Journal of Computational Physics 469 (2022) 111527

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Efficient algorithm for proper orthogonal decomposition of block-structured adaptively refined numerical simulations

Michael A. Meehan *, Sam Simons-Wellin, Peter E. Hamlington

Paul M. Rady Department of Mechanical Engineering, University of Colorado, Boulder, 427 UCB, Boulder, CO 80309, United States of America

ARTICLE INFO

Article history:
Received 1 June 2021
Received in revised form 20 May 2022
Accepted 3 August 2022
Available online 18 August 2022

Keywords:
Proper orthogonal decomposition
Adaptive mesh refinement
Computational fluid dynamics

ABSTRACT

Adaptive mesh refinement (AMR) is increasingly being used to simulate fluid flows that have vastly different resolution requirements throughout the computational domain. Proper orthogonal decomposition (POD) is a common tool to extract coherent structures from flow data and build reduced order models, but current POD algorithms do not take advantage of potential efficiency gains enabled by multi-resolution data from AMR simulations. Here, we explore a new method for performing POD on AMR data that eliminates repeated operations arising from nearest-neighbor interpolation of multi-resolution data onto uniform grids. We first outline our approach to reduce the number of computations with examples and provide the complete algorithms in the appendix. We examine the computational acceleration of the new algorithms compared to the standard POD method using synthetically generated AMR data and operation counting. We then use CPU times and operation counting to analyze data from an AMR simulation of an axisymmetric buoyant plume, finding that we are able to reduce the computational time by a factor of approximately $2-5$ when using three levels of grid refinement. The new POD algorithm is the first to eliminate redundant operations for matrix multiplications with repeated values in each matrix, making it ideal for POD of data from AMR simulations.

1. Introduction

High Reynolds number turbulent fluid flows found in nature and engineering typically involve large temporal and spatial scale separations [1]. In many cases, there is also considerable variation in the extent of small-scale motions at different locations. This is most apparent in free shear flows, such as mixing layers, jets, and wakes, where substantial complexity and small-scale structure occur near regions of high shear, while other locations in the flow are relatively (or even completely) quiescent across a broad range of scales. Similar spatial variations in scale separation occur, for example, in supersonic flows with shock waves and in reacting flows with spatially localized chemical reactions. In other flows, such as those with moving shocks or flame fronts, the location of fine-scale flow features can vary rapidly.

These types of flows pose considerable challenges for numerical simulations, particularly those employing uniform grids where very fine grid resolution is used in part of the domain to resolve small-scale features, but the same fine-scale grid is also applied in regions where such resolutions are not required to capture the local flow physics. Statically refined (i.e., non-uniform) grids provide a possible solution to this inefficiency, for example in free shear flows where the region of fine-scale structure does not vary substantially in time. However, static approaches still incur a large computational cost in

* Corresponding author.
E-mail address: mime5507@colorado.edu (M.A. Meehan).

https://doi.org/10.1016/j.jcp.2022.111527
0021-9991/

flows where the region of fine-scale physics changes dynamically (e.g., in simulations of propagating flames or problems with variable geometry).

An increasingly common technique to overcome these challenges is to reduce the computational cost of fluid flow simulations using adaptive mesh refinement (AMR). In AMR simulations, the computational mesh dynamically changes to resolve a particular phenomenon or feature of the flow, typically based on gradients or error estimates. There are many different ways of implementing AMR, including hp refinement using finite elements [2], unstructured grid deformation [3], and block-structured AMR [4]. In the present study, we focus on the last of these approaches and consider block structured AMR to be the union of rectilinear grids that span the entirety of the computational domain, where the increase in resolution along each dimension is an exact integer multiple from the coarsest to any finer resolution. There are many ways to achieve this dynamic refinement, including by tagging and splitting cells and then solving the governing equations on all cells simultaneously [5], or by tagging and solving finer resolutions independent of the coarser underlying grids [6]. For the present purposes it is not necessary to distinguish between these two approaches, and we only require that a simulation be based on block-structured AMR.

Although AMR has the potential to reduce the computational cost of simulating many different flows, it introduces new complexities when post-processing and analyzing the resulting data. In this study, we develop a new tool to perform more efficient proper orthogonal decomposition (POD) of nearest-neighbor-interpolated data from block-structured AMR simulations. The POD methodology [7] is a versatile tool for studying flow fields, including building reduced-order models [8] and analyzing coherent structures [9]. The resulting basis functions, or POD modes, are optimal in the sense that they capture the greatest amount of variance in the fewest possible modes [10]. POD analysis has been applied in a wide range of fluid flow applications, including jets [11], wakes [9], and flames [12], and a review of applications in which POD has been used can be found in Ref. [7].

Despite the utility of both AMR and POD individually, there are many complications that arise when performing POD on data from AMR simulations. First, POD requires that the grid be fixed for all times, which is not generally the case in AMR simulations. Second, the non-uniformity of AMR grids requires a dynamic weighting function. Third, cells at different resolutions in an AMR simulation represent different spatial locations when solution values are defined, for example, at cell centers (as is the case in many finite-volume codes).

The most common approach to overcome these issues is to interpolate data from each simulation output to a fixed reference mesh that is constant in time, and to use an appropriate weighting for the inner product during POD if the mesh is non-uniform [13]. To date, spatially adaptive numerical grids and POD have been used together in a variety of contexts, most commonly in the construction of reduced-order adaptive POD models. In particular, for simulations performed using AMR, it is most efficient to construct a POD-based model that also uses AMR. This technique has been used in ocean modeling [13,14] and wake flows [15], including applications using dynamic mode decomposition [16,17]. In each of these examples, AMR simulation data was first interpolated to a fixed non-uniform reference mesh to make the POD computation possible.

While technically feasible and advantageous from a memory usage perspective, interpolation of AMR simulation data to a non-uniform mesh requires a series of steps that depend on the structure of the output data, which is generally not consistent between different simulation codes. Prior to non-uniform mesh interpolation, all output data from the simulation must first be scanned to determine the finest grid level at each location; typically, this information would be stored on an auxiliary fixed uniform mesh refined at the level of the finest resolution. After this scan is complete, a fixed non-uniform reference mesh is generated and the output data are interpolated to this mesh prior to performing POD.

Because of the cost associated with the first step in this process (or complexity, if a simulation code must be modified to output this information at run-time), as well as the increasing availability of high-memory nodes on modern high performance computing (HPC) systems, it is often more convenient to instead interpolate the AMR output data directly to a fixed uniform mesh, in many cases spanning only a sub-region of the full computational domain. The resulting data can then be straightforwardly used for visualization [18,19], calculation of statistics such as temporal averages [20], and analysis of integrated quantities such as total heat release [21]. Many different software packages also already provide simple tools to extract data onto uniform meshes [22–24], and extraction tools for non-uniform meshes are less common. Although the uniform mesh approach does impose larger memory requirements, the additional cost can often be mitigated by only analyzing part of the simulation domain, particularly if the region of interest is fixed (e.g., in a jet flow). There may also be little difference in memory requirements if the region of refinement does change substantially during the simulation, such that non-uniform and uniform meshes have similar cell counts.

Although various interpolation methods (e.g., cubic or Akima splines) can be used to map AMR output data onto either non-uniform or uniform meshes, an interesting opportunity arises when using nearest-neighbor interpolation onto fixed uniform meshes. Namely, the POD calculation itself can be accelerated by taking advantage of the data repetition introduced by the nearest-neighbor method when representing data at coarse mesh locations on the finer resolution fixed mesh. As outlined in this study, repeated computations during the calculation of POD modes and temporal coefficients can be weighted and skipped to reduce the computational cost. Our approach is most similar to a prior wavelet-based approach [25] where biorthogonal wavelets were used to refine and coarsen the block-structured grid. The wavelet coefficients can then be used directly in the POD computation in a fast and efficient manner. This, however, relies on the availability of the wavelet coefficients. In contrast, our proposed algorithm can be computed easily from any reasonable representation of primitive variables on disk. It should also be noted that other approaches in variational settings [26,27] have been used to

handle both grid non-uniformity and varying numbers of spatial points through time. The present study addresses the same challenges, but in an approach that uses the redundancies created by nearest-neighbor interpolation of AMR data to achieve an efficient implementation.

In Section 2, we outline the matrix operations associated with POD, as well as provide a discussion of why we consider data interpolated to a uniform grid using a nearest-neighbor algorithm. From there, we provide an overview of the new algorithm in Section 3, with details of the algorithm provided in the appendix. We quantify improvements in computational efficiency in Section 4 by counting operations and tracking processing times. Finally, we conclude with a summary and additional remarks on future work in Section 6.

2. Background: proper orthogonal decomposition

The POD technique extracts dominant spatio-temporal features in a flow by computing an orthogonal set of basis functions, or POD modes, based on time series of velocity fields or other quantitative flow data. These modes are optimal in the sense that no alternative set of orthogonal basis functions can capture more of the variance in the chosen field using the same number of modes, or fewer [10]. Typically, POD in turbulent flows is performed using fluctuating velocities due to their connection with the turbulence kinetic energy, although thermodynamic variables can also be included in the decomposition when they are dynamically relevant [28]. Temporal coefficients are computed by projecting flow-field data onto the spatial modes.

In the following description of POD, we distinguish various forms of matrices and snapshots by using a boldface upper-case symbol with no subscripts to denote the full matrix, a boldface upper-case symbol with a single subscript to denote the corresponding data in the form of the original data at the mode number or time indicated by the subscript, and a lower-case boldface symbol for the corresponding vector form of these matrices. For example, the matrix containing all POD modes is $\mathbf{\Phi}$, the fourth POD mode with the same shape as the original data is denoted $\mathbf{\Phi}_4$, and the corresponding data reshaped into a vector is denoted $\boldsymbol{\phi}_4$. The only exception to this formalism is the snapshot matrix where the corresponding matrices (in the same order as above) are $\mathbf{X}$, $\mathbf{U}_i$, and $\boldsymbol{u}_i$ to remain consistent with notation used in prior studies. To refer to a particular element of a vector or matrix, we do not use boldface for the symbol and the subscripts refer to that element of the vector or matrix. For example, the element in the i^{th} row and j^{th} column of $\mathbf{X}$ is given as X_{ij}.

2.1. Snapshot POD

In this study, the algorithm for efficient POD of AMR data is based on the widely-used snapshot POD method first proposed by Sirovich [29]. The first step in this method is to form the snapshot matrix $\mathbf{X}$, in which each column contains the discrete spatial solution $\boldsymbol{u}_j$ for each instant j in time (denoted a 'snapshot'). The total number of spatial points, N_s, in each snapshot corresponds to the number of rows in $\mathbf{X}$, and the number of time steps, N_t, corresponds to the number of columns. The snapshot matrix is generally formed by taking two- or three-dimensional (2D or 3D, respectively) flow-field data, $\mathbf{U}_j$, reshaping this data into a column vector $\boldsymbol{u}_j$ of length N_s, and then assigning this vector to the appropriate column of $\mathbf{X}$. Note that the subscript j corresponds here to instances in time, as opposed to velocity components. However, an arbitrary number of flow-field variables can be used in the computation by 'stacking' the variables in the same column of $\mathbf{X}$ (the appendix in Taira et al. [9] provides a more detailed description of this formulation). Although the more efficient algorithm outlined here is presented for a single variable to improve clarity, it can easily be generalized to account for multiple variables. Finally, as in the standard snapshot POD method, we assume that each spatial location carries the same weight, as is the case when AMR grids are interpolated to uniform grids for post-processing (as described in more detail in Section 2.2).

After the snapshot matrix $\mathbf{X}$ has been constructed from the flow-field data, the next step in the snapshot POD method is to compute the covariance matrix $\mathbf{R}$ as

$$\mathbf{R} = \mathbf{X}^\text{T}\mathbf{X}. \tag{1}$$

The eigenvalues and eigenvectors of $\mathbf{R}$ are then computed by solving the eigenvalue problem

$$\mathbf{R}\mathbf{\Psi} = \mathbf{\Lambda}\mathbf{\Psi}, \tag{2}$$

where $\mathbf{\Psi}$ is a matrix containing the eigenvectors $\boldsymbol{\psi}_n$ of $\mathbf{R}$, and $\mathbf{\Lambda}$ is a diagonal matrix containing the corresponding eigenvalues λ_n. The orthonormal spatial modes are calculated as

$$\mathbf{\Phi} = \mathbf{X}\mathbf{\Psi}\mathbf{\Lambda}^{-1/2}. \tag{3}$$

Each eigenvalue, λ_n, corresponds to the amount of variance, or energy, contained in each spatial mode, $\boldsymbol{\phi}_n$. Additionally, these eigenvalues and modes are ordered such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Finally, temporal coefficients, $\boldsymbol{a}_n$, corresponding to each spatial mode, $\boldsymbol{\phi}_n$, are calculated by

$$\mathbf{A} = \mathbf{X}^\text{T}\mathbf{\Phi}, \tag{4}$$

where **A** is the resulting combination of temporal coefficients for modes **Φ** and constitutes the outcome of the POD procedure. Given the sorting of the eigenvalues, the modes are ordered such that the lower modes account for the greatest variance in the data.

To summarize the snapshot POD method, there are five primary steps: (*i*) forming the snapshot matrix **X**, (*ii*) computing the covariance **R**, (*iii*) computing the eigenvalues and eigenvectors of **R**, (*iv*) computing the spatial modes **Φ**, and (*v*) computing the temporal coefficients **A**. These are the steps that will be addressed in creating the more efficient POD algorithm for AMR data.

2.2. Computing snapshot POD on AMR data

Two challenges arise when performing the snapshot POD method on block-structured AMR data: (*i*) the non-uniformity of the grid and (*ii*) the unequal number of spatial points, represented by the total number of computational grid cells N_s, as a function of time in the simulation. The former challenge can be solved by changing the definition of the inner product in the equations in Section 2.1 to weight cells appropriately (i.e., **R** is computed as $R_{ij} = \langle \boldsymbol{u}_i, \boldsymbol{u}_j \rangle$, where $\langle \cdot, \cdot \rangle$ is the inner product). However, the latter challenge is not as easily addressed because the number of spatial points varies as a function of time, causing **X** to have a "ragged" bottom edge due to the time dependence of N_s, making it impossible to perform any true matrix operation without zero padding or another data insertion approach. Additionally, for dynamically evolving simulations, the context of each spatial location would be lost as the grid evolves, causing ambiguity when reshaping the flow-field data into each column of **X**. In general, the most common workaround is to simply interpolate the data with sufficiently high-order to one fixed, well-resolved (possibly non-uniform) mesh [13,14]. Other approaches, such as those developed in Refs. [26,27], can perform POD directly on the AMR data, and here we additionally focus on developing an efficient implementation for modern HPC architectures.

An even simpler procedure to overcome these issues is to interpolate the data using a nearest-neighbor algorithm to a uniform grid with a resolution equivalent to the finest grid cell size in the simulation. This method gives **X** with dimensions $N_s \times N_t$, where $N_s = N_x N_y N_z$ is the number of spatial points in 3D, or $N_s = N_x N_y$ in 2D. Here, N_x, N_y, and N_z represent the number of grid cells in each direction of the static uniform grid with resolution equivalent to that used at the finest AMR level. This method overcomes the challenges mentioned above because: (*i*) interpolating to a uniform grid ensures no ragged bottom edge in **X**, since all snapshots are interpolated to the same number of grid points and (*ii*) nearest-neighbor interpolation naturally gives additional weighting to coarser cells by repeating the value of that cell; the number of repetitions in each dimension is equal to the ratio of coarse to fine cell sizes. It is assumed that despite the inaccuracies incurred by using a low-order nearest-neighbor interpolation, the first several POD modes will not be substantially affected because POD is targeted at understanding large-scale coherent structures; however, it is important to confirm this for different data sets. Additionally, as will be discussed in Section 5, this approach is amenable to an efficient and scalable implementation.

In the context of POD, a provocative opportunity emerges when using nearest-neighbor interpolation on AMR data. Namely, there are several steps in the snapshot POD method where operations are performed on repeated data. As the computational domain size and number of AMR levels increase, these repeated calculations can occupy an increasingly large portion of the overall time taken for the snapshot POD method, limiting the ability to apply POD to long time series, large data sets, and rapid development of reduced-order models for design and control purposes. However, we can also take advantage of this repetition, and in this study we present a new algorithm that leverages the repetition of nearest-neighbor-interpolated AMR data to reduce the computational cost of snapshot POD.

