proof of concept, both standalone and adjoint NMD were applied to a stochastic diffusion problem that took as input many coefficients for a Karhunen-Loève expansion of a random diffusivity field. First and second derivatives of this problem were found with exactly the same precision as that of the original outputs, as were Hessian-vector products. The cost of the calculation was disproportionate to the amount of new information found, but the same could be said of many AD tools.

As a corollary contribution, it was shown that the existing complex-step differentiation (CD) technique can likewise be applied to the adjoint method to find a Hessian-vector product. If existing code is complex-analytic and purely real-valued at all steps, the CD-adjoint hybrid is a superior choice because it has a significantly lower cost.

Bibliography

[1] Christian H Bischof, Alan Carle, George F Corliss, Andreas Griewank, and Paul D Hovland. ADIFOR: Generating derivative codes from fortran programs. Scientific Programming, 1992.

[2] Christian H Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of fortran 77 programs. IEEE Computational Science & Engineering, 1996.

[3] Christian H Bischof, Lucas Roh, and Andrew Mauer. ADIC — an extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience, 1997.

[4] J D Pryce and J K Reid. A fortran 90 code for automatic differentiation. tech. rep. ral-tr-1998-057. Technical report, Chilton, Didcot, Oxfordshire., 1998.

[5] Eric T Phipps, Roscoe A Bartlett, Gay M David, and Robert J Hoekstra. Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. Advances in Automatic Differentiation, 2008.

[6] A Walther and A Griewank. Getting started with ADOL-C. In Combinatorial Scientific Computing, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[7] L Hascoët and V Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. ACM Transactions on Mathematical Software, 2013.

[8] Mark Schmidt. minfunc. http://www.cs.ubc.ca/~schmidtm/Software/minFunc.html, 2005.

[9] Openmdao. http://openmdao.org/, 2005.

[10] HEEDS MDO. http://www.redcedartech.com/products/heeds_mdo, 2016.

[11] Michael S Eldred, Anthony A Giunta, Samuel S Collis, et al. Second-order corrections for surrogate-based optimization with model hierarchies. American Institute of Aeronautics and Astronautics, 2004.

[12] William W Hager and Hongchao Zhang. A survey of nonlinear conjugate gradient methods. Pacific journal of Optimization, 2(1):35–58, 2006.

[13] James W Daniel. The conjugate gradient method for linear and nonlinear operator equations. SIAM Journal on Numerical Analysis, 4(1):10–26, 1967.

[14] Neculai Andrei. Scaled memoryless BFGS preconditioned conjugate gradient algorithm for unconstrained optimization. Optimization Methods and Software, 22(4):561–571, 2007.

[15] Neculai Andrei. Accelerated conjugate gradient algorithm with finite difference hessian/vector product approximation for unconstrained optimization. Journal of Computational and Applied Mathematics, 230(2):570–582, 2009.

[16] James Martens. Deep learning via hessian-free optimization. Proceedings of the 27th International Conference on Machine Learning, 2010.

[17] James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pages 1033–1040, 2011.

[18] Xi He, Dheevatsa Mudigere, Mikhail Smelyanskiy, and Martin Takác. Large scale distributed hessian-free optimization for deep neural network. CoRR, abs/1606.00511, 2016.

[19] Barak A Pearlmutter. Fast exact multiplication by the hessian. Neural Computation, 1993.

[20] Mike B Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Advances in Automatic Differentiation, pages 35–44. Springer, 2008.

[21] I Ozak, F Kimura, and M Berz. Higher-order sensitivity analysis of finite element method by automatic differentiation. Computational Mechanics, 16:223–234, 1995 1995.

[22] Markus P Rumpfkeil, Wataru Yamazaki, and Dimitri J Mavriplis. Design optimization utilizing gradient/hessian enhanced surrogate model. 2010.

[23] Markus P Rumpfkeil and Dimitri J Mavriplis. Efficient hessian calculations using automatic differentiation and the adjoint method with applications. AIAA journal, 48(10):2406–2417, 2010.

[24] Markus P Rumpfkeil, Wataru Yamazaki, and Dimitri J Mavriplis. Uncertainty analysis utilizing gradient and hessian information. In Computational Fluid Dynamics 2010, pages 261–268. 2011.

[25] D. I. Papadimitriou and K. C. Giannakoglou. Direct, adjoint and mixed approaches for the computation of hessian in airfoil design problems. International Journal for Numerical Methods in Fluids, 56:1929–1943, 2008.

[26] Jeffrey Alan Fike. Multi-objective optimization using hyper-dual numbers. PhD thesis, Stanford University, 2013.

[27] Neculai Andrei. A scaled BFGS preconditioned conjugate gradient algorithm for unconstrained optimization. Applied Mathematics Letters, 20(6):645–650, 2007.

[28] Stephen G Nash. A survey of truncated-newton methods. Journal of Computational and Applied Mathematics, 124(1):45–59, 2000.

[29] Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. Neural computation, 14(7):1723–1738, 2002.

[30] H Pearl Flath, Lucas C Wilcox, Volkan Akçelik, Judith Hill, Bart van Bloemen Waanders, and Omar Ghattas. Fast algorithms for bayesian uncertainty quantification in large-scale linear inverse problems based on low-rank partial hessian approximations. SIAM Journal on Scientific Computing, 33(1):407–432, 2011.

[31] Alexander Kalmikov and Patrick Heimback. A hessian-based method for uncertainty quantification in global ocean state estimations. SIAM Journal of Scientific Computing, 36:S267–S295, 2014.

[32] Raphael T Haftka. Second-order sensitivity derivatives in structural analysis. AIAA journal, 20(12):1765–1766, 1982.

[33] Raphael T Haftka and Z Mroz. First- and second-order sensitivity analysis of linear and nonlinear structures. AIAA journal, 24(7):1187–1192, 1986.

[34] Hil Meijer, Fabio Dercole, and Bart Oldeman. Numerical bifurcation analysis. In Encyclopedia of Complexity and Systems Science, pages 6329–6352. Springer, 2009.

[35] PG Drazin and Y Tourigny. Numerical study of bifurcations by analytic continuation of a function defined by a power series. SIAM Journal on Applied Mathematics, 56(1):1–18, 1996.

[36] Martin Berz. Differential algebraic description of beam dynamics to very high orders. Part. Accel., 24(SSC-152):109–124, 1988.

[37] Pascal Sebah and Xavier Gourdon. Newton's method and high order iterations. 2001.

[38] Annie AM Cuyt and Louis B. Rall. Computational implementation of the multivariate halley method for solving nonlinear systems of equations. ACM Transactions on Mathematical Software (TOMS), 11(1):20–36, 1985.

[39] G Lantoine, R P Russell, and T Dargent. Using multicomplex variables for automatic computation of high-order derivatives. ACM Transactions on Mathematical Software, 38, 2012 2012.

[40] Alberto Abad. Computing derivatives of a gravity potential by using automatic differentiation. Celestial Mechanics and Dynamical Astronomy, 117:187–200, 2013 2013.

[41] Nicol N Schraudolph and Thore Graepel. Combining conjugate direction methods with stochastic approximation of gradients. Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, 2003.

[42] Andreas Griewank and Andrea Walther. Evaluating derivatives: principles and techniques of algorithmic differentiation. Society for Industrial and Applied Mathematics, 2008.

[43] Magnus Rudolph Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems, volume 49. 1952.

[44] Ronald L Iman and Jon C Helton. An investigation of uncertainty and sensitivity analysis techniques for computer models. Risk analysis, 8(1):71–90, 1988.

[45] Andrea Saltelli, Karen Chan, E Marian Scott, et al. Sensitivity analysis, volume 1. Wiley New York, 2000.

[46] Andrea Saltelli, Stefano Tarantola, and Francesca Campolongo. Sensitivity analysis as an ingredient of modeling. Statistical Science, pages 377–395, 2000.

[47] Ji Peng, Jerrad Hampton, and Alireza Doostan. On polynomial chaos expansion via gradient-enhanced l1-minimization. Journal of Computational Physics, 310:440–458, 2016.

[48] André I Khuri and Siuli Mukhopadhyay. Response surface methodology. Wiley Interdisciplinary Reviews: Computational Statistics, 2(2):128–149, 2010.

[49] Wataru Yamazaki and Dimitri J Mavriplis. Derivative-enhanced variable fidelity surrogate modeling for aerodynamic functions. AIAA journal, 51(1):126–137, 2012.

[50] Yonas B Dibike, Slavco Velickov, Dimitri Solomatine, and Michael B Abbott. Model induction with support vector machines: introduction and applications. Journal of Computing in Civil Engineering, 15(3):208–216, 2001.

[51] J Sreekanth and Bithin Datta. Multi-objective management of saltwater intrusion in coastal aquifers using genetic programming and modular neural network based surrogate models. Journal of Hydrology, 393(3):245–256, 2010.

[52] Rudolph van der Merwe, Todd K Leen, Zhengdong Lu, Sergey Frolov, and Antonio M Baptista. Fast neural network surrogates for very high dimensional physics-based models in computational oceanography. Neural Networks, 20(4):462–478, 2007.

[53] MS Eldred and DM Dunlavy. Formulations for surrogate-based optimization with data fit, multifidelity, and reduced-order models. American Institute of Aeronautics and Astronautics, 2006.

[54] Anindya Chatterjee. An introduction to the proper orthogonal decomposition. Current science, 78(7):808–817, 2000.

[55] Natalia M Alexandrov, John E Dennis Jr, Robert Michael Lewis, and Virginia Torczon. A trust-region framework for managing the use of approximation models in optimization. Structural optimization, 15(1):16–23, 1998.

[56] Danny C Sorensen. Newton's method with a model trust region modification. SIAM Journal on Numerical Analysis, 19(2):409–426, 1982.

[57] Steven T Smith. Optimization techniques on riemannian manifolds. 1994.

[58] Craig Bakker, Geoff T Parks, and Jerome P Jarrett. On the application of differential geometry to MDO. In 12th Aviation Technology, Integration and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. AIAA.

[59] C Bakker, GT Parks, and JP Jarrett. Optimization algorithms and ode's in mdo. In ASME 2013 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, ASME, Portland, Oregon, 2013.

[60] Bruno Cochelin and Marc Medale. Power series analysis as a major breakthrough to improve the efficiency of asymptotic numerical method in the vicinity of bifurcations. Journal of Computational Physics, 236:594–607, 2013.

[61] B. Fornberg. Finite difference method. Scholarpedia, 6(10):9685, 2011. revision no. 91262.

[62] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. SIAM Journal on Numerical Analysis, 4:202–210, 1967.

[63] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J Alonso. The complex-step derivative approximation. ACM Transactions on Mathematical Software, 29:245–262, 2003.

[64] B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. Mathematics of Computation, 51:699–706, 1988.

[65] B. Fornberg. Calculation of weights in finite difference formulas. SIAM Review, 40:685–691, 1998.

[66] Michael B Giles and Niles A Pierce. An introduction to the adjoint approach to design. Flow, turbulence and combustion, 65(3-4):393–415, 2000.

[67] Andrew M. Bradley. Pde-constrained optimization and the adjoint method. Tutorial, Stanford Computer Science, 2013.

[68] Larry L. Green Perry A. Newman Gene W. Hou Laura L. Sherman, Arthur C. Taylor III and Vamshi Mohan Korivi. First- and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. JOURNAL OF COMPUTATIONAL PHYSICS, 129:307–331, 1996.

[69] Gene J-W Hou and Jeenson Sheen. Numerical methods for second-order shape sensitivity analysis with applications to heat conduction problems. International journal for numerical methods in engineering, 36(3):417–435, 1993.

[70] Devendra Ghate and Michael B Giles. Efficient hessian calculation using automatic differentiation. In 25th AIAA Applied Aerodynamics Conference. American Institute of Aeronautics and Astronautics, Inc, 2007.

[71] Tools for automatic differentiation. http://www.autodiff.org/?module=Tools. Accessed 2016-10-08.

[72] Andreas Griewank. On automatic differentiation. In Masao Iri and K. Tanabe, editors, Mathematical Programming, pages 83–108. Kluwer Academic Publishers, Dordrecht, 1989.

[73] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM. Technical Report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959. (In Russian).

[74] Masao Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding errors : Complexity and practicality. Departmental bulletin paper, Kyoto University, 1984.

[75] COmputational INfrastructure for Operations Research (COIN-OR). Automatic Differentiation by OverLoading in C++ (ADOL-C). ADOL-C 2.6.2 Online Documentation.

[76] Anthony S. Wexler. An algorithm for exact evaluation of multivariate functions and their derivatives to any order. Computational Statistics & Data Analysis, 6(1):1 – 6, 1988.

[77] Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. ACM Trans. Math. Softw., 18(2):159–173, June 1992.

[78] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate taylor series. Mathematics of Computation of the American Mathematical Society, 69(231):1117–1130, 2000.

[79] Andrea Walther. Computing sparse hessians with automatic differentiation. ACM Transactions on Mathematical Software (TOMS), 34(1):3, 2008.

[80] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. SIAM review, 47(4):629–705, 2005.

[81] LAPACK–Linear Algebra PACKage. http://www.netlib.org/lapack/, 2015.

[82] Intel ® MKL. Intel® Math Kernel Library. https://software.intel.com/en-us/intelmkl, 2016.

[83] OpenBLAS: An optimized BLAS library. http://www.openblas.net/, 2016.

[84] Apple iOS Library. Accelerate Framework Reference. https://developer.apple.com/library/ios/documentation/Accelerate/Reference/AccelerateFWRef/, 2016.

[85] Paul S Dwyer and MS MacPhail. Symbolic matrix derivatives. The annals of mathematical statistics, pages 517–534, 1948.

[86] Christian H. Bischof, H. Martin Bücker, Bruno Lang, Arno Rasch, and Andre Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), pages 65–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[87] Christian Bischof, Bruno Lang, and Andre Vehreschild. Automatic differentiation for matlab programs. PAMM, 2(1):50–53, 2003.

[88] Sandia National Laboratories. Trilinos home page. Trilinos Online Documentation (https://trilinos.org/).

[89] Sandia National Laboratories. Sacado: Automatic differentiation tools for C++ applications. Trilinos Online Documentation (https://trilinos.org/docs/dev/packages/sacado/doc/html/index.html).

[90] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[91] Cleve B. Moler. Complex step differentiation. http://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/, 2013.

[92] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. SIAM Review, 40:110–112, 1998.

[93] Joaquim R. R. A. Martins. A COUPLED-ADJOINT METHOD FOR HIGH-FIDELITY AERO-STRUCTURAL OPTIMIZATION. PhD thesis, stanford university, 2002,

[94] Awad H. Al-Mohy and Nicholas J. Higham. The complex step approximation to the fréchet derivative of a matrix function. Numerical Algorithms, 53(1):133, 2009.

[95] Corrado Segre. The real representation of complex elements and hyperalgebraic entities (italian). Mathematische Annalen, 40:413–67, 1892.

[96] G Baley Price. An Introduction to Multicomplex Spaces and Functions. Marcel Dekker, 1991.

[97] Benjamin Z Dunham and Ryan P Starkey. Qcg: Quasicomplex gradients for efficient and accurate computation of any-order derivative tensors. In 12th Aviation Technology, Integration and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. AIAA.

[98] W K Clifford. Preliminary sketch of bi-quaternions. Proceedings of the London Mathematical Society, 4:381–395, 1873.

[99] Barak A Pearlmutter and Jeffery Mark Siskind. Lazy multivariate higher-order forward-mode ad. 2007.

[100] bxsfan [pseud.]. ad4julia/DualNumbers.jl. https://github.com/bsxfan/ad4julia/blob/master/DualNumbers.jl, 2013.

[101] Jeffrey Fike. hyperdual.h. http://adl.stanford.edu/hyperdual/hyperdual.h, 2014.

[102] Jeffrey A Fike and Juan 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. AIAA.