3. Methods: efficient algorithm to compute POD on AMR datasets

In the following, we describe each of the steps used in the efficient snapshot POD algorithm for nearest-neighbor-interpolated AMR data; these steps correspond to the five primary steps used in the traditional snapshot POD method summarized at the end of Section 2.1. We use the terms "standard" and "AMR" to refer to standard matrix multiplication and the newly developed algorithm, respectively. Each step in POD presents a unique arrangement of repeated computations that require different tactics to minimize the repetition; hence, we outline the improvements for each step of snapshot POD individually. It is likely that, for some datasets, there is no computational advantage to using the improved algorithm for certain steps of the POD method, so presenting the steps individually will allow users to decide whether a standard technique or the AMR algorithm would be more efficient for a particular step. It should also be noted that we present the algorithms in this section independent of computer language and choice of compiler, and a discussion of quantitative improvements in algorithm speed is provided later in Section 4.

Before continuing, we require additional notation beyond that used for the description of standard snapshot POD. A companion matrix $\mathbf{X}_{\text{grid}}$ of identical size to **X** after nearest-neighbor interpolation is used to identify the grid level at the corresponding location in **X**. The coarsest grid level (sometimes called the "base" grid) is denoted by $\ell = 0$ and the finest grid level is denoted by $\ell = f$, such that $\ell \in [0, f]$ and there are $f + 1$ total grid levels. The matrix $\mathbf{X}_{\text{grid}}$ thus consists of the finest AMR grid level ℓ used for the nearest-neighbor interpolation at each spatial location in the simulation domain. We can then identify how many times a cell is repeated at a given level using $\mathbf{X}_{\text{grid}}$ and the dimensionality of the POD computation. We define c_ℓ to be the number of repeated cells at AMR grid level ℓ resulting from the nearest-neighbor

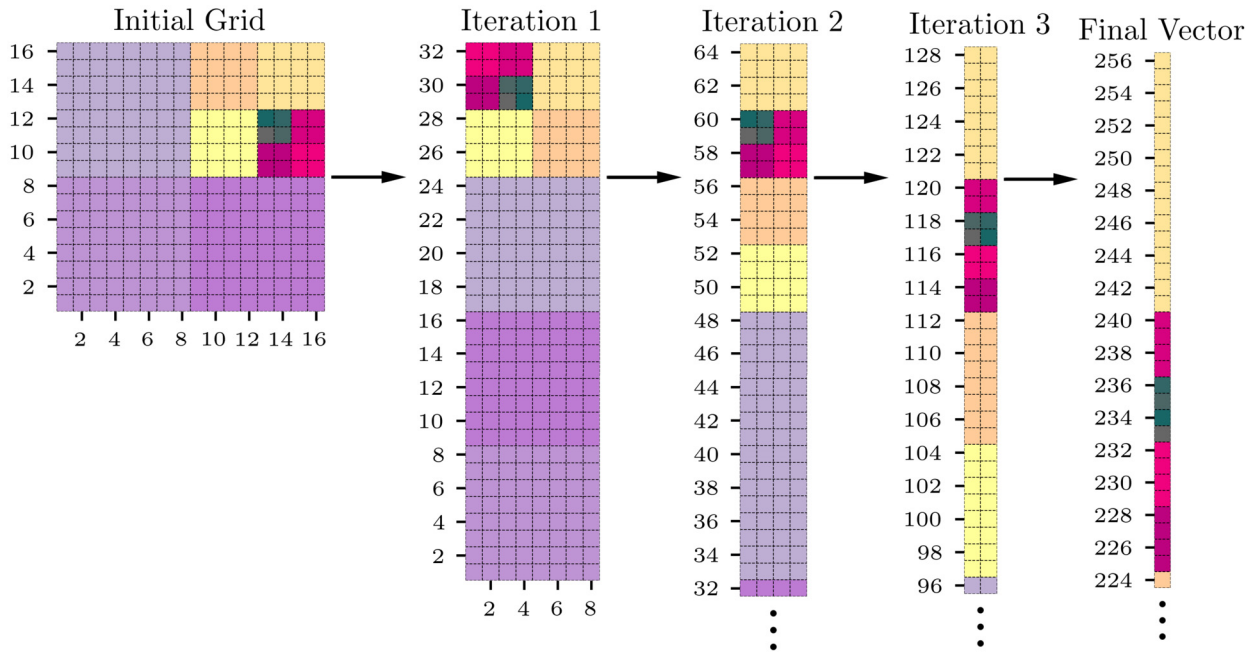

Fig. 1. Illustration of the iterative reshaping procedure given in Algorithm 1 of the appendix with an initial grid of $N_x = N_y = 16$, $d = 2$ and $f = 3$. Shades of purple indicate $\ell = 0$; shades of yellow indicate $\ell = 1$; shades of pink indicate $\ell = 2$; and shades of gray indicate $\ell = 3$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

interpolation of coarse grid levels to the finest grid. For a simulation with total number of dimensions d (e.g., $d = 2$ in 2D and $d = 3$ in 3D), the total number of repeated cells is then c_ℓ^d.

Below, we present the algorithms as generally as possible, but we do make a few simplifications to improve the clarity of the presentation. In particular, for all AMR levels, we assume a refinement ratio of 2, meaning that a coarse cell is always split in half along each spatial dimension for the next finer resolution. As a result, $c_\ell = 2^{f-\ell}$ gives the total number of repeated cells in one dimension at AMR grid level ℓ. Additionally, we only consider POD with one variable, even though it is typical to use more than one variable (for example, the three components of the velocity vector). It should be noted, however, that the algorithmic improvements outlined here do not depend on these simplifications, and the new algorithm can be easily extended to account for different refinement ratios or additional variables.

3.1. Forming the snapshot matrix **X**

As described in Section 2.1, the snapshot matrix $\mathbf{X}$ is formed by reshaping the solution values of each snapshot, $\mathbf{U}_j$, into a column vector $\boldsymbol{u}_j$ and storing these in the j^{th} column of X_{ij}, where the rows of $\mathbf{X}$ correspond to the N_s spatial locations. The order of the elements in a particular column of $\mathbf{X}$ is irrelevant for snapshot POD as long as a particular row of $\mathbf{X}$ corresponds to the same physical location for all times. We take advantage of this property by proposing a new procedure that converts each snapshot $\mathbf{U}_j$ to a column $\boldsymbol{u}_j$ such that cells that have identical values (as a result of the nearest-neighbor interpolation) are contiguous in the final snapshot vector $\boldsymbol{u}_j$. The more common approach would be simply constructing $\boldsymbol{u}_j$ with the same order as the elements in $\mathbf{U}_j$ that are contiguous in memory (this is the normal procedure used by many built-in reshaping routines). Either approach is valid since they both adhere to rows of $\mathbf{X}$ corresponding to the same physical location. However, the new procedure allows us to more easily weight and skip repeated operations in subsequent POD steps.

To construct $\mathbf{X}$ in this manner, we first reshape $\mathbf{U}_j$ over $f + 1$ iterations, where f is the finest AMR grid level. For a single iteration, this procedure consists of various calls to reshape and permutation routines such that, after a single iteration, repeated values are contiguous in the matrix, but not memory, until the last iteration. In 2D after each iteration, the matrix has been reshaped into a more elongated matrix with width c_ℓ while in 3D, there is one elongated direction and two dimensions with equal widths of c_ℓ. This procedure visually makes the matrix appear as if blocks are being re-shaped; consequently, we call this a "blockwise" re-shaping procedure.

An illustration of this procedure is shown in Fig. 1, and the detailed algorithm is provided in the appendix as Algorithm 1. The final result is a one-dimensional (1D) column vector, $\boldsymbol{u}_j$, of length N_s with contiguous repeated values (since vectors are 1D, this also implies contiguous in memory) that can be inserted into columns of $\mathbf{X}$. The same procedure is also applied to the companion matrix $\mathbf{X}_{\text{grid}}$.

It should be noted that this procedure is sensitive to how the code stores matrix values in memory, since the permutation and reshaping must be performed considering the memory layout to retain the contiguity of cells. Therefore, we provide this reshaping methodology for both row- and column-major programming languages in Algorithm 1 of the appendix.

3.2. Computing the covariance matrix **R**

Each element of the covariance matrix $\mathbf{R}$ is an inner product of snapshots defined as $R_{ij} = \langle \boldsymbol{u}_i, \boldsymbol{u}_j \rangle = \sum_k X_{ki} X_{kj}$. If $X_{\text{grid},ki} < f$ and $X_{\text{grid},kj} < f$, then we know there must be $c^d_{\ell_{\max}}$ repeated computations, where $\ell_{\max} = \max\left[X_{\text{grid},ki}, X_{\text{grid},kj}\right]$ for a given k. We can eliminate this repetition by instead weighting a single computation by the number of repeated operations, $c^d_{\ell_{\max}}$, and then skipping the next $c^d_{\ell_{\max}} - 1$ operations. We can accomplish this in a straightforward manner because the blockwise reshaping technique described in Section 3.1 forces all cells with the same value to be contiguous in the final 1D column vector.

To compute $R_{ij} = \sum_k X_{ki} X_{kj}$, we start by initializing a scalar variable for storage, $r_{\text{sum}} = 0$, that will contain the final value of the i, j component of $\mathbf{R}$. Iterating along the spatial dimension, k, we first determine $\ell_{\max}$. If $i = j$, we simply use $\ell_{\max} = X_{\text{grid},ki}$. Next, if $\ell_{\max} = f$, we add $X_{ki} X_{kj}$ to r_{sum} for k to $k + c^d_{\ell_{\max}} - 1$. Otherwise, we add $X_{ki} X_{kj}$ to r_{sum} weighted by the number of repeated operations $c^d_{\ell_{\max}}$, then skip the next $c^d_{\ell_{\max}} - 1$ operations. After iterating over all k, we assign r_{sum} to R_{ij}. This sequence is only performed for lower triangular elements of $\mathbf{R}$, with the symmetry of $\mathbf{R}$ used to assign the remaining off-diagonal components of $\mathbf{R}$ as $R_{ji} = R_{ij}$. The detailed algorithm is given as Algorithm 2 in the appendix.

The one peculiarity in this algorithm is that if $\ell_{\max} = f$, not only do we add an unweighted $X_{ki} X_{kj}$ to r_{sum}, but we also immediately know that the next $c^d_{f-1} - 1$ computations do not need to be weighted. We thus use an unweighted version because it is always true that $c^d_f = 1$, independent of the numeric values for f and d. Additionally, due to the nature of AMR, there is a minimum of c^d_{f-1} cells at $\ell_{\max} = f$ that must be grouped together. This can be seen in Fig. 1, where the smallest possible grouping of cells at $\ell = f$ must be the same size as the groupings for a cell at $\ell = f - 1$; in that example, the smallest possible grouping is 4.

To illustrate this algorithm further, consider a 1D simulation with two levels of AMR (i.e., $d = 1$ and $f = 2$), with elements of $\mathbf{X}$ and $\mathbf{X}_{\text{grid}}$ defined, respectively, in this example as

$$\mathbf{X} \equiv \begin{bmatrix} X_{11} & X_{12} & X_{13} & \cdots \\ X_{21} & X_{22} & X_{23} & \cdots \\ X_{31} & X_{32} & X_{33} & \cdots \\ X_{41} & X_{42} & X_{43} & \cdots \\ X_{51} & X_{52} & X_{53} & \cdots \\ X_{61} & X_{62} & X_{63} & \cdots \\ X_{71} & X_{72} & X_{73} & \cdots \\ X_{81} & X_{82} & X_{83} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}, \qquad \mathbf{X}_{\text{grid}} \equiv \begin{bmatrix} 0 & 1 & 2 & \cdots \\ 0 & 1 & 2 & \cdots \\ 0 & 1 & 1 & \cdots \\ 0 & 1 & 1 & \cdots \\ 1 & 1 & 0 & \cdots \\ 1 & 1 & 0 & \cdots \\ 2 & 2 & 0 & \cdots \\ 2 & 2 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{5}$$

Comparing the computation of a diagonal element of $\mathbf{R}$, such as R_{11}, using standard matrix multiplication and the new AMR algorithm, we obtain

$$R^{\text{standard}}_{11} = X^2_{11} + X^2_{21} + X^2_{31} + X^2_{41} + X^2_{51} + X^2_{61} + X^2_{71} + X^2_{81} + \cdots, \tag{6}$$

$$R^{\text{AMR}}_{11} = 4X^2_{11} + 2X^2_{51} + X^2_{71} + X^2_{81} + \cdots. \tag{7}$$

The computation of an off-diagonal element, such as R_{32}, is given by

$$R^{\text{standard}}_{32} = X_{13}X_{12} + X_{23}X_{22} + X_{33}X_{32} + X_{43}X_{42} + X_{53}X_{52} + \cdots, \tag{8}$$

$$R^{\text{AMR}}_{32} = X_{13}X_{12} + X_{23}X_{22} + 2X_{33}X_{32} + 2X_{53}X_{52} + \cdots. \tag{9}$$

These two different elements of $\mathbf{R}$ highlight how the computations can be weighted to reduce the total number operations. There is, however, additional computational overhead associated with checking the grid level for the AMR algorithm. Specifically, for R_{11}, we need to check the grid level before computing the first three terms, but not before the fourth term; for R_{32}, we need to check the grid level before computing the first, third, and fourth terms, but not the second term. The computational overhead associated with checking the grid level is generally smaller compared to the redundant additions and multiplications required in the standard approach, but this issue is nevertheless explored in more detail in Section 4.

3.3. Computing eigenvalues and eigenvectors of the covariance matrix **R**

We next consider the computation of the eigenvalues λ_n and eigenvectors $\boldsymbol{\psi}_n$ of $\mathbf{R}$. Each element of $\mathbf{R}$ corresponds to the dot product of snapshots at different times. However, because there is no spatial dependence, only temporal dependence,

there is no way to take advantage of the spatial repetitions due to AMR. It is important to note that, in many AMR codes, there is sub-cycling in time that could lead to a variable temporal output at different AMR levels; however, in practice, extracting information during sub-cycling is not advised because the data are often incorrect until a correction/synchronization step has been performed when all levels of AMR are at the same instant in time. Therefore, we do not consider POD on simulations with adaptive temporal resolution. However, the outer timestep between snapshots is allowed to vary, for example due to a maximum Courant-Friedrichs-Levy (CFL) condition.

3.4. Computing the POD spatial modes $\mathbf{\Phi}$

Accelerating the computation of the POD spatial modes $\mathbf{\Phi}$ generated using nearest-neighbor-interpolated AMR datasets poses a different challenge than the computation of $\mathbf{R}$. In particular, the computation of $\mathbf{\Phi}$ is difficult to accelerate because the first step in the computation of a single element of $\mathbf{\Phi}$ is a dot product of a row of $\mathbf{X}$ with a column of $\mathbf{\Psi}$, both of which are vectors varying in time, and thus contain no repeated values from nearest-neighbor interpolation. After this step, no computations can be eliminated when multiplying $\mathbf{X\Psi}$ by $\mathbf{\Lambda}^{-1/2}$, since $\mathbf{\Lambda}$ is a diagonal matrix.