[103] Juan J. Alonso Jeffrey A. Fike, Sietse Jongsma and Edwin van der Weide. Optimization with gradient and hessian information calculated using hyper-dual numbers. In 29th AIAA Applied Aerodynamics Conference. AIAA.

[104] Jeffrey A Fike and Juan J Alonso. Automatic differentiation through the use of hyper-dual numbers for second derivatives. volume 87 of Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2012.

[105] Joaquim RRA Martins and John T Hwang. Review and unification of methods for computing derivatives of multidisciplinary computational models. AIAA journal, 51(11):2582–2599, 2013.

[106] anon. adnumber: Software to compute derivatives of arbitrary order to machine precision.

[107] ELEKS Software. eleks/ADEL: Template-based automatic differentiation library.

[108] Dario Izzo and Francesco Biscani. The algebra of truncated polynomials. http://darioizzo.github.io/audi/theory.html, 2015.

[109] COmputational INfrastructure for Operations Research (COIN-OR). A Package for Differentiation of C++ Algorithms. ADOL-C 2.6.2 Online Documentation.

[110] Ramon E Moore. Methods and applications of interval analysis, volume 2. SIAM.

[111] Richard D Neidinger. Computing multivariable taylor series to arbitrary order. In ACM SIGAPL APL Quote Quad, volume 25, pages 134–144. ACM, 1995.

[112] Richard Neidinger. Directions for computing truncated multivariate taylor series. Mathematics of computation, 74(249):321–340, 2005.

[113] Kantor and Solodovnikov. Hypercomplex numbers. Springer-Verlag, Berlin, New York, 1978 (in German), 1989 (in English).

[114] Karen Parshall. Wedderburn and the structure of algebras. Archive for History of Exact Sciences, 32:223–349, 1985.

[115] O Knill. Lecture notes in linear algebra and differential equations – Exhibit: Other Numbers. http://www.math.harvard.edu/archive/21b_spring_08/exhibits/dualnumbers/, 2008.

[116] E. W. Weisstein. Dual number. http://mathworld.wolfram.com/DualNumber.html.

[117] E. W. Weisstein. Exact numeric Nth derivatives. http://jliszka.github.io/2013/10/24/exact-numeric-nth-derivatives.html, 2013.

[118] Bogdan V. Constantin. Application of the complex step method to chemistry-transport modeling. Master's thesis, Massachusetts Institute of Technology, 5 2014.

[119] A. Hakami D. K. Henze and J. H. Seinfeld. Development of the adjoint of geos-chem. Atmospheric Chemistry and Physics, 7:2413–2433, 2007.

[120] Ruye Wang. E161: Computer Image Processing and Analysis – Karhunen-Loeve Transform (KLT). http://fourier.eng.hmc.edu/e161/lectures/klt/node3.html, 2016.

[121] Alireza Doostan. ASEN 6619/MCEN 6228 Uncertainty Quantification, Lecture 6 Notes, 2011.

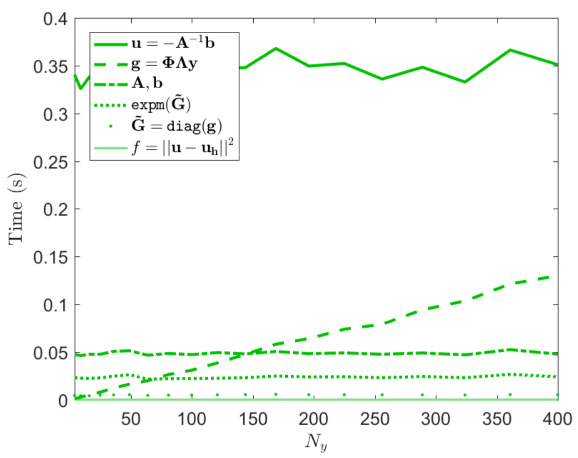
[122] TIMOTHY A. DAVIS and IAIN S. DUFF. An unsymmetric-pattern multifrontal method for sparse lu factorization. SIAM Journal of Matrix Analysis Applications, 18(1):140–158, 1997.

Appendix A

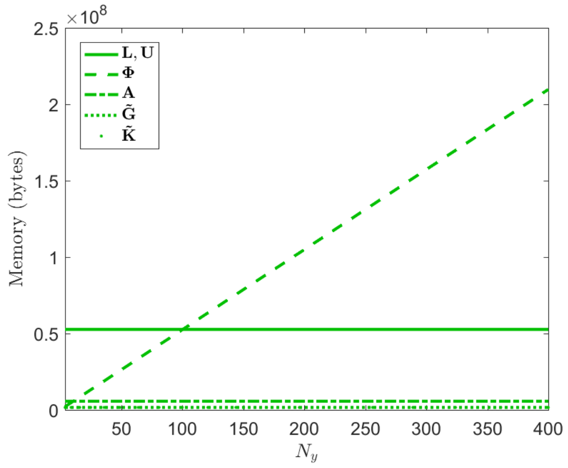
Time and Memory Benchmarks

A.1 Costs vs Number of Eigenfunctions

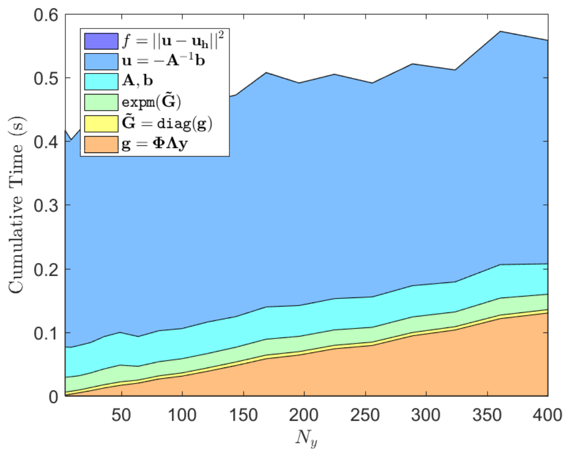
A.1.1 Original Forward Problem



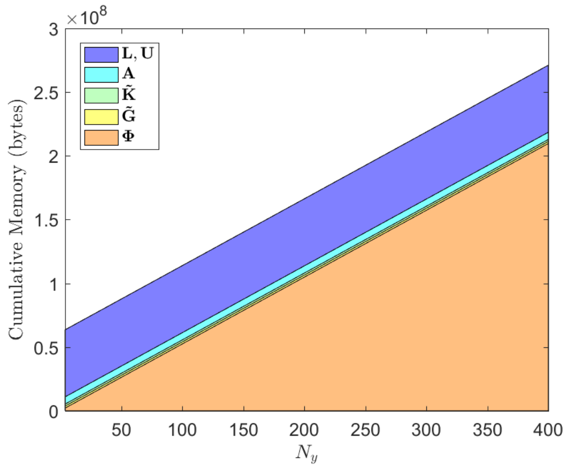
Wall-Clock Time of Program Steps



Memory Requirements of Matrices



Cumulative Wall-Clock Time

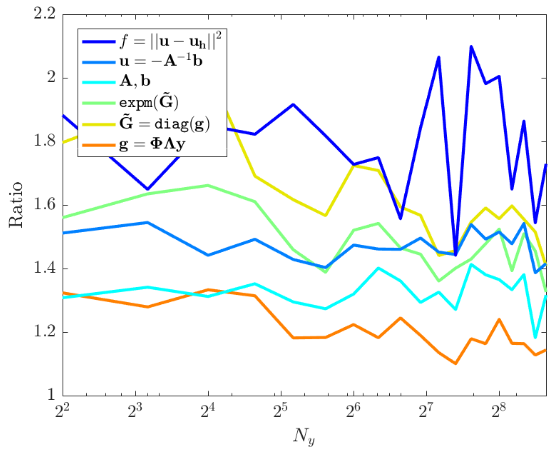


Cumulative Memory Requirements

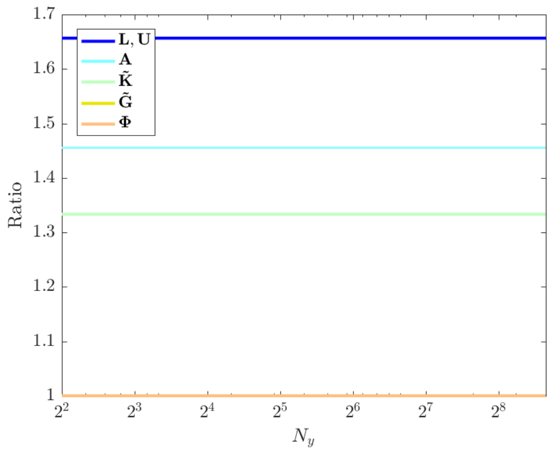
Figure A.1: Costs of the original matrix-focused diffusion simulation (before differentiation was applied), excluding cost of the KLE, as a function of the number of eigenfunction coefficients N_y (i.e., the number of program inputs). Left column: time-cost of each significant program step. Right: memory cost of the biggest matrices. Top: Individual costs. Bottom: Cumulative cost.

For the forward problem, only the size of the eigenfunction matrix $\mathbf{\Phi}$ is dependent on N_y (linearly so), and only the calculation of $\mathbf{g}$ is in turn dependent on $\mathbf{\Phi}$. All other costs, including the solve, are independent of N_y. Total costs of the program are thus $\mathcal{O}(N_y)$, although the overwhelming majority of the computation – even for large N_y – is $\mathcal{O}(1)$.

A.1.2 Forward Complex-Step Differentiation



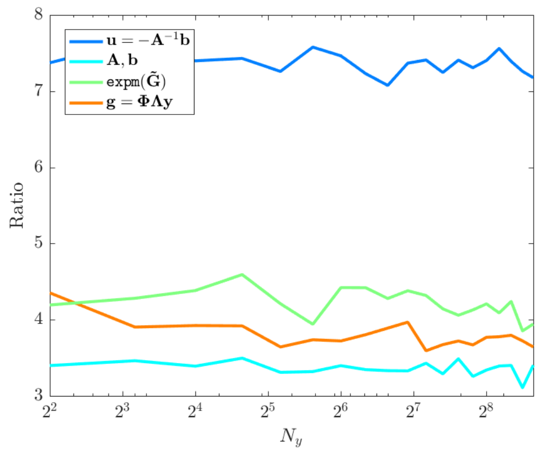
Time Relative to Original



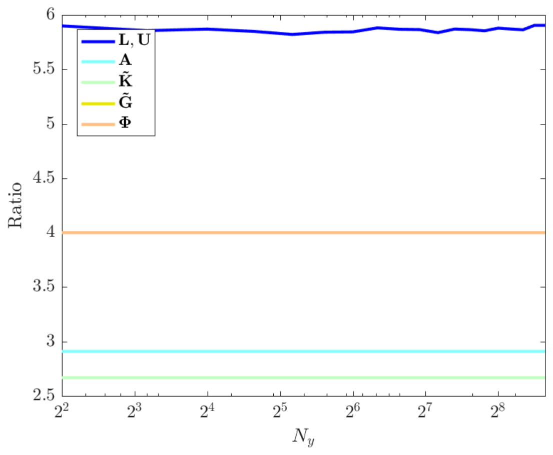
Memory Relative to Original

Figure A.2: Relative costs of the forward code after CD has been applied. Each line represents fractional increase of cost compared to original program. All cost-growth is independent of N_y – if the original was $\mathcal{O}(1)$, then CD would be, as well.

A.1.3 Forward Dual-Number Stencil Nilpotent Matrix Differentiation



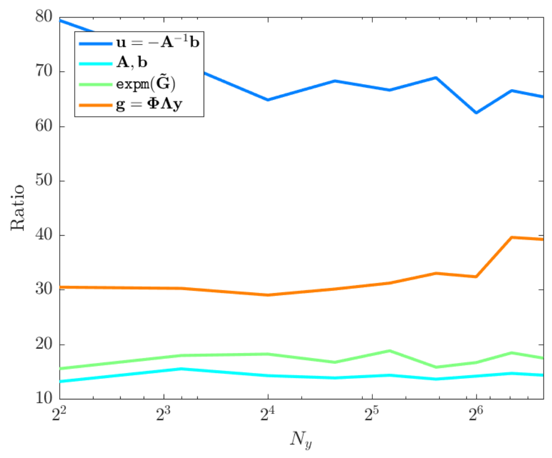
Time Relative to Original



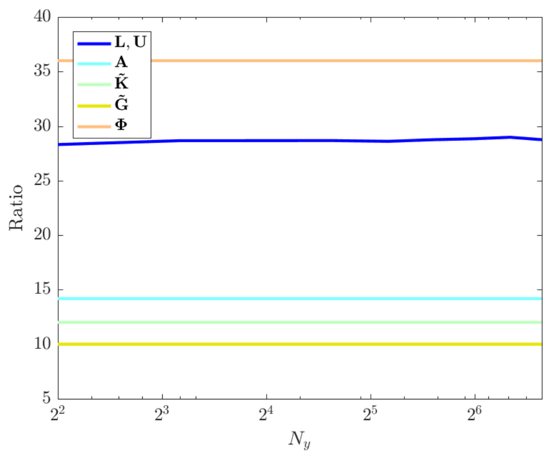
Memory Relative to Original

Figure A.3: Relative costs of the forward code under dual-number NMD. All cost-growth is independent of N_y.

A.1.4 Forward Two-Variable, Second-Order Stencil Nilpotent Matrix Differentiation



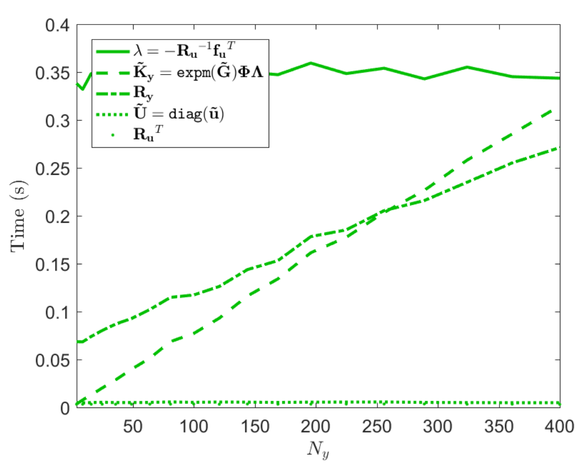
Time Relative to Original



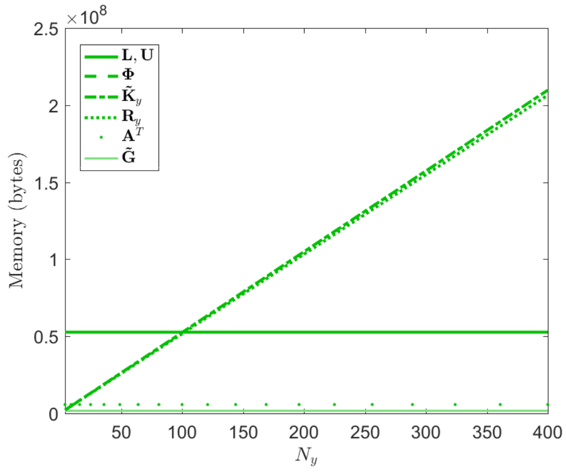
Memory Relative to Original

Figure A.4: Relative costs of the forward code after the stencil from Equation 8.81 has been applied. Each line represents fractional increase of cost compared to original program. All cost-growth is independent of N_y.