Despite these challenges, however, there are at least two ways in which the computation of $\mathbf{\Phi}$ can be accelerated when performing POD on nearest-neighbor-interpolated data from AMR simulations. This is simplest to illustrate through an example. Consider a small simulation with $d = 1$, $f = 2$, $N_s = 8$, and $N_t = 4$, where $\mathbf{X}$ is defined as in Eq. (5) and $\mathbf{X}_{\text{grid}}$, $\mathbf{\Psi}$, and $\mathbf{\Lambda}$ are defined, respectively, as

$$\mathbf{X}_{\text{grid}} \equiv \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 0 & 2 \\ 2 & 2 & 0 & 2 \\ 2 & 2 & 0 & 2 \end{bmatrix}, \qquad \mathbf{\Psi} \equiv \begin{bmatrix} \Psi_{11} & \Psi_{12} & \Psi_{13} & \Psi_{14} \\ \Psi_{21} & \Psi_{22} & \Psi_{23} & \Psi_{24} \\ \Psi_{31} & \Psi_{32} & \Psi_{33} & \Psi_{34} \\ \Psi_{41} & \Psi_{42} & \Psi_{43} & \Psi_{44} \end{bmatrix}, \qquad \mathbf{\Lambda} \equiv \begin{bmatrix} \Lambda_{11} & 0 & 0 & 0 \\ 0 & \Lambda_{22} & 0 & 0 \\ 0 & 0 & \Lambda_{33} & 0 \\ 0 & 0 & 0 & \Lambda_{44} \end{bmatrix}. \tag{10}$$

The element Φ_{11} is computed as

$$\Phi_{11} = (X_{11}\Psi_{11} + X_{12}\Psi_{21} + X_{13}\Psi_{31} + X_{14}\Psi_{41})\Lambda_{11}^{-1/2}. \tag{11}$$

None of the terms in this expression for Φ_{11} are redundant, regardless of the grid level for each value of X_{ij}. Now consider the computation of the element Φ_{21}, which is given by

$$\Phi_{21} = (X_{21}\Psi_{11} + X_{22}\Psi_{21} + X_{23}\Psi_{31} + X_{24}\Psi_{41})\Lambda_{11}^{-1/2}. \tag{12}$$

This computation does not contain any redundancies within itself. However, since $X_{\text{grid},11} = 0$, this implies that $X_{11} = X_{21} = X_{31} = X_{41}$, which then implies that $X_{11}\Psi_{11} = X_{21}\Psi_{11}$. Therefore, the first term in computing Φ_{21} is a redundant computation since it was already computed for Φ_{11}. In fact, since $X_{\text{grid},1i} < f$ for every i, we find that $\Phi_{11} = \Phi_{21}$, making the entire computation of Φ_{21} redundant.

The redundancy of the individual terms that contribute to the computation of an element $\mathbf{\Phi}$ is exploited in the following sections to accelerate the computation of $\mathbf{\Phi}$. To this end, we propose two different algorithms that are advantageous for different AMR patterns.

3.4.1. Method 1 for computing $\mathbf{\Phi}$

The first method for computing $\mathbf{\Phi}$, denoted "method 1," takes advantage of the fact that if a particular spatial location (i.e., a row of $\mathbf{X}$) never reaches the finest AMR level at any time, the elements of $\mathbf{\Phi}$ at that spatial location must be equal, such that we can entirely eliminate the computation of the redundant elements of $\mathbf{\Phi}$. The advantages of this approach are that it is simple to identify where repetitions occur and it requires very little additional computational cost compared to standard matrix multiplication. Simulations with AMR that are relatively static in time, such as many shear flow problems (e.g., jets, wakes, and plumes) where outer boundary regions are not refined, can easily leverage the repetition. However, if the simulation is dynamically evolving, for example in the case of a propagating flame, and all spatial locations are refined to the finest level for any of the snapshots, this algorithm will not yield any computational improvement.

The general outline to compute $\mathbf{\Phi}$ using method 1 is as follows. We initialize a vector $\boldsymbol{g}$ of length c_0^d. The maximum grid level that occurs in the first c_0^d rows of $\mathbf{X}_{\text{grid}}$ is stored in $\boldsymbol{g}$; in other words, $g_i = \max_j(X_{\text{grid},ij})$. We then compute the first c_0^d rows of the first column of Φ_{i1}. If $g_i < f$, we know that the next $c_{g_i}^d - 1$ elements in the column j of Φ_{ij} are identical to Φ_{ij}. So, after computing $\Phi_{ij} = \sum_k X_{ik}\Psi_{kj}\Lambda_{jj}^{-1/2}$, this value is immediately assigned to the next $c_{g_i}^d - 1$ elements in the column j without repeating the computation. This sequence is then repeated for the remaining columns of $\mathbf{\Phi}$. After these remaining values are computed, we consider the next c_0^d rows of $\mathbf{X}$. This is repeated until all spatial locations are computed. The detailed algorithm for method 1 is given in the appendix as Algorithm 3.

As a demonstration of the method, consider $\mathbf{X}_{\text{grid}}$, $\mathbf{\Psi}$, and $\mathbf{\Lambda}$ from Eq. (10). We first compute $\boldsymbol{g}$ to be $g_i = [1; 1; 1; 1]$ then compute the first element of $\mathbf{\Phi}$, which is given in Eq. (11). We then find that $g_1 = 1 < f$, implying that Φ_{11} is repeated $c^d_{g_1} = 1$ additional times. Therefore, we assign $\Phi_{21} = \Phi_{11}$. Continuing, Φ_{31} is computed analogously to Eq. (11) and, since $g_3 = 1 < f$, we assign $\Phi_{41} = \Phi_{31}$. We next compute the first $c^d_0 = 4$ rows of the second column of $\mathbf{\Phi}$ using the same values for $\boldsymbol{g}$. This is then also done for the third and fourth columns of $\mathbf{\Phi}$. Afterwards, we move to the next $c^d_0 = 4$ rows of $\mathbf{X}$, recomputing $\boldsymbol{g}$ as $g_i = [2, 2, 2, 2]$. Since all elements of $\boldsymbol{g}$ are equal to f, there are no repeated elements in these rows of $\mathbf{\Phi}$, and each element in rows 5-8 of $\mathbf{\Phi}$ must be computed individually.

There is additional computational overhead compared to standard matrix operations due to the computation of $\boldsymbol{g}$ and the need to check which elements of $\mathbf{\Phi}$ are repeated based on $\boldsymbol{g}$. Although the cost of these steps is, in general, small compared to the full computation of $\mathbf{\Phi}$, in order to achieve a computational acceleration we require regions of the flowfield where $\ell < f$ for all of time. This constraint motivates the second method for computing $\mathbf{\Phi}$.

3.4.2. Method 2 for computing $\mathbf{\Phi}$

We now consider a second method, denoted "method 2," to reduce redundant computations in the calculation of $\mathbf{\Phi}$. When considering the individual terms that contribute to a single element of $\mathbf{\Phi}$, such as the terms to compute Φ_{11} in Eq. (11), if the grid level associated with the term to compute the $\mathbf{X}$ element is less than f, then the value of that term is identical to the terms for a different element of $\mathbf{\Phi}$.

For example, in Eq. (10), the grid level associated with the first term, $X_{11}\Psi_{11}$, in Eq. (11) is one, which is less than f; therefore, in computing Φ_{21}, instead of repeating the multiplication $X_{11}\Psi_{11}$, we could have stored that value and simply added it to the last three terms in Eq. (12). This idea of storing individual contributions to a particular element of $\mathbf{\Phi}$ is the basic premise behind method 2. The advantage of this approach is that most of the repeated computations will be avoided, even if the mesh is dynamically changing from snapshot to snapshot. However, this method also requires more overhead than that either method 1 or standard matrix operations, due to the increased storage requirements and the cost of the search algorithm used to determine the grid levels.

Overall, the method 2 algorithm is quite complex, so we refer the reader to Algorithm 4 in the appendix for the complete methodology. A rough outline of the algorithm is as follows. We will consider the computation of c^d_0 rows of $\mathbf{\Phi}$ at a time, just as we did with method 1. First, we aggregate cell locations of $\mathbf{X}$ corresponding to the level in the first c^d_0 rows and store the indices in an initially empty matrix $\mathbf{G}$. We next initialize an additional matrix $\mathbf{H}$ of size $N_s \times (f+1)$ filled with zeros. The columns here correspond to the contributions of $\Phi_{ij} = \sum_k X_{ik}\Psi_{kj}\Lambda_{jj}^{-1/2}$ for a single grid level value (i.e., column 1 corresponds to $\ell = 0$ contributions, column 2 corresponds to $\ell = 1$ contributions, etc.). This computation is only performed for unique elements of $\mathbf{H}$. So, for example, if H_{11} is unique then H_{21}, H_{31}, $\cdots$ are not unique because they are equal to H_{11}. Values in $\mathbf{H}$ that are not unique (i.e., the elements of $\mathbf{H}$ that are identical to already computed elements of $\mathbf{H}$) are then filled by the unique value. Finally, we sum rows of $\mathbf{H}$ and divide by the appropriate value of $\mathbf{\Lambda}$, which directly corresponds to values of the appropriate elements of $\mathbf{\Phi}$. This process is repeated over groups of c^d_0 rows for all N_s. Overall, this method requires more computational resources than either the standard or method 1 approaches to computing $\mathbf{\Phi}$, due to the need to tabulate cells in $\mathbf{G}$ and to store an additional matrix $\mathbf{H}$, but this method can still be advantageous for dynamically evolving simulations.

As a demonstration of the performance gains enabled by method 2, consider again the example in Eq. (10). We first initialize the empty matrix $\mathbf{G}$, which has size $(f+1) \times c^d_1 \times N_t = 3 \times 2 \times 4$. We then store the indices for $\ell = 0$ elements, then $\ell = 1$ elements, and finally $\ell = 2$ elements for the first c^d_0 rows of $\mathbf{X}$; for this example, this leads $\mathbf{G}$ to be

$$G_{1jk} = \begin{bmatrix} 1 & 4 & - & - \\ 1 & 4 & - & - \end{bmatrix}, \qquad G_{2jk} = \begin{bmatrix} 2 & 3 & - & - \\ 2 & 3 & - & - \end{bmatrix}, \qquad G_{3jk} = \begin{bmatrix} - & - & - & - \\ - & - & - & - \end{bmatrix}, \tag{13}$$

where dashes are empty values of $\mathbf{G}$. The second dimension of $\mathbf{G}$ only needs to be of length c^d_1 because the smallest grouping of cells at the grid level is c^d_{f-1}, as discussed in Section 3.2, so we only need length c^d_1 to track the grid level (although the minimum grouping of $\ell = f$ cells is c^d_{f-1}, these are not identical values in general due to the use of AMR).

Using $\mathbf{G}$, we next compute $\mathbf{H}$ of size $c^d_0 \times n_\ell$ for unique cells then assign the remaining non-unique cells, giving

$$H_{ij} = \begin{bmatrix} X_{11}\Psi_{11} + X_{14}\Psi_{41} & X_{12}\Psi_{21} + X_{13}\Psi_{31} & 0 \\ \downarrow & \downarrow & 0 \\ \downarrow & X_{32}\Psi_{21} + X_{33}\Psi_{31} & 0 \\ \downarrow & \downarrow & 0 \end{bmatrix}, \tag{14}$$

where the down arrow represents a value that was filled from the unique computation above it. The column of Φ_{i1} is computed by summing over rows of $\mathbf{H}$ and dividing by the square root of the appropriate diagonal element of Λ

$$\Phi_{i1} = \left[\sum_j (H_{ij})\right] \Lambda_{11}^{-1/2}. \tag{15}$$

This process is then repeated for the remaining columns of $\mathbf{\Phi}$. For this, $\mathbf{G}$ does not need to be re-tabulated, but $\mathbf{H}$ does need to be recomputed. After all columns of $\mathbf{\Phi}$ are computed, we then move to rows 5-8 of $\mathbf{\Phi}$. More briefly, for these rows, $\mathbf{G}$ would be

$$G_{1jk} = \begin{bmatrix} 3 & - & - & - \\ 3 & - & - & - \end{bmatrix}, \qquad G_{2jk} = \begin{bmatrix} 1 & 2 & - & - \\ - & - & - & - \end{bmatrix}, \qquad G_{3jk} = \begin{bmatrix} 4 & - & - & - \\ 1 & 2 & 4 & - \end{bmatrix}. \tag{16}$$

Using this, we compute $\mathbf{H}$ for the first column of $\mathbf{\Phi}$, which is

$$H_{ij} = \begin{bmatrix} X_{53}\Psi_{31} & X_{51}\Psi_{11} + X_{52}\Psi_{21} & X_{54}\Psi_{41} \\ \downarrow & \downarrow & X_{64}\Psi_{41} \\ \downarrow & 0 & X_{71}\Psi_{11} + X_{72}\Psi_{21} + X_{74}\Psi_{41} \\ \downarrow & 0 & X_{81}\Psi_{11} + X_{82}\Psi_{21} + X_{84}\Psi_{41} \end{bmatrix}. \tag{17}$$

Elements of $\mathbf{\Phi}$ can be computed using $\mathbf{H}$ analogous to the computation of Φ_{i1} above. This process is again repeated for the remaining columns of $\mathbf{\Phi}$.

3.5. Computing the POD temporal coefficients $\mathbf{A}$

Elements in $\mathbf{A}$ are calculated as inner products of snapshots and POD spatial modes, namely $A_{ij} = \langle \boldsymbol{u}_i, \boldsymbol{\phi}_j \rangle$. Similar to the calculation of $\mathbf{R}$, repeated computations for this inner product only occur if the maximum level between $\boldsymbol{u}_i$ and $\boldsymbol{\phi}_j$ at a given spatial location is less than the finest level. As is discussed in Section 3.4, repetitions in $\boldsymbol{\phi}_j$ can only occur if a spatial location is never refined to the finest level for all N_t (method 1 of computing $\mathbf{\Phi}$ takes advantage of this repetition). Therefore, to determine whether there will be any repeated computations, we simply need to determine the maximum grid level reached at a particular spatial location for all snapshots. This makes the more efficient algorithm for computing $\mathbf{A}$ fairly straightforward to implement.

The general procedure is as follows, with the detailed algorithm provided as Algorithm 6 in the appendix. We first determine the maximum grid level occurring over all N_t for a particular spatial location i and store this in a vector $\boldsymbol{g}$ (specifically, $g_i = \max_j(X_{\mathrm{grid},ij})$). The grid level at g_i is then converted to a weight as $g_i = c^d_{g_i}$. For each element A_{jk}, we then follow a similar procedure to that outlined for the computation of $\mathbf{R}$ in Section 3.2. We initialize a scalar a_sum, then iterating along the spatial location i, we identify the number of repeated computations using g_i for the operation $X_{ij}\Phi_{ik}$. If $g_i = 1$, this implies $g_i = f$ and we perform the next c^d_{f-1} operations without weighting the contribution. The motivation for not weighting the next c^d_{f-1} computations is the same as that described for the computation of $\mathbf{R}$ in Section 3.2. Otherwise, we weight $X_{ij}\Phi_{ik}$ by g_i, and skip the next $c^d_\ell - 1$ contributions. The contributions (weighted or unweighted) are cumulatively added to a_sum. After iterating over all N_s, we assign a_sum to the appropriate element of $\mathbf{A}$, then repeat over the remaining elements of $\mathbf{A}$, without repeating the computation of $\boldsymbol{g}$.

Using the example from Eq. (10), as well as the terminology in Eq. (5) for $\mathbf{X}$ and an analogous definition for $\mathbf{\Phi}$, we illustrate the algorithm. First, we compute $g_i = [1; 1; 1; 1; 2; 2; 2; 2]$, which is then converted to the number of repetitions $g_i = [2; 2; 2; 2; 1; 1; 1; 1]$. We then compute each element of $\mathbf{A}$ using $\boldsymbol{g}$ for the weighting. Element A_{11} would be computed as

$$A_{11} = 2X_{11}\Phi_{11} + 2X_{31}\Phi_{31} + X_{51}\Phi_{51} + X_{61}\Phi_{61} + X_{71}\Phi_{71} + X_{81}\Phi_{81}\,. \tag{18}$$

The weightings for the first and second terms are simply g_1 and g_3, respectively. This is then repeated for other elements of $\mathbf{A}$ without recomputing g_i. As with the computation of $\mathbf{R}$, there is additional computational overhead associated with computing the weighting vector $\boldsymbol{g}$ and checking the grid level during the inner product. Specifically, for A_{11}, we need to check the grid level before computing the first, second, third, and fifth terms, but not before the fourth and sixth terms. In the following section describing the algorithm performance, we explore the trade-off between the additional computational overhead and the removal of repeated calculations.

After performing the computation of the POD temporal coefficients $\mathbf{A}$, we must undo the reshaping procedure described in Section 3.1 to put cells in $\mathbf{\Phi}$ in their original physical locations. This can be done by a procedure similar to that outlined in Section 3.1, with the main differences being that each iteration shrinks the elongated dimension and the last iteration puts the grid into the original dimensions. The detailed algorithm is provided in the appendix as Algorithm 5.

4. Results and discussion: algorithm performance

4.1. Performance metrics

To quantify the performance of the algorithm outlined in Section 3, as compared to the snapshot POD approach using standard matrix operations, we will use two measures of the computational cost, T: the number of primitive operations (denoted 'ops') and the CPU time (denoted 'CPU'). We use primitive operation counting to eliminate uncertainties in the coding language, compiler, processing speed, environment, etc; this is known as a random access machine (RAM) model, or

Table 1
Several examples demonstrating how we count primitive operations.