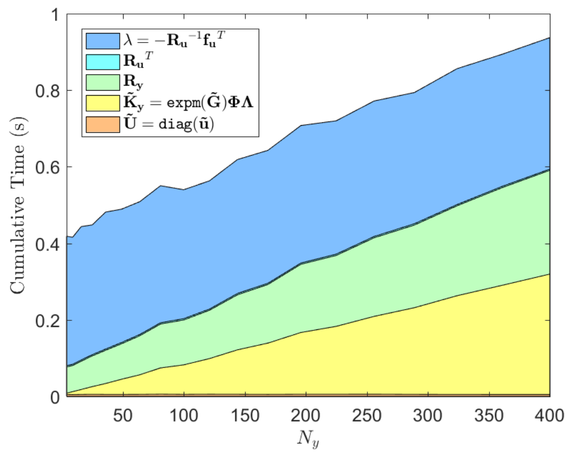
A.1.5 Original Adjoint Problem



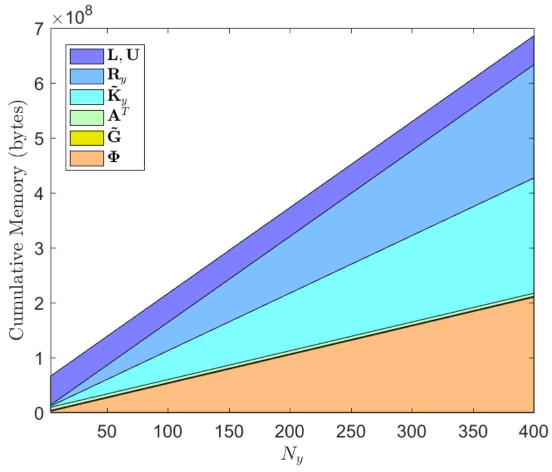
Wall-Clock Time of Program Steps



Memory Requirements of Matrices



Cumulative Wall-Clock Time

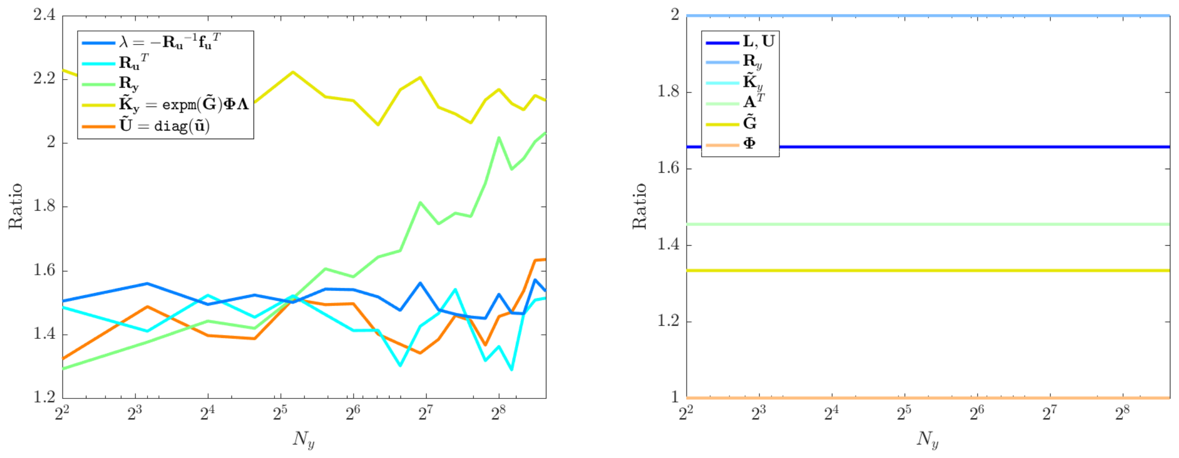


Cumulative Memory Requirements

Figure A.5: Costs of the original adjoint code (before differentiation was applied), excluding cost of the KLE, as a function of the number of eigenfunction coefficients N_y (i.e., the number of program inputs). Left column: time-cost of each significant program step. Right: memory cost of the biggest matrices. Top: Individual costs. Bottom: Cumulative cost.

For the adjoint problem, three matrices are linearly dependent on N_y: Φ, $\tilde{\mathbf{K}}_\mathbf{y}$, and $\mathbf{R_y}$. The costs of multiplying by these matrices similarly increase. All other costs, including the solve, are independent of N_y. Costs of this implementation of the adjoint method are thus unfortunately $\mathcal{O}(N_y)$ – although note that (1) the forward method is similarly $\mathcal{O}(N_y)$ and (2) the total adjoint cost is less than twice that of the forward problem (Figure A.1).

A.1.6 Adjoint Complex-Step Differentiation

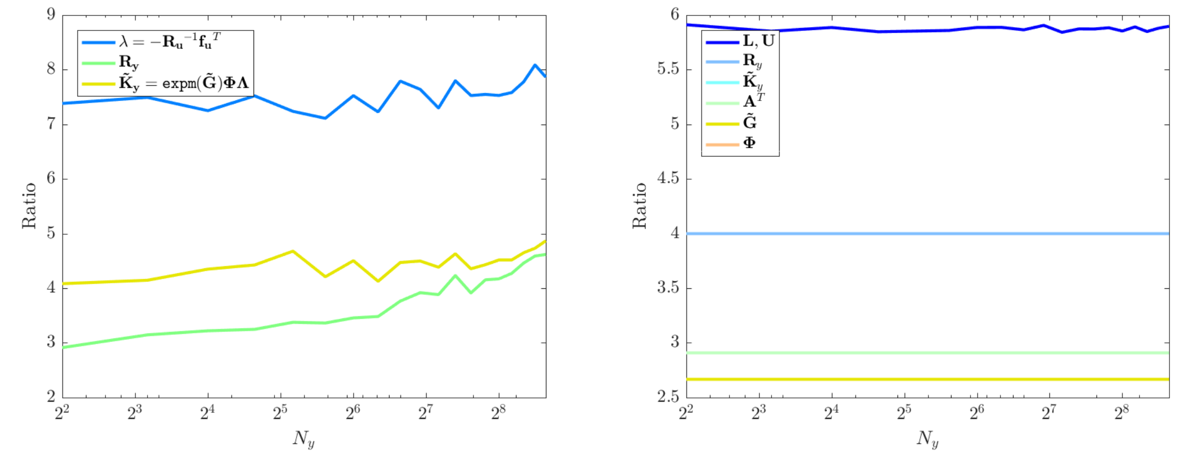


Time Relative to Original

Memory Relative to Original

Figure A.6: Relative costs of the adjoint code after CD has been applied. The cost of constructing $\mathbf{R_y}$ arguably grows somewhat, but note the logarithmic x-axis and the tight range of the y-axis – the growth is insignificant.

A.1.7 Adjoint Dual-Number Stencil Nilpotent Matrix Differentiation

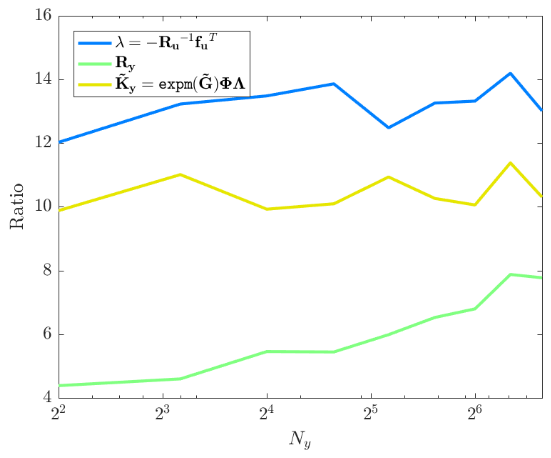


Time Relative to Original

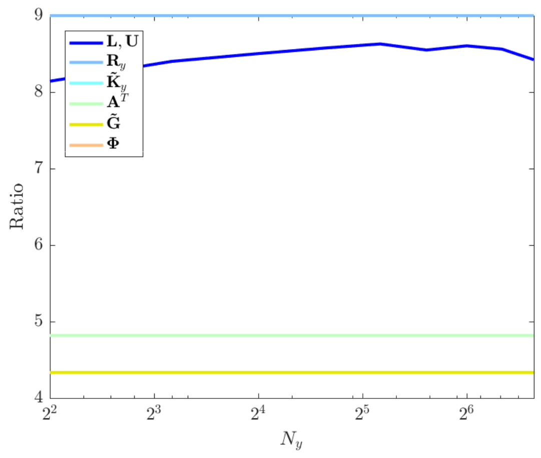
Memory Relative to Original

Figure A.7: Relative costs of the adjoint code after dual-number NMD has been applied. All cost-growth is independent of N_y or insignificant.

A.1.8 Adjoint Two-Variable, First-Order Stencil Nilpotent Matrix Differentiation



Time Relative to Original

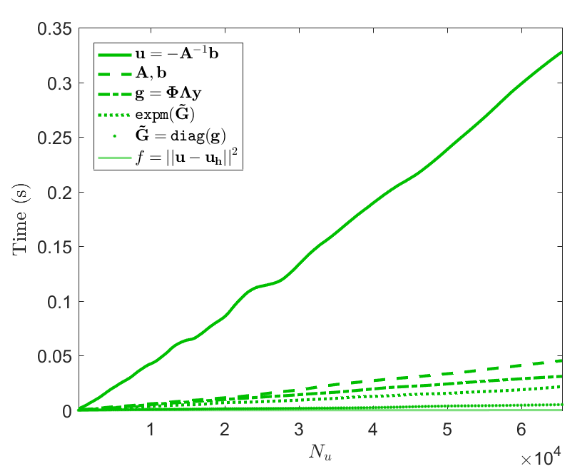


Memory Relative to Original

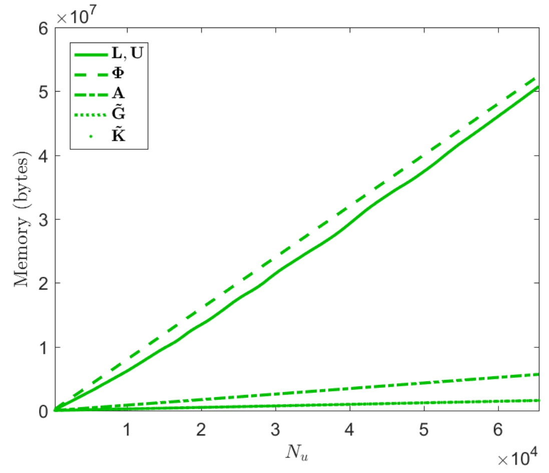
Figure A.8: Relative costs of the adjoint code after the stencil from Equation 8.82 has been applied. Each line represents fractional increase of cost compared to unmodified adjoint. All cost-growth is independent of N_y or insignificant.

A.2 Costs vs Size of System (Number of Mesh Nodes)

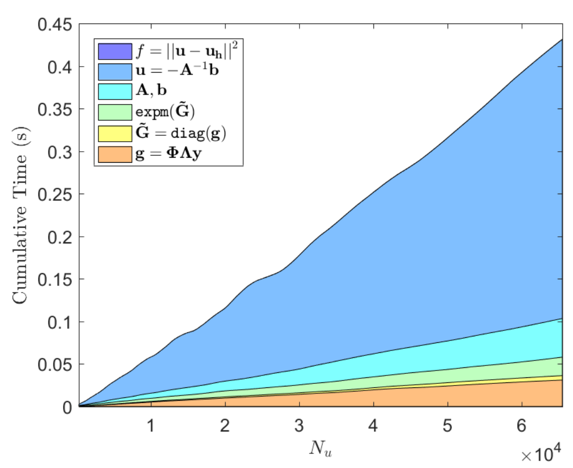
A.2.1 Original Forward Problem



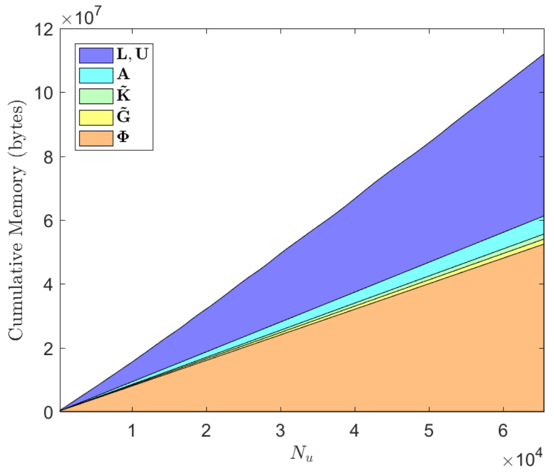
Wall-Clock Time of Program Steps



Memory Requirements of Matrices



Cumulative Wall-Clock Time

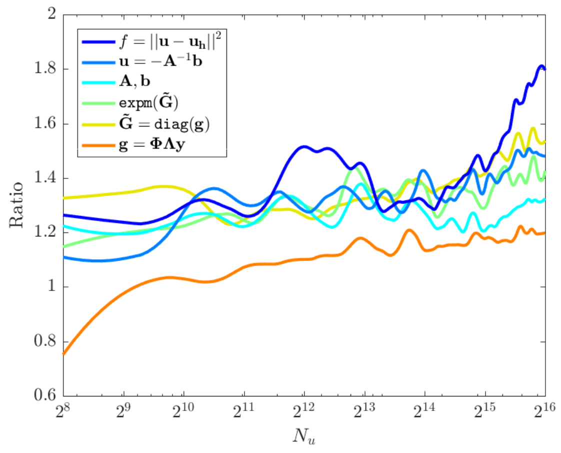


Cumulative Memory Requirements

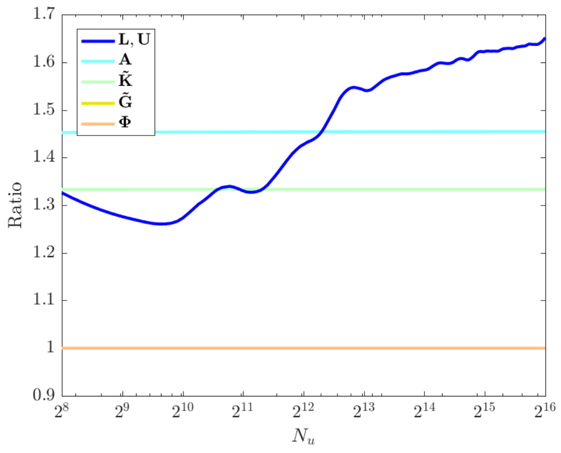
Figure A.9: Costs of the original matrix-focused diffusion simulation (before differentiation was applied), excluding cost of the KLE, as a function of the number of mesh nodes. Left column: time-cost of each significant program step. Right: memory cost of the biggest matrices. Top: Individual costs. Bottom: Cumulative cost.

Solving the system requires the overwhelming bulk of the computational effort, while memory requirements are equally split between the eigenfunctions $\mathbf{\Phi}$ and the $\mathbf{L}, \mathbf{U}$ decomposition built automatically (and hidden from the user and the profiler) by MATLAB's sparse solver.

A.2.2 Forward Complex-Step Differentiation



Time Relative to Original

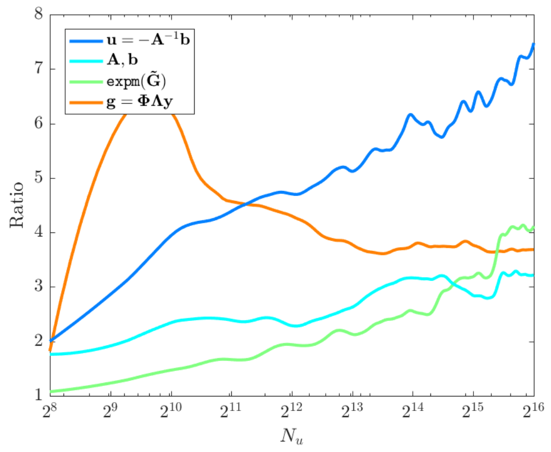


Memory Relative to Original

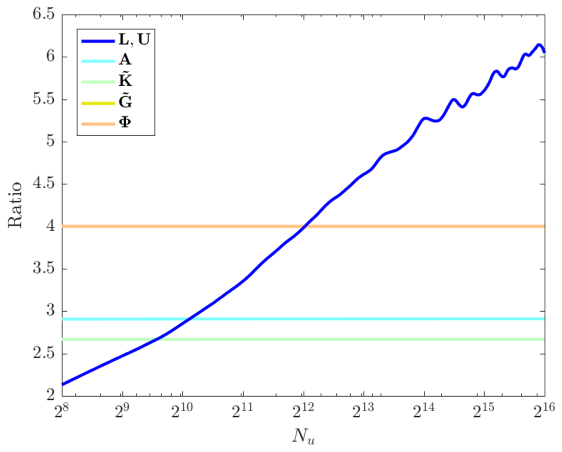
Figure A.10: Costs of the diffusion program after CD has been applied in the forward mode. Each line represents fractional increase of cost compared to original program. Despite providing twice as much information, costs are less than twice that of the original program. Note that $\mathbf{\Phi}$ does not carry derivative information, and thus is purely real-valued and requires no more memory than the original.