Code	n_{arth}	n_{log}	n_{acc}	n_{asn}	n_{fun}	n_{tot}
$r_{\text{sum}} \leftarrow r_{\text{sum}} + c_{\text{val}} * \mathbf{X}[k,i] * \mathbf{X}[k,j]$	3	0	2	1	0	6
$\mathbf{A} \leftarrow \text{empty}(N_{\text{t}}, N_{\text{t}})$	0	0	0	1	1	2
for $i \leftarrow 1$ **to** n **do**	n	0	0	n	0	$2n$
$i \leftarrow 0$; **while** $i < n$ **do** $\{i \leftarrow i+1\}$	n	n	0	$n+1$	0	$3n+1$
$\mathbf{\Phi}[m+i,n] \leftarrow h_{\text{sum}} * (\lambda[n])^{-1/2}$	3	0	2	1	1	7

a RAM algorithm [30]. In Section 4.2, we will use this model on synthetic AMR data to unambiguously quantify performance gains of the new algorithm as a function of the proportion of the domain refined at different AMR levels and the length of the time series, N_{t}. In Section 4.3, we use both the RAM model and CPU time as metrics to show that the new algorithm is faster than the snapshot POD method with standard matrix operations for genuine AMR data from a simulation of a buoyant plume. All of the code used to compute the computational cost is publicly available at https://github.com/tesla-cu/amrPOD. The operation counting code is written in python in order to provide a user-friendly environment, and the CPU timing code is written in Fortran 90 [31].

With respect to operation counting, the primitive operations we consider are arithmetic operations (e.g., $a-b$, $a*b$, a/b, a^2), logical operations (e.g., $a<b$, $a \geq b$, $a == b$), accessing operations (e.g., $\boldsymbol{a}[i]$, $\mathbf{X}[i,j]$), assignment operations (e.g., $a \leftarrow b$), and function calls (e.g., $f(a)$); the numbers of each operation are denoted n_{arth}, n_{log}, n_{acc}, n_{asn}, and n_{fun}, respectively. The total number of operations is then given by $n_{\text{tot}} = n_{\text{arth}} + n_{\text{log}} + n_{\text{acc}} + n_{\text{asn}} + n_{\text{fun}}$, where we equally weight each of the operations in computing n_{tot}. The decision to use equal weightings is based on previous studies [32] that approximate the computational effort involved in each of these operations as taking $\mathcal{O}(1)$ time. This decision has its limitations, especially for the deep memory hierarchy in modern HPC systems, but identifying the exact coefficients is a research topic in itself. Rather, we ignore any constant coefficients associated with the time complexity of individual operations and, for simplicity, we count them as single units. These coefficient values can be easily changed using the code provided at https://github.com/tesla-cu/amrPOD. In Table 1, we provide several examples of pseudocode and a demonstration of how operations are counted.

For more complicated programmatic structures such as `for` loops, we reduce each structure to a series of primitive operations. For example, we deconstruct `for` loops into an arithmetic operation used to increment the iterative variable and an assignment operation used to assign the new incremented value to the iterative variable. We deconstruct `while` loops into a `for` loop coupled with an `if` statement, where the `if` statement serves as the logical check to determine whether iteration within the `for` loop should stop. Accessing operations such as those seen in the third example of Table 1 are treated as unit operations regardless of the dimension or size of the data being accessed.

It should be noted before continuing that we are not attempting to change the order $\mathcal{O}(N_{\text{t}}^2)$ of the POD computation. Rather, we are attempting to show that we reduce the leading constant of the order of the computation, which we deem to be a faster algorithm practically. Since we are not changing the order, it is not obvious whether we have improved the computation or not with just one metric. Hence, we use two metrics, primitive operation count and CPU time, to demonstrate the computational advantages of the new algorithm.

Lastly, we do not discuss the performance in terms of error between the standard and AMR algorithms. While error is an important metric to consider, especially for techniques that rely on statistical properties such as the randomized singular value decomposition (rSVD) [33], the presented algorithms are only affected by finite precision arithmetic errors. Using double precision (as we use herein) leads to an error in terms of the relative maximum absolute difference in the covariance matrix $\mathbf{R}$ of error($\mathbf{R}$) $\approx 10^{-14} - 10^{-16}$. Again using this error metric, we see that the error is a little larger for $\mathbf{\Phi}$ and $\mathbf{A}$, roughly error($\mathbf{\Phi}$) $\approx 10^{-8} - 10^{-10}$ and error($\mathbf{A}$) $\approx 10^{-10} - 10^{-12}$. However, the larger relative errors are exclusively on the less energetic modes while the more energetic modes have an error roughly equivalent to $\mathbf{R}$. We consider these errors sufficiently low for most purposes.

4.2. Tests using synthetic AMR data

We first test the algorithm outlined in Section 3 on synthetically generated AMR data that is intended to mimic, in a parametrically controlled fashion, data from genuine AMR simulations. To generate this data, we randomly create different AMR grids for specified values of the proportion of the domain refined at each level, with no temporal correlation between the grids in successive snapshots. The purpose of these tests is to parametrically control the redundancy of the data in each snapshot and to subsequently demonstrate the efficiency of the new algorithm for a "worst case" scenario where there is no temporal correlation between snapshots. We consider this the "worst case" scenario because, in a typical AMR simulation, grid refinement is performed to capture particular features in the flow, which are naturally correlated from snapshot to snapshot. In some flows (e.g., turbulent jets), large quiescent regions also result in persistent coarse grids at the same locations in successive snapshots.

Subsequent to these tests, we then introduce temporally correlated grid refinements where a persistent static grid is imposed in part of the domain across all snapshots. As will be seen in the following sections, these correlated refinement regions actually make our algorithm *more* efficient. Moreover, the computation of $\mathbf{A}$ and method 1 for computing $\mathbf{\Phi}$ require

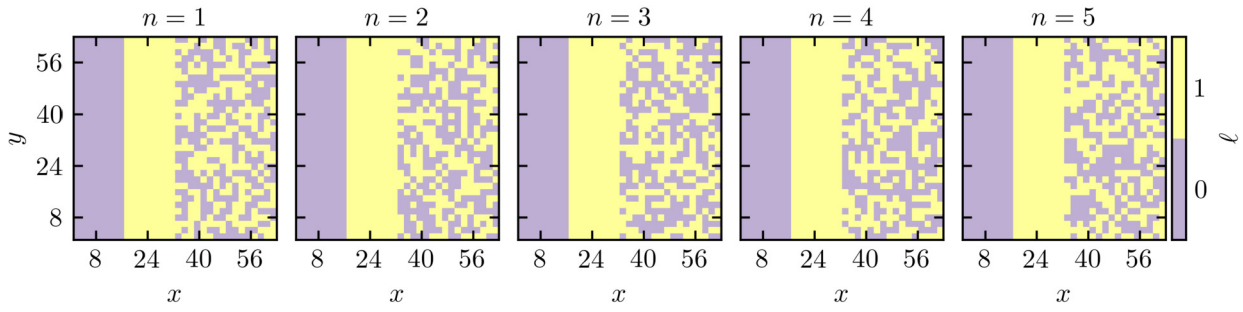

Fig. 2. Five snapshots of synthetically generated AMR data with $N_x = N_y = 64$, $d = 2$, and $f = 1$ for $p_0 = p_1 = 1/2$ and $\tilde{p}_0^{\max} = \tilde{p}_1^{\max} = 1/4$. This data is generated by assigning $\ell = 0 \,\forall\, x \in [1, 16]$, $y \in [1, 64]$ and $\ell = 1 \,\forall\, x \in [17, 32]$, $y \in [1, 64]$. The remaining portion of the domain is randomly refined according to the desired values of p_ℓ. Note that $1/4 \leq p_0^{\max} \leq 1/2$, and $1/2 \leq p_1^{\max} \leq 3/4$, $\lim_{N_t \to \infty} p_0^{\max} = 1/4$, and $\lim_{N_t \to \infty} p_1^{\max} = 3/4$.

regions of $\ell < f$ for all N_t to have any practical advantage; this motivates the inclusion of the statically refined regions for some snapshot sequences.

In the following synthetic AMR tests, we will separately examine the algorithmic speed-ups for the computations of **R**, **Φ** and **A**. This approach will allow us to assess each step of the snapshot POD approach individually, enabling a user to choose which steps to accelerate as a result of the specific properties of their AMR dataset. For example, with a highly dynamic mesh, it will likely be fastest to use only the AMR algorithms for **R** and **Φ** (specifically, method 2 for the calculation of **Φ**) and to use regular matrix operations for **A**. Here we quantify the algorithmic improvements using ratios of the run-time for the new AMR algorithm over the run-time for the standard approach, denoted $\overline{T}_{\text{AMR}}/\overline{T}_{\text{std}}$, where $\overline{(\cdot)}$ is the run-time averaged over the number of samples, n_{samp}, tested for those parameters. Using this approach, the new AMR algorithm is determined to be faster than the standard algorithm when $\overline{T}_{\text{AMR}}/\overline{T}_{\text{std}} < 1$ or $\log_{10}(\overline{T}_{\text{AMR}}/\overline{T}_{\text{std}}) < 0$. Note that, we do not include the reshaping cost in the calculation since this procedure becomes negligible in cost as compared to other operations with increasing N_t.

4.2.1. Parameterization of the synthetic AMR data

We specify two new quantities to parameterize the synthetic AMR data. The first measures the proportion of the domain for each snapshot that is at a particular grid level ℓ, as given by

$$p_\ell(n) = \frac{1}{N_s} \sum_{i=1}^{N_s} \delta(\ell - X_{\text{grid},in}) \,, \tag{19}$$

where $n = 1, \ldots, N_t$ denotes the snapshot number, N_s is the total number of spatial locations, δ is the Dirac measure, and $p_\ell \in [0, 1]$ for any n. The second quantity measures the proportion of the domain that has reached a specified maximum level of refinement ℓ within any of the snapshots in the data record; this quantity is defined as

$$p_\ell^{\max} = \frac{1}{N_s} \sum_{i=1}^{N_s} \delta \left[\ell - \max_n (X_{\text{grid},in}) \right] , \tag{20}$$

where $p_\ell^{\max} \in [0, 1]$. Note that $\sum_{\ell=0}^{f} p_\ell = \sum_{\ell=0}^{f} p_\ell^{\max} = 1$ and, in the limit as $N_t \to \infty$, a grid that is entirely randomly refined will have $p_\ell^{\max} \to 0$ for $\ell < f$ and $p_\ell^{\max} \to 1$ for $\ell = f$.

In the synthetically generated AMR data, we are able to exactly control p_ℓ from snapshot to snapshot, and this is one of the parameters that we will vary in the following analysis. In order to generate data where $p_\ell^{\max} > 0$ for $\ell < f$ when $N_t \to \infty$, we set regions of constant ℓ for all of time, denoted by $\tilde{p}_\ell^{\max}$. For example, in Fig. 2, we set $\tilde{p}_0^{\max} = \tilde{p}_1^{\max} = 1/4$ by setting $\ell = 0 \,\forall\, x \in [1, 16]$, $y \in [1, 64]$ and $\ell = 1 \,\forall\, x \in [17, 32]$, $y \in [1, 64]$. The remaining portion of the domain is randomly refined. This results in $\lim_{N_t \to \infty} p_0^{\max} = 1/4$ and $\lim_{N_t \to \infty} p_1^{\max} = 3/4$. Note that, in general, $p_\ell = p_\ell(n)$ varies in time, and in Section 4.3 where we test the algorithm on genuine AMR data we will instead report the average value $\langle p_\ell \rangle = (1/N_t) \sum_{n=1}^{N_t} p_\ell(n)$.

4.2.2. Synthetic AMR test results: fully random grids

We first examine the performance of the new algorithm for different values of p_1 and N_t with refinement up to one AMR level, $f = 1$, using synthetic AMR data where every location in the domain is randomly refined (that is, there are no statically refined regions, $\tilde{p}_0^{\max} = \tilde{p}_1^{\max} = 0$). The synthetic data is two dimensional ($d = 2$) with $N_x = N_y = 64$ and $N_s = 64^2$. Both p_1 and N_t are varied to span the parameter space, and we obtain ensemble statistics for the run-time calculations by testing $N_{\text{samp}} = 64$ independent randomly generated datasets for each value of p_1 and N_t.

The top row of Fig. 3 shows the resulting parameter spaces of $\overline{T}_{\text{AMR}}/T_{\text{std}}$ for the calculations of **R**, **Φ** using methods 1 and 2, and **A**, where T is a measure of the total operation count n_{tot}. For each of the snapshot POD steps in Fig. 3, as $N_t \to \infty$,

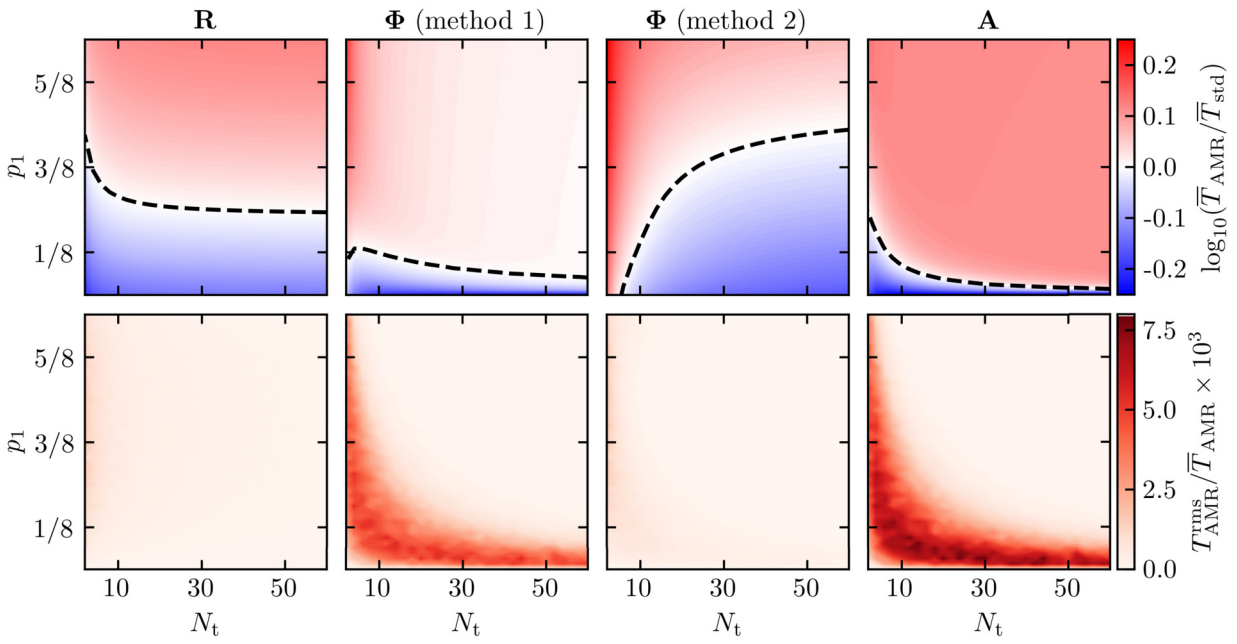

Fig. 3. Top row: Run-time ratios $\log_{10}(\overline{T}_{AMR}/T_{std})$, where T corresponds to the total number of operations n_{tot}, as functions of p_1 and N_t for the computations of **R**, **Φ** (methods 1 and 2), and **A** (left to right) using randomly refined synthetic AMR data with $N_x = N_y = 64$, $N_s = 64^2$, $d = 2$ and $f = 1$. The ratios are computed over an ensemble with $N_{samp} = 64$ samples. Shaded blue regions indicate parameters where the new AMR algorithm is faster than the standard algorithm, and shaded red regions are the inverse; the black dashed line shows the boundary between these two regions. Bottom row: Root-mean square fluctuations in the sampled values of T with respect to the average $\overline{T}$ for each step in the snapshot POD method.

the boundary defining the transition between $\overline{T}_{AMR}/T_{std} < 1$ (i.e., the new algorithm is faster, indicated by blue regions in the figure) and $\overline{T}_{AMR}/T_{std} > 1$ (i.e., the standard algorithm is faster, indicated by red regions) asymptotically approaches specific values of p_1. This is to be expected because the new algorithm includes additional fixed costs as compared to the standard algorithm, such as pre-computing p_ℓ^m in the calculation of both **Φ** and **A**. These additional costs can be dominant for small N_t but become less relevant as N_t increases, resulting in the asymptotic behaviors of the algorithm efficiency boundary (i.e., the $\overline{T}_{AMR}/\overline{T}_{std} = 1$ contour in Fig. 3).