Note this is a semilog plot – the straight-line growth of the $\mathbf{L}, \mathbf{U}$ memory requirements for $N_u > 2^{13}$ may imply logarithmic growth, which also appears (albeit much more clearly) for NMD in Figure A.11; more likely for this version, though, is an asymptotic approach to a constant memory ratio – probably exactly two, where every scalar has an imaginary component.

A.2.3 Forward Dual-Number Stencil Nilpotent Matrix Differentiation



Time Relative to Original

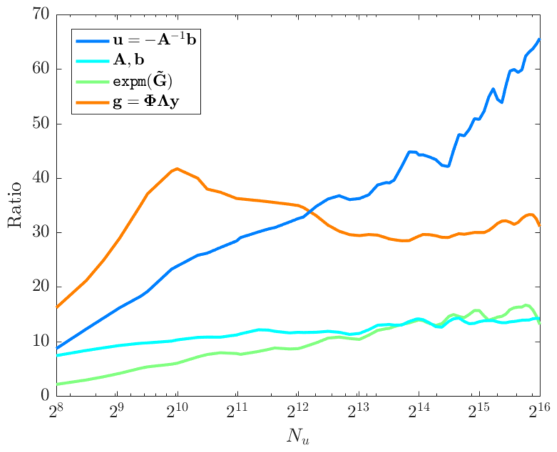


Memory Relative to Original

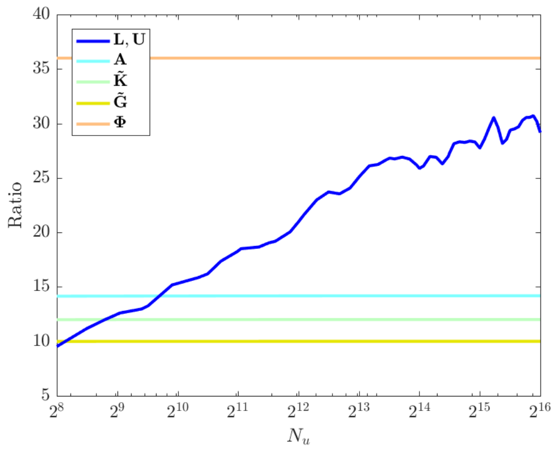
Figure A.11: Costs of the diffusion program after dual-number NMD has been applied in the forward mode. Each line represents fractional increase of cost compared to original program. Within measurement error, most of the less-important functions and matrices (see relative sizes in Figure A.9) are constant multiples of the original. Exceptions appear to include the matrix exponential `expm()` function (which was a very small fraction of the program to begin with, and thus not a concern), as well as the much more important sparse solve and LU factorization.

Noting that this is a semilog-x plot, the straight-line behavior of the solve time and the memory requirements of $\mathbf{L}, \mathbf{U}$ imply logarithmic growth. The memory growth is especially concerning – a dual-number stencil should use at most three times as much memory for a sparse matrix, or four times as much for a dense matrix like Φ. This rule is satisfied by every matrix except the two factors.

A.2.4 Forward Two-Variable, Second-Order Stencil Nilpotent Matrix Differentiation



Time Relative to Original

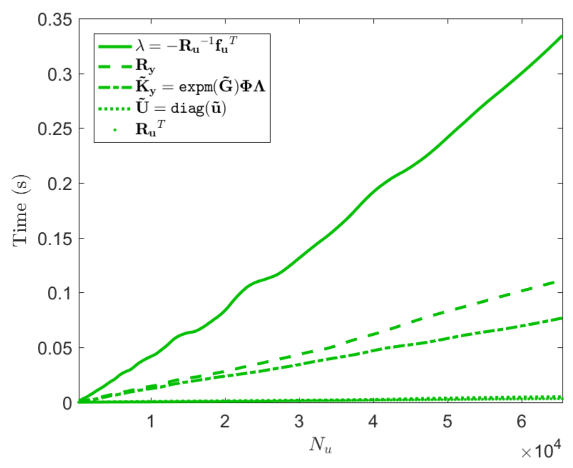


Memory Relative to Original

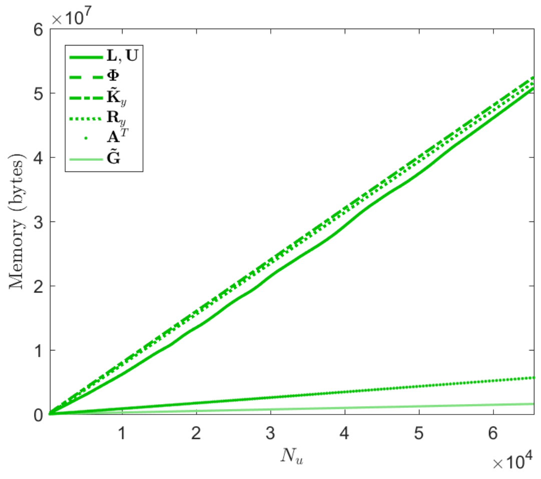
Figure A.12: Costs of the diffusion program after NMD has been applied in the forward mode. Each line represents fractional increase of cost compared to original program. These results are for the stencil defined in Equation 8.81 that calculates two first derivatives and all three second derivatives from a two-input Hessian matrix, for a total of six times as much information; the 6×6 stencil should contain at maximum 15 non-zero values for a sparse representation, or 36 scalar values for a dense representation.

Similar observations to Figure A.11 apply: the sparse solver time and LU factorization memory costs appear to have logarithmic growth. Note that while $\mathbf{L}, \mathbf{U}$ have not exceeded expectations as egregiously as for the dual-number stencil (in that they do not take more memory than even a dense fill-in would imply), they have still gone past the expected maximum for a sparse representation.