Fig. 3 shows that the efficiency boundary approaches $p_1 = 0$ for both method 1 of computing **Φ** and in the calculation of **A**. This occurs because both of these algorithms require $\sum_{\ell=0}^{f-1} p_\ell^{max} > 0$ in order to produce any computational advantage. However, with randomly refined grids, as $N_t \to \infty$, $p_f^{max} \to 1$ and $\sum_{\ell=0}^{f-1} p_\ell^{max} \to 0$, and there is thus no computational advantage using the AMR algorithm for these two steps of the snapshot POD method.

By contrast, Fig. 3 shows that the efficiency boundaries for the calculations of **R** and **Φ** using method 2 approach nonzero values of p_1. For **R**, this occurs because the primary computational advantage of the new algorithm is realized when the grid levels at k in $R_{ij} = X_{ki}X_{kj}$ for both i and j are smaller than f. For the method 2 calculation of **Φ**, the asymptotic approach to nonzero p_1 occurs because cells are first tabulated according to grid level before computing any element of Φ_{ij}, which removes any dependence in time of repeated computations.

The bottom row of Fig. 3 shows how much fluctuation there is in the ensemble samples, measured by the root-mean-square, $T^{rms} = (T'^2)^{1/2}$ where $T' = T - \overline{T}$, relative to the average number of operations. The new algorithms for computing both **R** and **Φ** (method 2) show very little fluctuation between samples. This is for similar reasons as mentioned above; there is a computational advantage primarily as a result of p_ℓ, not p_ℓ^{max}. The other two algorithms, however, rely primarily on p_ℓ^m, which changes as N_t increases. Specifically, the probability of p_0^{max} at a specific spatial location for randomly refined grids, as is the case here, is $p_0^{max} = (p_0)^{N_t} \to 0$ since $p_0 < 1$ when $p_1 > 0$.

The regions of relatively large fluctuations in the computations of **Φ** (method 1) and **A** in the bottom row of Fig. 3 occur because p_0^{max} can vary substantially between samples, thus giving larger T^{rms}. However, even in these regions, the fluctuations are still relatively small in comparison to the average number of operations (note the multiplier of 10^3 on the colorbar). Given these results, we thus only report the mean of the samples in subsequent sections since there is very little deviation from the mean between samples.

Finally, we use the fully randomized synthetic AMR data to examine the role of the data dimensionality, d. Fig. 4 shows the same parameter space as in Fig. 3, for data with $d = 3$, $N_x = N_y = N_z = 16$ and $N_s = 16^3$ (note that we also use fewer samples, N_{samp}, for $d = 3$). The trends for $d = 3$ are similar to those for $d = 2$, except for changes in the asymptotic values of p_1 for the efficiency boundaries in the algorithms for **R** and **Φ** (method 2). For a higher dimension d, there are more repeated values for a given coarse cell than for smaller d, thus giving more repeated computations that can be skipped.

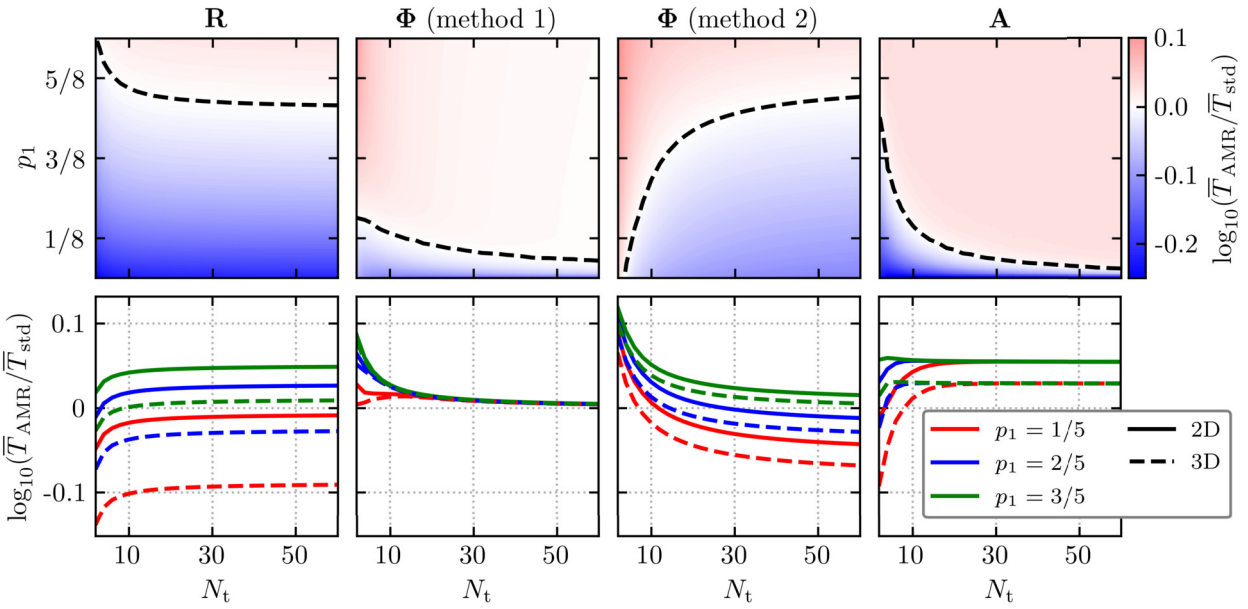

Fig. 4. Top row: Run-time ratios $\log_{10}(\overline{T}_{\mathrm{AMR}}/\overline{T}_{\mathrm{std}})$, where T corresponds to the total number of operations n_{tot}, as functions of p_1 and N_t for the computations of **R**, **Φ** (methods 1 and 2), and **A** (left to right) using randomly refined synthetic AMR data with $N_x = N_y = N_z = 16$, $N_s = 16^3$, $d = 3$ and $f = 1$. The ratios are computed over an ensemble with $N_{\mathrm{samp}} = 8$ samples. Colors are the same as described in the caption of Fig. 3. Bottom row: Comparison between the $d = 2$ results from Fig. 3 and the $d = 3$ results in the top row for three values of p_1.

On the bottom row of Fig. 4, the ratio $\overline{T}_{\mathrm{AMR}}/\overline{T}_{\mathrm{std}}$ is consistently smaller for all N_t and the three chosen values of p_1 for **R** and **Φ** (method 2), indicating that the new AMR algorithms become increasingly efficient, as compared to the standard algorithms, as the dimensionality increases.

4.2.3. Synthetic AMR test results: mixed static and random grids

From the results in Section 4.2.2, it may appear that under no circumstance do the new algorithms for computing **Φ** (method 1) and **A** have any computational advantage in the asymptotic limit as $N_t \to \infty$. This is certainly true when the entire computational domain is randomly refined at each instance in time and $p_\ell^{\max} \approx 0$ for $\ell < f$. However, in most AMR simulations of practical flows, we generally find that $p_\ell^{\max} > 0$ for $\ell < f$ and, for these conditions, there can be a substantial computational advantage when using the new AMR algorithms. We demonstrate this improvement in Fig. 5, where we examine run-times for different values of $\tilde{p}_0^{\max}$ and $\tilde{p}_1^{\max}$ using synthetic AMR data with $p_0 = p_1 = 1/2$, $N_x = N_y = 64$, $N_s = 64^2$, $N_t = 50$, $d = 2$, and $f = 1$. The resulting data includes regions of both static and random grid refinement, as shown for example in Fig. 2 for $\tilde{p}_0^{\max} = \tilde{p}_1^{\max} = 1/4$.

Fig. 5 shows that the run-time improvements in the calculations of **Φ** (method 1) and **A** depend almost exclusively on the choice of $\tilde{p}_0^{\max}$, due primarily to the fact that these algorithms can only take advantage of repeated computations if $p_0^{\max} > 0$. In the region of the domain that is randomly refined, it is very likely that all spatial locations will be tagged with $\ell = f = 1$ in this case. For example, if $p_0 = p_1 = 1/2$, $N_t = 50$, and we fix $\tilde{p}_0^{\max} = \tilde{p}_1^{\max} = 1/4$, the probability of having $\ell = 0$ for all times at a given spatial location that is randomly refined is

$$\left[\frac{1 - (p_1 - \tilde{p}_1^{\max}) - \sum_{\ell=0}^{f} \tilde{p}_\ell^{\max}}{1 - \sum_{\ell=0}^{f} \tilde{p}_\ell^{\max}}\right]^{N_t} = \left(\frac{1}{2}\right)^{50} \approx 10^{-17}. \tag{21}$$

Thus, for many of the cases, the regions of random refinement are likely to be all tagged with $\ell = 1$ at some instant in time, giving $p_1^{\max} \approx 1 - \tilde{p}_0^{\max}$, and varying $\tilde{p}_1^{\max}$ has essentially no impact since $\tilde{p}_0^{\max}$ primarily dictates $p_1^{\max}$. Note that the thin blue regions at the top of the parameter spaces for **Φ** (method 1) and **A** in Fig. 5 are due to the fact that setting $\tilde{p}_1^{\max} = p_1 = 1/2$ necessarily means that $p_0 = p_0^{\max} = 1/2$, thus providing repeated computations.

We can also use Fig. 5 to understand how the choice of $\tilde{p}_\ell^{\max}$ affects the calculations of **R** and **Φ** (method 2). For **R**, Fig. 5 shows that setting $\tilde{p}_\ell^{\max}$ for any ℓ increases the correlation between cells where $\ell = 0$, in turn increasing the number of repeated computations. There is approximate reflection symmetry about $\tilde{p}_0^{\max} = \tilde{p}_1^{\max}$ because setting either above zero necessarily increases correlations between cells where $\ell = 0$. For **Φ** (method 2), there is very little change in the run-time ratio when varying $p_\ell^{\max}$, since cells are first tabulated according to grid level before computing any element in **Φ**, removing all dependence on time. Variations in the run-time ratio would only come from $f = 0$ for any ℓ for a spatial location, which only changes the performance of the algorithm marginally by not computing contributions for that ℓ in that location.

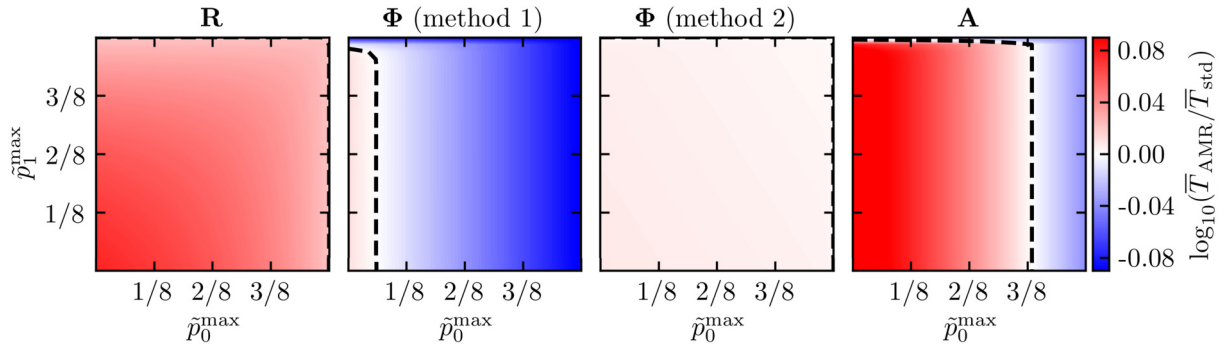

Fig. 5. Run-time ratios $\log_{10}(\overline{T}_{\mathrm{AMR}}/\overline{T}_{\mathrm{std}})$, where T corresponds to the total number of operations n_{tot}, as functions of $\tilde{p}_0^{\max}$ and $\tilde{p}_1^{\max}$ for the computations of **R**, **Φ** (methods 1 and 2), and **A** (left to right) using synthetic AMR data with both static and random refinement regions and simulation parameters $p_0 = p_1 = 1/2$, $N_x = N_y = 64$, $N_{\mathrm{s}} = 64^2$, $N_{\mathrm{t}} = 50$, $d = 2$, and $f = 1$ using $N_{\mathrm{samp}} = 8$. The shading and colors are identical to those in Figs. 3 and 4.

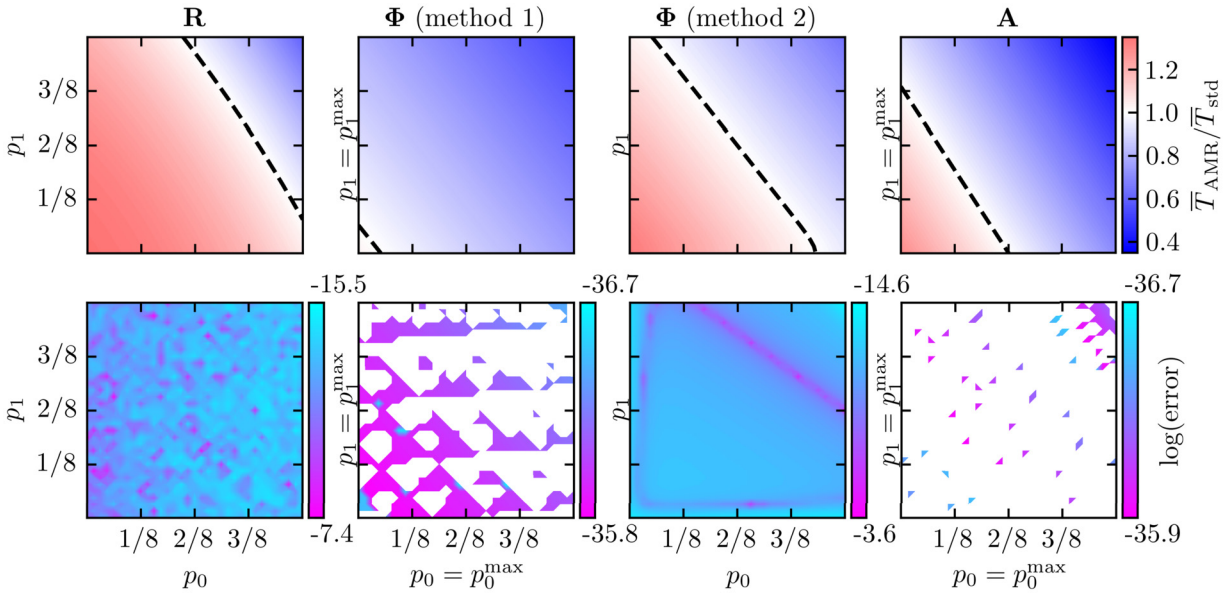

Fig. 6. Run-time ratios $\overline{T}_{\mathrm{AMR}}/\overline{T}_{\mathrm{std}}$, where T corresponds to the total number of operations n_{tot}, as functions of p_0 and p_1 (and implicitly p_2) for **R** and **Φ** (method 2), and as functions of $\tilde{p}_0^{\max} = p_0$ and $\tilde{p}_1^{\max} = p_1$ (and implicitly $\tilde{p}_2^{\max} = p_2$) for **Φ** (method 1) and **A**. The analysis is performed on synthetic AMR data with both static and random refinement regions and simulation parameters $N_x = N_y = 64$, $N_{\mathrm{s}} = 64^2$, $N_{\mathrm{t}} = 50$, $d = 2$ and $f = 2$ using $N_{\mathrm{samp}} = 8$. Bottom row: Relative error between the data and the proposed fit in Table 2. The numbers on the top and bottom of each colorbar represent the bounds of the colorbar. Pink shading corresponds to a higher relative error for a given computation, and white regions indicate where the error is identically zero for double precision.