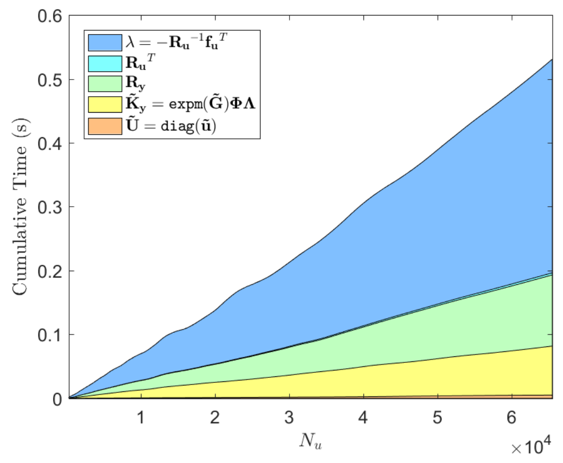
A.2.5 Original Adjoint Problem



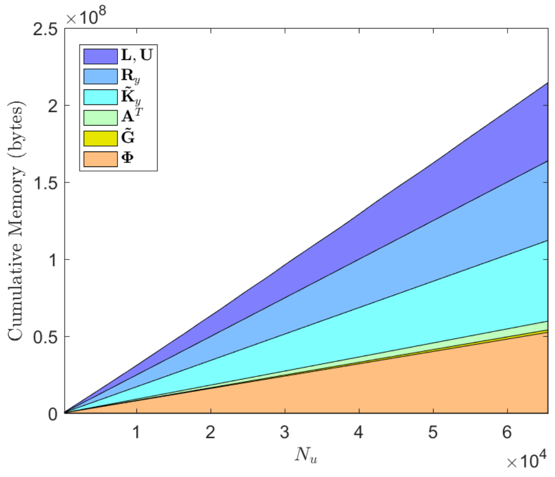
Wall-Clock Time of Program Steps



Memory Requirements of Matrices



Cumulative Wall-Clock Time

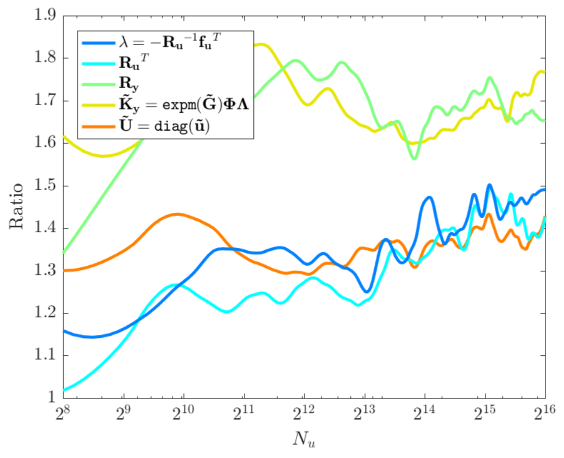


Cumulative Memory Requirements

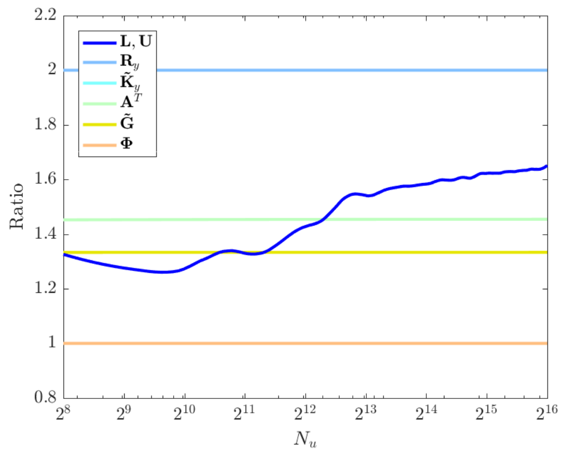
Figure A.13: Costs of the original adjoint code (before differentiation was applied), excluding cost of the KLE and the forward sweep, as a function of the number of mesh nodes. For this experiment, the number of eigenfunctions (and thus the number of first derivatives) is $N_y = 100$. Left column: time-cost of each significant program step. Right: memory cost of the biggest matrices. Top: Individual costs. Bottom: Cumulative cost.

Solving the system requires a slight majority of the computational effort, with the bulk of the remaining time spent on building $\mathbf{R_y}$ and on the dense multiplication required to find $\tilde{\mathbf{K}}_\mathbf{y}$. Memory requirements are equally split four ways between $\mathbf{\Phi}$, $\tilde{\mathbf{K}}_\mathbf{y}$, $\mathbf{R_y}$, and the $\mathbf{L}, \mathbf{U}$ decomposition.

A.2.6 Adjoint Complex-Step Differentiation



Time Relative to Original

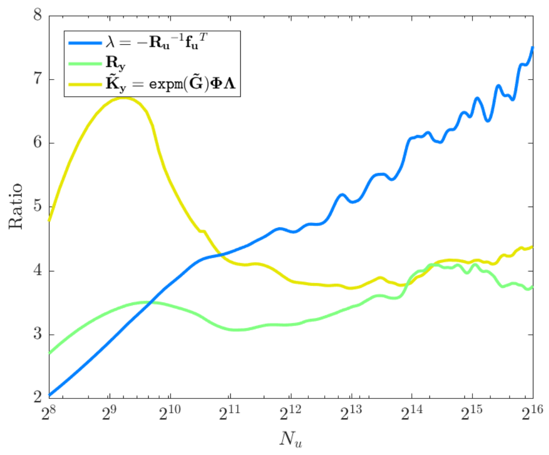


Memory Relative to Original

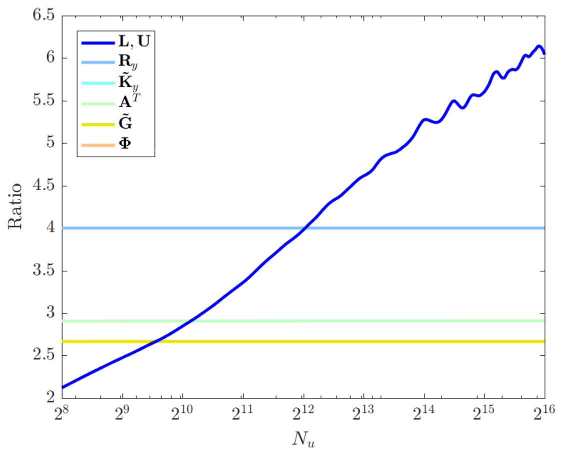
Figure A.14: Costs of the adjoint code after CD has been applied. Each line represents fractional increase of cost compared to unmodified adjoint.

Similar observations apply as in the forward mode (Figure A.10): despite providing twice as much information, costs are less than twice that of the original program; most are constant multiples of the original. The solve and LU factorization exhibit either logarithmic growth or asymptotic approach to a constant, but the total growth is insignificant.

A.2.7 Adjoint Dual-Number Stencil Nilpotent Matrix Differentiation



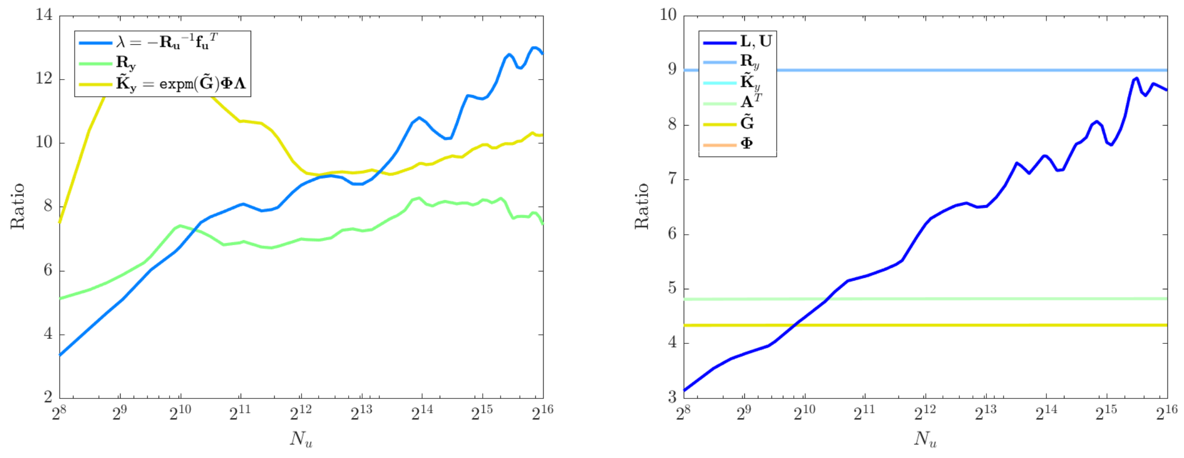
Time Relative to Original



Memory Relative to Original

Figure A.15:
Costs of the diffusion program after dual-number NMD has been applied in the adjoint mode. Each line represents fractional increase of cost compared to unmodified adjoint. The same observations as Figure A.11 still apply: the time-cost of the solve and the memory-cost of the LU factorization appear to be logarithmic. Again, the latter exhibits dense fill-in in excess of the size of the stencil.

A.2.8 Adjoint Two-Variable, First-Order Stencil Nilpotent Matrix Differentiation



Time Relative to Original

Memory Relative to Original

Figure A.16:
Costs of the adjoint code after NMD has been applied. Each line represents fractional increase of cost compared to unmodified adjoint. These results are for the 3×3 stencil given by Equation 8.82 that calculates two first derivatives of the gradient, for a total of three times as much information as the real adjoint; it thereby finds the same derivatives found by the forward-mode 6×6 second-order stencil from Figure A.12, which found the 2×2 sub-Hessian with respect to the first two inputs. More than that, by finding the entirety of the first two columns, this adjoint stencil finds a 100×2 subset of the Hessian.

The same logarithmic growth in the sparse solve and factorization appears as in the preceding NMD experiments.