4.2.4. Synthetic AMR test results: two levels of AMR

To this point, we have only examined synthetic AMR data with one level of AMR (i.e., $f = 1$), where it is generally easier to assess the effects of different parameter choices on the performance of the new algorithms. In practical AMR simulations, however, it is common to use several levels of AMR in order to drive the computational cost down without significant loss of accuracy. It is therefore of interest to demonstrate the performance of the new algorithms for more than one level of AMR

We first consider the performance of the algorithms when $f = 2$. The top row of Fig. 6 shows $\overline{T}_{\mathrm{AMR}}/T_{\mathrm{std}}$ for synthetic AMR data with both static and random refinement using simulation parameters $N_x = N_y = 64$, $N_{\mathrm{s}} = 64^2$, $N_{\mathrm{t}} = 50$, $d = 2$, and $f = 2$, with $N_{\mathrm{samp}} = 8$. Here we only consider how p_ℓ affects the calculations of **R** and **Φ** (method 2), and how $p_\ell^{\max}$ affects **Φ** (method 1) and **A**. Note that setting $\tilde{p}_0^{\max} = p_0$ implies $\tilde{p}_0^{\max} = p_0^{\max}$. Overall, Fig. 6 shows that, in comparison to the results using synthetic AMR data with $f = 1$, there is a significant improvement in the speed of the snapshot POD method when using the new algorithms for data with $f = 2$, because coarser cells have more repeated computations.

While Fig. 6 provides a good sense of the parameter space for $f = 2$, we still do not have a complete description of the performance of the algorithm because in Fig. 6, we are only visualizing two-dimensional slices of three-dimensional spaces. We would like to understand the performance more generally in a way that *does* provide as complete description as

Table 2
Fitted coefficients for B_{ij} and b_i from Eq. (22) using $d = 2$, $N_s = 64^2$, $N_t = 50$, $N_{samp} = 8$, $f = 2$, $p_0 = k/16$, $p_1 = m/16$ and $p_2 = 1 - p_0 - p_1$, where $k, m \in [0, 8]$ are integers, for $\mathbf{R}$ and $\mathbf{\Phi}$ (method 2). The same values are used for $\mathbf{\Phi}$ (method 1) and $\mathbf{A}$, but we also set $\tilde{p}_\ell^{\max} = p_\ell$. Quantities after $\pm$ are two standard deviation errors of the fitted parameters as provided by the `scipy` curve fitting toolbox.

R (p_ℓ)		
$B_{00} = 0.19 \pm 1.7 \times 10^{-4}$	$B_{01} = 0.74 \pm 1.2 \times 10^{-4}$	$B_{02} = 1.33 \pm 6.6 \times 10^{-5}$
$B_{11} = 0.74 \pm 1.7 \times 10^{-4}$	$B_{12} = 1.34 \pm 6.6 \times 10^{-5}$	$B_{22} = 1.35 \pm 2.8 \times 10^{-5}$
$\mathbf{\Phi}$ (method 1) $(\tilde{p}_\ell^{\max} = p_\ell)$	$\mathbf{\Phi}$ (method 2) (p_ℓ)	$\mathbf{A}$ $(\tilde{p}_\ell^{\max} = p_\ell)$
$b_0 = 0.47 \pm 2.7 \times 10^{-17}$	$b_0 = 0.56 \pm 8.2 \times 10^{-4}$	$b_0 = 0.14 \pm 2.2 \times 10^{-17}$
$b_1 = 0.58 \pm 2.7 \times 10^{-17}$	$b_1 = 0.73 \pm 8.2 \times 10^{-4}$	$b_1 = 0.53 \pm 2.2 \times 10^{-17}$
$b_2 = 1.03 \pm 1.3 \times 10^{-17}$	$b_2 = 1.35 \pm 3.9 \times 10^{-4}$	$b_2 = 1.29 \pm 1.1 \times 10^{-17}$

possible such that detailed parameters sweeps are not necessary. In order to do so, we propose the following functions to closely approximate the performance of the new algorithms

$$\mathbf{R}: \sum_{i=0}^{f}\sum_{j=0}^{f} B_{ij} p_i p_j, \qquad \mathbf{\Phi}\ \text{(method 1) and}\ \mathbf{A}: \sum_{i=0}^{f} b_i p_i^{\max}, \qquad \mathbf{\Phi}\ \text{(method 2)}\ : \sum_{i=0}^{f} b_i p_i, \tag{22}$$

where B_{ij} and b_i are fitting parameters. Note the symmetry $p_i p_j = p_j p_i$; hence, there are only $(f+1) + (f+1)f/2$ values to fit for $\mathbf{R}$, rather than $(f+1)^2$.

The functions in Eq. (22) make intuitive sense for the following reasons. Each element of R_{ij} is computed by the inner product of two snapshots, and the resulting number of repetitions is based on the maximum grid level between the two snapshots at a given spatial location. So, for $\mathbf{R}$, we use two summations because the number of repetitions is dependent upon the composition of each of the two snapshots. For $\mathbf{\Phi}$ (method 1) and $\mathbf{A}$, only one pattern of AMR matters in how many repeated computations there are: $p_\ell^{\max}$, for reasons given in Section 3. Since each of these operations are linear operations, only one summation is required. Finally, only one of the matrices (i.e., $\mathbf{X}$) in the computation of $\mathbf{\Phi}$ (method 2) has repeated values from the AMR, and this computation method is essentially independent of $p_\ell^{\max}$ (as shown in Fig. 5), so we can get a good approximation of performance using one summation and p_ℓ.

We use non-linear least squares to fit the coefficients B_{ij} and b_i in Eq. (22) to the data $\overline{T}_{AMR}/T_{std}$ shown in the top row of Fig. 6. These coefficients are given in Table 2 along with 95% confident intervals. Using these values of B_{ij} and b_i, we then estimate the performance of our algorithm for the parameters in the top row of Fig. 6, and in the bottom row of this figure we show the relative error between the computed performance (top row) and the estimated performance.

The bottom row of Fig. 6 shows that for $\mathbf{R}$, $\mathbf{\Phi}$ (method 1), and $\mathbf{A}$, our proposed functions provide excellent agreement with the computed performance. For $\mathbf{R}$, the estimate is only off by a maximum of $\approx 10^{-7.6}$ and the variations in the error are fairly random throughout the parameter space, indicating a good fit. For $\mathbf{\Phi}$ (method 1) and $\mathbf{A}$, the estimate is essentially exact, with a maximum error of $\approx 10^{-36}$ and many areas of white, where the estimate was exactly what was computed to double precision. This is to be expected because setting $p_\ell^{\max} = p_\ell$ removes any statistical variation from snapshot to snapshot in a simulation; since these are linear operations, it would be appropriate that a linear approximation would fit exactly. Even if there was a statistical variation, this would only affect the pre-computation of the maximum grid level, which would not substantially impact the fit.

By contrast, the estimate for $\mathbf{\Phi}$ (method 2) does not appear to fit the data as closely, as indicated by the distinct triangular feature in the error plot shown in the bottom row of Fig. 6. This feature is the result of a nuance in the code that avoids computing the contribution of a level ℓ if that ℓ does not appear in that spatial location for all of time. The pink regions for $\mathbf{\Phi}$ (method 2) are where there is intermittent behavior of ℓ not appearing for all spatial locations. For example, consider $p_0 = 3/8$, $p_1 = 1/32$, and $p_2 = 19/32$, which is one of parameters shown in Fig. 6. The probability of a spatial location containing at least one instance of ℓ for a given N_t is $1 - (1 - p_\ell)^{N_t}$. For this situation, the probability of having $\ell = 0$, $\ell = 1$, and $\ell = 2$ for a given spatial location would be $\approx 1 - 10^{-11}$, ≈ 0.8, and ≈ 1, respectively. Thus, there will be many instances of spatial locations that will not have $\ell = 1$ for all times, providing a change in the number of operation counts that cannot be accounted for by the proposed linear fit. Regions of high error are where the probability of having an ℓ is not close to 0 or 1. Nonetheless, even with this imperfection in the function, the estimate still has very small error, with a maximum of $\approx 10^{-3.5}$ for the entire parameter space.

4.2.5. Synthetic AMR test results: generalizations for additional levels of AMR

In addition to providing a good fit for data with $d = 2$ and $f = 2$, we can identify more general trends in the coefficients to better understand the algorithm performance with additional levels of AMR. The B_{ij} in Table 2 for the computation of $\mathbf{R}$ show that some of these values are approximately equal, namely $B_{02} \approx B_{12} \approx B_{22}$ and $B_{01} \approx B_{11}$. From previous discussions, the number of repeated computations is determined by the maximum grid level at a spatial location between the two snapshots. Since B_{02}, B_{12}, and B_{22} account for the computational advantage between cells with $\ell = 0$ and $\ell = 2$,

Table 3
Fitted coefficients for B_{ij} and b_i from Eq. (22) using $d = 3$, $N_s = 16^2 \times 32$, $N_t = 50$, $N_{samp} = 8$, $f = 3$, $p_0 = k/16$, $p_1 = m/16$, $p_2 = n/16$ and $p_3 = 1 - p_0 - p_1 - p_2$, where $k, m, n \in [0, 5]$ are integers, for **R** and **Φ** (method 2). The same values are used for **Φ** (method 1) and **A**, but we also set $\tilde{p}_\ell^{\max} = p_\ell$. Quantities after $\pm$ are two standard deviation errors of the fitted parameters as provided by the `scipy` curve fitting toolbox.

R (p_ℓ)		
$B_{00} = 0.0054 \pm 2.5 \times 10^{-3}$	$B_{01} = 0.045 \pm 1.3 \times 10^{-3}$	$B_{02} = 0.37 \pm 1.3 \times 10^{-3}$
$B_{03} = 1.16 \pm 5.6 \times 10^{-4}$	$B_{11} = 0.046 \pm 2.5 \times 10^{-3}$	$B_{12} = 0.37 \pm 1.3 \times 10^{-3}$
$B_{13} = 1.15 \pm 5.6 \times 10^{-4}$	$B_{22} = 0.37 \pm 2.5 \times 10^{-3}$	$B_{23} = 1.16 \pm 5.6 \times 10^{-4}$
$B_{33} = 1.18 \pm 1.7 \times 10^{-4}$		
Φ (method 1) ($\tilde{p}_\ell^{\max} = p_\ell$)	**Φ** (method 2) (p_ℓ)	**A** ($\tilde{p}_\ell^{\max} = p_\ell$)
$b_0 = 0.44 \pm 8.1 \times 10^{-17}$	$b_0 = 0.53 \pm 4.8 \times 10^{-3}$	$b_0 = 0.0044 \pm 6.4 \times 10^{-17}$
$b_1 = 0.44 \pm 8.1 \times 10^{-17}$	$b_1 = 0.54 \pm 4.8 \times 10^{-3}$	$b_1 = 0.034 \pm 6.4 \times 10^{-17}$
$b_2 = 0.51 \pm 8.1 \times 10^{-17}$	$b_2 = 0.64 \pm 4.8 \times 10^{-3}$	$b_2 = 0.27 \pm 6.4 \times 10^{-17}$
$b_3 = 1.03 \pm 2.7 \times 10^{-17}$	$b_3 = 1.34 \pm 1.6 \times 10^{-3}$	$b_3 = 1.14 \pm 2.1 \times 10^{-17}$

$\ell = 1$ and $\ell = 2$, and two $\ell = 2$ cells, respectively, there is approximately the same computational advantage between all three since the finest level in all three is $\ell = 2$. Similar reasoning leads to the conclusion that $B_{01} \approx B_{11}$. More generally, we would expect that

$$B_{ij} \approx B_{\ell\ell} \qquad \text{where} \qquad \ell = \max(i, j). \tag{23}$$

Note that $B_{ff} > 1$ because there is no computational advantage as a result of the AMR.

For $\ell < f$, there is a clear pattern that emerges in the coefficients $B_{\ell\ell}$ and b_ℓ for computing **R** and **A**, respectively. In these coefficients for the fit for **A**, we can see that

$$\frac{B_{\ell\ell}}{B_{(\ell-1)(\ell-1)}} \text{ or } \frac{b_\ell}{b_{\ell-1}} \approx 2^d \qquad \text{for} \qquad 1 \leq \ell < f\,. \tag{24}$$

This occurs because, if there is any repetition as a result of the AMR, that contribution to R_{ij} or A_{ij} is weighted and skipped in the exact same manner if $\ell < f$. Since we assume a refinement ratio of 2, there are 2^d additional repetitions for each successive level of refinement, which is also clearly indicated by c_ℓ^d. This pattern is not present between $\ell = f$ and $\ell = f - 1$ because we do not weight and skip operations if $\ell = f$.

There is a markedly different pattern in the case of computing **Φ** for $\ell < f$. Namely, we do not see ratios similar to Eq. (24), and it appears that $\lim_{f\to\infty} b_0 \not\approx 0$, unlike **R** and **A**. In computing **Φ**, we need to fill the entirety of **Φ**, which still requires accessing and assigning all N_s and N_t. Since the computation of a single element of **Φ** requires five operations, and we need to access and assign each element of **Φ**, which we deem two operations in our operation counting scheme, we see $\lim_{f\to\infty} b_0 \approx 2/5$, where the result is only approximate due to the many other features of the algorithm that may marginally vary this value. This limit will be more clear in further discussions. Of course, this number would change if a different scheme was used for operation counting.

Note that this limit is true for both methods 1 and 2 of computing **Φ**. In method 1, this limit is more clear since performance is only based on $p_\ell^{\max}$, while in method 2, the convoluted maneuvering through the matrices complicates this limit, but the same principle is still apparent in computing **H** before computing **Φ**. As a result of this different asymptotic behavior, the ratios as expressed by Eq. (24) are not present for **Φ**, and a more thorough examination of the code would be required to identify the behavior of b_i.

To validate this approach, we perform the exact same analysis on a different set of parameters: $d = 3$, $N_s = 16^2 \times 32$, $N_t = 50$, $N_{samp} = 8$, $f = 3$, $p_0 = k/16$, $p_1 = m/16$, $p_2 = n/16$ and $p_3 = 1 - p_0 - p_1 - p_2$, where $k, m, n \in [0, 5]$ are integers, for **R** and **Φ** (method 2). The same values are used for **Φ** (method 1) and **A**, but we also set $\tilde{p}_\ell^{\max} = p_\ell$. The fitted values are given in Table 3; here, we identify similar trends as found with $f = 2$ and $d = 2$; namely, Eq. (23) in computing **R**, Eq. (24) in computing **R** and **A**, $\lim_{f\to\infty} b_0 \approx 0$ for **R** and **A**, and $\lim_{f\to\infty} b_0 \approx 2/5$ for **Φ**. By verifying these trends, we are confident that we can approximately predict the performance of our algorithm using operation counts under a wide range of conditions.

4.3. Tests using genuine AMR data

Ultimately, the most important assessment of the new algorithm is to determine whether, and under what conditions, the algorithm accelerates the snapshot POD method on genuine AMR data from a fluid flow simulation. In this section, we show that the new algorithm can reduce operation counts and CPU time in a compiled and optimized code for data from an AMR simulation of an axisymmetric buoyant jet. The speedups are examined for three different finest levels of AMR: $f = 1$, $f = 2$, and $f = 3$.

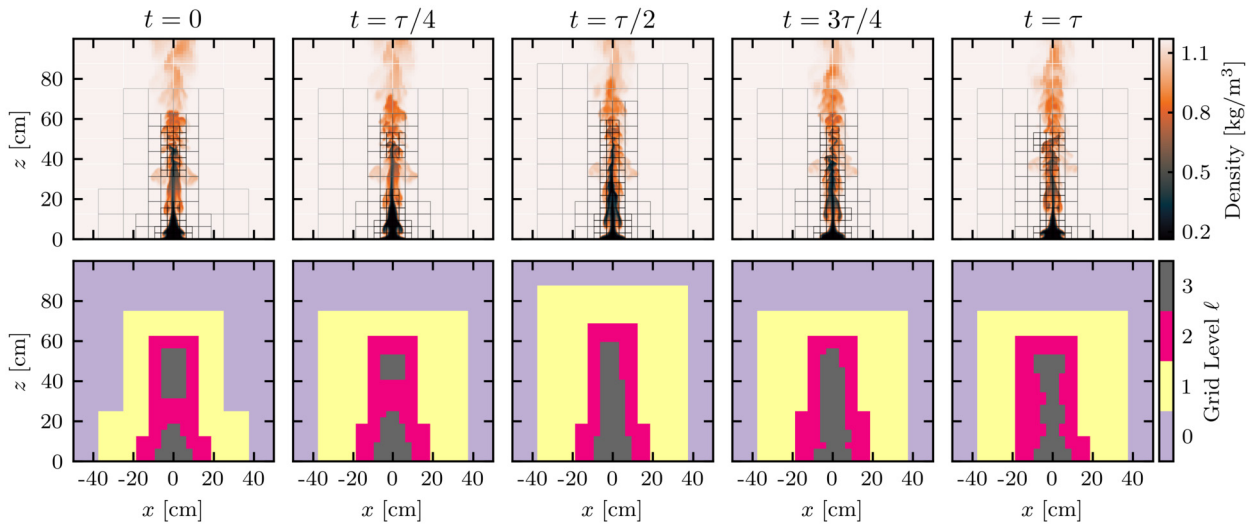

Fig. 7. Top row: Two-dimensional slices of the density field from an AMR simulation of an axisymmetric buoyant jet with solution grids annotated. Columns correspond to instances in time relative to the eddy turnover time, τ, where $\tau \approx 22\Delta t$ and $\Delta t = 0.01$ s is the simulation output rate. Bottom row: Two-dimensional slices of the corresponding grid level ℓ at each time.

4.3.1. Computational implementation

We programmed the algorithms in Fortran 90 and compiled using Intel Fortran Compiler version 17.0.5 with the command `ifort` with optimization setting `-O3` and the Intel Math Kernel Library, `-mkl` [34]. All computations in this section were done on Onyx, a supercomputer that is part of the U.S. Department of Defense High Performance Computing Modernization Program. Onyx is a Cray XC40/50 with two 2.8-GHz Intel Xeon E5-2699v4 Broadwell 22-core processors per compute node. Computations for the $f = 1$ and $f = 2$ data sets presented below were done on the standard compute nodes with 128 GB of DDR4 memory, and computations for the $f = 3$ data sets were done on the large memory nodes with 1 TB of DDR4 memory. All computations were done in serial in order to mitigate any complexities due to parallelization, but parallelization is fairly straightforward for these matrix computations by splitting $\mathbf{X}$ and $\mathbf{X}_{\text{grid}}$ into sub-matrices along each spatial dimension so that each processor can easily perform the same operations on smaller sections of the data.

We used the `dgemm` (matrix multiplication) routine from BLAS [35] as our benchmark for standard operations. Because Fortran stores arrays in column-major order, we needed to make slight modifications for the computation of $\mathbf{\Phi}$ in order to traverse memory properly, since the algorithms in the appendix were developed without knowledge of memory layout. The slight modifications are, generally, inverting loops in the matrix computation at the cost of additional accesses. This could not be done for the entirety of the algorithm without completely restructuring the code, so we only invert loops for the computation of $\mathbf{\Phi}$ that are done on the finest AMR level (i.e., $\ell = f$). We leave the full restructuring for future work. The computation of $\mathbf{R}$ and $\mathbf{A}$ naturally aligns with the order in which data is stored in Fortran, and thus requires no modifications from the original AMR algorithms. All of this code is publicly available at https://github.com/tesla-cu/amrPOD.

4.3.2. Numerical simulations

The AMR data we use is generated from a simulation of a buoyant plume, where helium is axisymmetrically injected into quiescent ambient air with a radius of 6.25 cm at the bottom of a 1.5 m^3 computational domain. Using AMR for plumes is particularly beneficial because the flow stream needs to be sufficiently far from the outer boundary condition for proper entrainment, but only a small fraction of the domain has complex flow features that require high grid resolution. The simulation was conducted using `PeleLM` [20], but we will not discuss the solution methods used to produce the data here (see Wimer et al. [36] for a detailed discussion) since this information is not of critical importance to the computation of POD on AMR data sets.

In Fig. 7, we show five two-dimensional slices of the density field and the corresponding grid level for the $f = 3$ data. The data were truncated to a 1 m^3 portion of the domain to reduce memory requirements. We then extracted nearest-neighbor-interpolated data using `yt` [23] at three different levels of finest resolution: $f = 1$, $f = 2$, and $f = 3$, leading to grid sizes, respectively, of 128^3, 256^3, and 512^3. We show the corresponding average composition, $\langle p_\ell \rangle$ and $p_\ell^{\max}$, as a function of N_t in Fig. 8, where the average operator $\langle \cdot \rangle$ denotes an average in time as opposed to samples. In comparing the grid composition with the results from Section 4.2, we can see that this data is conducive for the new algorithm to provide a computational advantage compared to the standard snaphost POD method.

4.3.3. Genuine AMR test results

In order to quantify the computational advantage of the new algorithm, we use both operation counts and average CPU time, both of which we show in Fig. 9 for $N_t \in [10, 80]$. Blue lines indicate the ratio $\overline{T}_{\text{AMR}}/T_{\text{std}}$ using operation counts

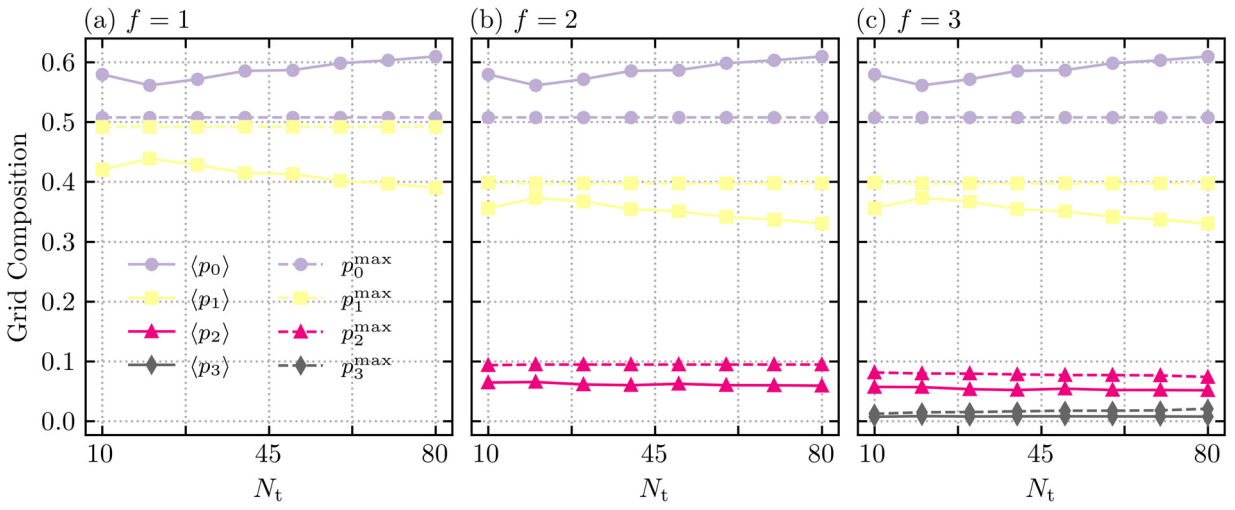

Fig. 8. Properties of the AMR grids for each of the axisymmetric buoyant jet simulations, showing $\langle p_\ell \rangle$ and $p_\ell^{\max}$ as functions of N_t for $\ell = 0, \ldots, 3$ and finest resolutions (a) $f = 1$, (b) $f = 2$, and (c) $f = 3$.

and the black lines are using CPU times. All quantities are normalized by T_{std} for $N_t = 10$. The fluctuations of CPU time within the $N_{\text{samp}} = 16$ samples of the ensemble were found to be small compared to the average CPU times, so they are not reported.

For all algorithms, we see several clear trends. The first is that there is an $\mathcal{O}(N_t^2)$ scaling of the computational cost, as anticipated, for all algorithms, with a slight deviation in CPU times for $\mathbf{\Phi}$ for larger N_t as a result of developing the algorithms without knowledge of memory layout. Asymptotically, the proposed algorithms in Section 3 do not change the order of computation, rather they change the leading coefficients. The next trend we see is that the new algorithms continue to improve with increasing f. This is to be expected because, as f increases, an increasing number of operations can be skipped with the AMR algorithm for a given ℓ. For example, comparing the amount of repeated information for $\ell = 0$ for $f = 2$ and $f = 3$, we have increased the number of repeated cells from $4^3 = 64$ to $8^3 = 512$, making repeated computations significantly more computationally advantageous to skip. This speed-up is important because many fluid flow simulations use more than three levels of AMR (e.g., in Ref. [36] five levels of AMR are used), and the computational savings will continue to improve as f increases.

In determining whether our new algorithms are practically faster than standard matrix multiplications, we look to the CPU times. For $\mathbf{R}$ and $\mathbf{A}$, we need $f > 2$ for any N_t to have any computational advantage, while both methods 1 and 2 of computing $\mathbf{\Phi}$ are advantageous when $f > 1$. Although these results depend on the specific details of the numerical simulations and would need to be re-evaluated for different data, they do again indicate that the computational speed-up of the new algorithm compared to the standard approach will continue to become more pronounced as the number of AMR levels increases.

In comparing $\overline{T}_{\text{AMR}}/T_{\text{std}}$ for operation counts and CPU time, we often see large disparities between the two for the following reasons. In the computations of $\mathbf{R}$ and $\mathbf{A}$, the disparity is likely due to a high cache miss rate when large amounts of operations are skipped. Eventually with many repeated computations, such as for $f = 3$, the new algorithm can overcome the increased latency. For $\mathbf{\Phi}$, the disparity is likely due to overestimating the weights associated with accesses and assignments that are needed for all elements in $\mathbf{\Phi}$, as discussed in Section 4.2. Further investigation is needed to validate that these are the exact reasons for the disparities. Nonetheless, the algorithm presented here has *not* been optimized for any particular hardware configuration, nor have we tuned the coefficients of the weights of each operation (e.g., arithmetic, assignment, etc. operations) to align with the performance of the machine, so these are not expected to align perfectly. These improvements are beyond the scope of this paper but are avenues for future work.

5. Practical considerations

In Sections 3 and 4, we presented the proposed algorithms and assessed their computational performance (in serial) when AMR output data had already been nearest-neighbor interpolated to a uniform common mesh of equivalent resolution to that of the finest cells. In those sections, we neglected important practical effects that would further affect the overall computation, such as the cache miss rate, memory hierarchy, and disk usage. These effects were neglected for two reasons: *(i)* to make the presentation of the new algorithms as clear as possible, given their already significant complexity; and *(ii)* these effects can vary substantially between different computing architectures, and we do not want to draw conclusions that could be unique to a specific computing environment. In this section, we discuss how the algorithms presented in Section 3 could be optimized for particular use-cases.

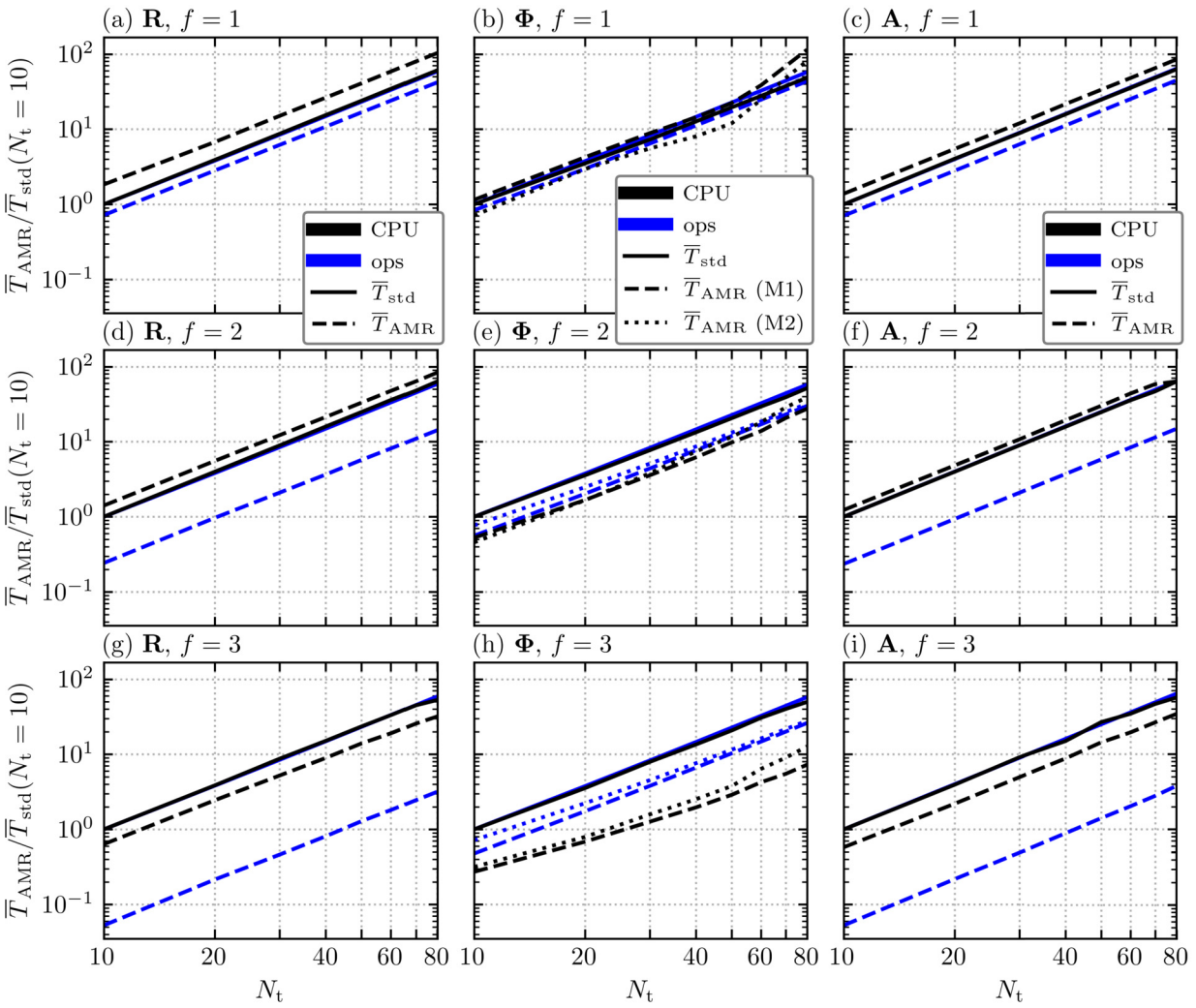

Fig. 9. Computed operation counts (blue) and CPU times (black) of the standard and AMR algorithms as functions of N_t for (a, d, g) **R**; (b, e, h) **Φ**; and (c, f, i) **A** when using a finest level of (a-c) $f = 1$, (d-f) $f = 2$, and (g-i) $f = 3$ in the axisymmetric buoyant plume simulations. The line styles correspond to the different algorithms used. All quantities are normalized by T_{std} at $N_t = 10$. The two different methods of computing **Φ** are given in (b, e, h) and denoted by M1 for method 1 and M2 for method 2. Note that the solid black and solid blue lines are almost identical.

5.1. Partial loading

One of the primary limitations of the presented algorithms is the immense memory requirements for some of the larger computations, such as the finest resolution in Section 4.3. However, we can reduce these requirements by noting that all of the data does not need to be stored in memory simultaneously and, instead, only portions of the data need to be loaded at any given time.

To demonstrate this, consider the computation of **R** as given in Section 3.2. The computation of any element R_{ij} can be split as

$$R_{ij} = \sum_{k=1}^{N_s} X_{ki} X_{kj} = \underbrace{\sum_{k=1}^{N_s/2} X_{ki} X_{kj}}_{R_{ij}^1} + \underbrace{\sum_{k=N_s/2+1}^{N_s} X_{ki} X_{kj}}_{R_{ij}^2}, \tag{25}$$

where R_{ij}^1 and R_{ij}^2 are the covariance matrices corresponding to the first and second "chunks" of data, respectively. By decomposing the computation this way, the memory requirements have been cut in half because R_{ij}^1 can first be computed using only half N_s and stored in R_{ij}. The first data chunk can then be unloaded after loading the second data chunk, and element-wise contributions of R_{ij}^2 can be added directly to R_{ij}.

More generally, we could consider decomposing the data into a certain number of chunks N_c as

$$R_{ij} = \sum_{c=1}^{N_c} \underbrace{\sum_{k=d}^{cN_s/N_c} X_{ki} X_{kj}}_{R_{ij}^c} \tag{26}$$

where $d = (c-1)N_s/N_c + 1$. By performing the computation of $\mathbf{R}$ in this way, we can reduce the memory requirements by a factor of N_c by loading only the data required for R_{ij}^c. The algorithm described in Section 3.2 is completely amenable to this decomposition as long as $N_s/N_c \geq c_0^d$, a condition that still permits many chunks. Note that the computation of $\mathbf{\Phi}$ and $\mathbf{A}$ can be split in almost an identical way.

While implementation of this approach is straightforward, we did not include this decomposition in Sections 3 or 4 because there is an additional practical consideration regarding the balance between input-output and memory requirements. Specifically, even though memory requirements decrease as N_c increases, the throughput on storage devices increases. Depending on the memory hierarchy, increasing the number of chunks such that each can fit into the cache could be optimal compared to increased throughput. As a result, the number of chunks required to minimize computational time can vary substantially between different computing environments. Therefore, we do not address this at the algorithmic level and instead take a worst-case scenario approach and show that our algorithm can reduce the computation time where memory hierarchy is not utilized to capacity.

5.2. Parallelization

The target application of these algorithms is on large scale simulation data and, as such, a discussion of parallelization is necessary. A simple way to parallelize the algorithms is to use the same methods outlined in Section 5.1, where it was noted that the computation of $\mathbf{R}$, $\mathbf{\Phi}$, and $\mathbf{A}$ can be computed incrementally with a small fraction of the data loaded at any given instant. With distributed memory (e.g., MPI), each processor is able to load a single small chunk, perform the small computation, then a collective call can be made to communicate between all processors. For $\mathbf{R}$ and $\mathbf{A}$, these collective calls would be summations to combine contributions from each processor, and for $\mathbf{\Phi}$ the collective call would be a write to disk.

Given that memory serves as a major limiter in the proposed algorithms, parallelization across many nodes in an HPC environment may be necessary. However, if this is done purely using distributed memory, many processors will be accessing the data. If N_s is not evenly divisible by N_c, the work will thus not be evenly distributed between processors. To circumvent this, we recommend a hybrid parallelization approach by using fewer tasks per node than cores, possibly as few as one task per node, and then make use of the remaining cores by using threads that share memory to parallelize mathematical operations for each data chunk. We expect this parallelization strategy to scale well given the very little communication required between processors (e.g., only collective calls at the very end of the operation) and the simple division of work between processors.

5.3. Bypassing the interpolation step

The entirety of Sections 3 and 4 have focused on the algorithms *after* the AMR output data has already been interpolated to a uniform grid. However, bypassing this interpolation step would reduce disk usage, retain compression of the data, remove a step in the overall computation, and could ultimately be leveraged to improve the algorithms further. However, bypassing the interpolation step to perform POD directly on the simulation data is highly sensitive to how the data is stored on disk and will vary between applications. For these reasons, we did not consider this approach in Section 3, but we provide a discussion here of how this approach could be taken and what advantages may be realized.

We illustrate some advantages of using the simulation data directly by considering an example. Consider simulation data that was produced with rectangular collections of cells of equivalent resolution denoted "grids" (this is a common approach of AMR codes [4]) and specifically focus on computing $\mathbf{R}$. We can iterate over spatial regions in chunk sizes (as discussed in Section 5.1) of the simulation data that correspond to the minimum grid size, which we will denote "blocks." These are often 4, 8, 16, or 32 cells per dimension. By doing this, the elements in the dot product between two blocks (these blocks would have the same spatial extent but different instances in time) would be weighted identically and, therefore, we could remove a substantial amount of computational overhead in Section 3.2 associated with checking the grid level in $\mathbf{X}_{\text{grid}}$ and weighting individual cell contributions (i.e., by replacing this with weighting blocks). This would further improve cache utilization and memory requirements. Moreover, if two blocks have the same resolution, a standard dot product can be taken between the two and can be weighted appropriately for the contribution of $\mathbf{R}$ without first interpolating to the finest resolution as required by our algorithm. By computing $\mathbf{R}$ in this manner, we retain the compression of the data and substantially reduce the additional computational overhead required when the data was interpolated using a nearest-neighbor method.

If we now consider the same type of simulation data but focus specifically on the computation of $\mathbf{\Phi}$ and $\mathbf{A}$, there is a reduced computational advantage to bypassing the interpolation step. Retaining identical compression of the simulation data as we did with $\mathbf{R}$ is impossible to do for $\mathbf{\Phi}$ with an adaptive mesh, for the reasons discussed in Section 3.4. However, some compression can still be achieved (relative to a grid that was uniformly at the finest resolution) if there are some spatial

regions of the flow that are never resolved to the finest resolution (i.e., $p_f^{\max} < 1$). This is essentially the approach already taken in Sections 3.4.1 and 3.5. To maximize compression for the POD modes, we would define the non-uniform common mesh that would be resolved only to the finest resolution that was observed in a particular spatial region throughout the simulation data. It would then be straightforward to perform the exact same computation as put forth in Section 3, except using a local value of finest resolution, rather than the global finest resolution, similar to that in the wavelet-based method outlined by Krah et al. [25].

Ultimately, we do not discuss this technique in Section 3 because this approach would most appropriately be applied in a format native to the original data. This would therefore require substantial analysis of whether the additional computational overhead is favorable, as compared to simply using a uniform common grid at the finest level that is followed by down-sampling repeated data to improve data compression.

6. Conclusions and future work

In this work, we explored new algorithms to compute each step of the snapshot POD method on data from AMR simulations, taking advantage of repeated solution values as a result of nearest-neighbor interpolation to weight and skip unnecessary computations. This was done by reshaping each snapshot iteratively, such that at the end of the iteration sequence, all cells that had identical values were contiguous in the column vector. From there, each step in the snapshot POD method was analyzed to determine algorithms that effectively leverage those repetitions and skip unnecessary operations.

After developing the algorithms, we analyzed their performance using operation counts and CPU time. Using operation counts, we performed detailed parameter sweeps with synthetically generated data that allowed us to identify key characteristics of our algorithms. We found that both $\mathbf{\Phi}$ (method 1) and $\mathbf{A}$ require that the maximum grid level ℓ for all N_t be less than the finest level (i.e., $\ell < f$) to have any computational advantage. By contrast, accelerating the calculation of $\mathbf{R}$ simply requires correlations of coarser cells and accelerating the calculation of $\mathbf{\Phi}$ (method 2) primarily requires the presence of coarser cells in the snapshot matrix $\mathbf{X}$. Additionally, we were able to determine the approximate behavior of each algorithm with additional levels of AMR by fitting equations with properties of the synthetically generated data. Using CPU time, we were able to show that, with a compiled Fortran code, we do in fact reduce CPU time by a factor of roughly $2-5$ for data from an AMR simulation of an axisymmetric buoyant plume when using three levels of grid refinement, without fully optimizing the algorithms to the computing environment (e.g., language, cache, etc.). This speed-up will become even more pronounced for a greater number of AMR levels.

This work has focused on the development of the new algorithm for computing snapshot POD on AMR data as concisely and as generally as possible. Therefore, we do not directly address many factors that would be crucial to performing this computation practically, including partial loading and parallelization. Although these are simple to implement in practice, by introducing any of these in the description of the fundamental algorithms provided in Section 3, we would also introduce dependencies of the algorithms on the system architecture, such as memory hierarchy, disk storage, and communication. In an effort to remain impartial to emerging hardware components, we only provide a discussion (in Section 5) of how to implement partial loading and parallelization. Further, the most optimal way to perform POD using AMR data could be to use the simulation data directly, rather than to require an intermediate interpolation step, but we do not do this here, again to retain generality.

We benchmark our algorithm against standard matrix multiplication which is $\mathcal{O}(N^3)$ if the matrices are of equal dimension (i.e., $N = N_s = N_t$ for the operations here). Our algorithm is not readily amenable to more advanced algorithms, such as Strassen's algorithm [37], to reduce the order below 3. However, we do not expect that these advanced algorithms will drastically increase the speed of the operation in general because it is expected that $N_s \gg N_t$ with $N_t \approx 10^3$ as an approximate upper bound. Even with some of the most recent code optimizations using modern computer architectures, this is the approximate N_t bound where the reduction in multiplications overcomes the overhead of more additions [38].

Recent advances in matrix computations have shown great promise in substantially improving the computational demands to perform POD for large-scale flow data. One of the most promising amongst these is the rSVD method developed from randomized numerical linear algebra [33]. This tool can dramatically reduce the computational cost by using a small random matrix to reduce the size of the problem $\mathbf{X}$ and produce the same POD data with very little error; see Brunton and Kutz [39] for further discussion. To integrate our algorithms with the rSVD approach (amongst other modal decompositions), a more general procedure for performing matrix operations with repeated solution values would need to be developed. This would likely hinge on ideas used in Section 3.4.2. Since we believe the present study is the first to deal with repeated values in matrix operations, we leave a more general approach as future work.

Finally, we did not attempt to match the CPU time with the operation counts. This is primarily due to the fact that the algorithm was developed independent of any computing environment (e.g., language, processor, etc.) and without any code optimization based on the caching architecture. Further developments could substantially improve the algorithm with respect to CPU time, and we could begin to bridge the gap between CPU time and operation counts, allowing us to ensure the authenticity of the bounds where we determine our AMR algorithm to be faster than the standard algorithm. This has only been done recently with standard matrix multiplication [40], and we leave these improvements for future work.

Ultimately, this is the first study that addresses the challenge of reducing the computational cost of linear operations when repeated values are present in data generated by AMR simulations. This is somewhat surprising because, considering the importance compression methods for images and audio [41], there are not algorithms that address computing quantities

across data that have already been compressed. Although we do not address compression methods to compute POD for AMR data, it might be possible to transform the AMR data using efficient compression methods and directly perform POD on the compressed data, as has recently been done using wavelet-based methods [25]. Additionally, this type of repeated computation has not been addressed in a theoretical context bounding the number of operations required.

CRediT authorship contribution statement

Michael Meehan: Conceptualization, Methodology, Software, Data Curation, Writing – Original Draft. **Sam Simons-Wellin:** Methodology, Validation, Investigation, Visualization. **Peter Hamlington:** Writing – Review & Editing, Supervision, Project Management.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

M.A.M. was supported by the NSF Graduate Research Fellowship Program, P.E.H. was supported, in part, by AFOSR Award No. FA9550–17-1–0144 and NSF Award No. 1847111.

Appendix A. Detailed algorithms

In this appendix, we provide the detailed algorithms that were introduced in Section 3 and analyzed in Section 4. In the pseudo-code, we use zero-based indexing for the array accesses. Lastly, we use typical terminology that is used in high-level computing languages to represent standard operations. These include: `size(arr,dim)` to compute the size of array `arr` along dimension `dim`; `permute(arr,dims)` to rearrange the indices of multi-dimensional array `arr` according to `dims`; `reshape(arr,dims)` to redistribute elements in `arr` into a new array with dimensions `dims` while maintaining contiguous elements in memory; and `empty(dim0,dim1,⋯)` and `zeros(dim0,dim1,⋯)` initialize arrays with dimensions `dim0` $\times$ `dim1` $\times\cdots$, respectively, with non-exist and zero values. The remaining nomenclature (e.g., indexing, assignment, math operations, etc.) is straightforward.

Algorithm 1 Reshaping $\mathbf{U}_i$ into $\boldsymbol{u}_i$ for a column of $\mathbf{X}$ for a row–major code. The difference for a column–major order code is provided on the right hand side.

$U_{1D} \leftarrow \mathbf{U}_i$
if $d = 1$ **then**
  No reshaping required
else if $d = 2$ **then**
  for c **in** c_ℓ **do**
    $n_{xr} \leftarrow \text{size}(U_{1D}, 0)$
    $m_1 \leftarrow N_s/(c * n_{xr})$
    $m_2 \leftarrow N_s/c$
    $U_{1D} \leftarrow \text{permute}(U_{1D}, (1, 0))$
    $U_{1D} \leftarrow \text{reshape}(U_{1D}, (m_1, c, n_{xr}))$ ▷ $U_{1D} \leftarrow \text{reshape}(U_{1D}, (n_{yr}, c, m_1))$
    $U_{1D} \leftarrow \text{permute}(U_{1D}, (1, 0, 2))$ ▷ $U_{1D} \leftarrow \text{permute}(U_{1D}, (0, 2, 1))$
    $U_{1D} \leftarrow \text{reshape}(U_{1D}, (m_2, c))$
  end for
else if $d = 3$ **then**
  for c **in** c_ℓ **do**
    $n_{xr} \leftarrow \text{size}(U_{1D}, 0)$
    $n_{yr} \leftarrow \text{size}(U_{1D}, 1)$
    $m_1 \leftarrow N_s/(c * n_{xr} * n_{yr})$
    $m_2 \leftarrow N_s/(c^2 * n_{xr})$
    $m_3 \leftarrow N_s/c^2$
    $U_{1D} \leftarrow \text{permute}(U_{1D}, (2, 1, 0))$
    $U_{1D} \leftarrow \text{reshape}(U_{1D}, (m_1, c, n_{yr}, n_{xr}))$ ▷ $U_{1D} \leftarrow \text{reshape}(U_{1D}, (n_{zr}, n_{yr}, c, m_1))$
    $U_{1D} \leftarrow \text{permute}(U_{1D}, (1, 0, 2, 3))$ ▷ $U_{1D} \leftarrow \text{permute}(U_{1D}, (0, 1, 3, 2))$
    $U_{1D} \leftarrow \text{reshape}(U_{1D}, (c, m_2, c, n_{xr}))$ ▷ $U_{1D} \leftarrow \text{reshape}(U_{1D}, (n_{zr}, c, m_2, c))$
    $U_{1D} \leftarrow \text{permute}(U_{1D}, (0, 2, 1, 3))$
    $U_{1D} \leftarrow \text{reshape}(U_{1D}, (c, c, m_3))$ ▷ $U_{1D} \leftarrow \text{reshape}(U_{1D}, (m_3, c, c))$
  end for
end if
$\boldsymbol{u}_i \leftarrow U_{1D}$

Algorithm 2 Computing $\mathbf{R} = \mathbf{X}^T\mathbf{X}$.

$\mathbf{R} \leftarrow \text{empty}(N_t, N_t)$
$c^d_{f-1} \leftarrow c^d[f-1]$
for $m \leftarrow 0$ **to** $N_t - 1$ **do**
  for $n \leftarrow 0$ **to** m **do**
    $r_{sum} \leftarrow 0$
    $i \leftarrow 0$
    if $m = n$ **then**
      while $i < N_s$ **do**
        if $\mathbf{X}_{grid}[i, m] = f$ **then**
          for $j \leftarrow i$ **to** $i + c^d_{f-1} - 1$ **do**
            $r_{sum} \leftarrow r_{sum} + \mathbf{X}[j, n] * \mathbf{X}[j, m]$
          end for
          $i \leftarrow i + c^d_{f-1}$
        else
          $c_{val} \leftarrow c^d[\mathbf{X}_{grid}[i, m]]$
          $r_{sum} \leftarrow r_{sum} + c_{val} * \mathbf{X}[i, n] * \mathbf{X}[i, m]$
          $i \leftarrow i + c_{val}$
        end if
      end while
    else
      while $i < N_s$ **do**
        if $\mathbf{X}_{grid}[i, m] = f$ **or** $\mathbf{X}_{grid}[i, n] = f$ **then**
          for $j \leftarrow i$ **to** $i + c^d_{f-1} - 1$ **do**
            $r_{sum} \leftarrow r_{sum} + \mathbf{X}[j, n] * \mathbf{X}[j, m]$
          end for
          $i \leftarrow i + c^d_{f-1}$
        else
          if $\mathbf{X}_{grid}[i, m] > \mathbf{X}_{grid}[i, n]$ **then**
            $c_{val} \leftarrow c^d[\mathbf{X}_{grid}[i, m]]$
          else
            $c_{val} \leftarrow c^d[\mathbf{X}_{grid}[i, n]]$
          end if
          $r_{sum} \leftarrow r_{sum} + c_{val} * \mathbf{X}[i, n] * \mathbf{X}[i, m]$
          $i \leftarrow i + c_{val}$
        end if
      end while
    end if
    $\mathbf{R}[m, n] \leftarrow r_{sum}$
    $\mathbf{R}[n, m] \leftarrow r_{sum}$
  end for
end for

Algorithm 3 Computing $\mathbf{\Phi} = \mathbf{X}\mathbf{\Psi}\mathbf{\Lambda}^{-1/2}$ – Method 1.

$\mathbf{\Phi} \leftarrow \text{empty}(N_s, N_t)$
$c_0 \leftarrow c^d[0]$
for $i \leftarrow 0$ **to** $N_s - 1$ **inc** c_0 **do**
  $\mathbf{G} \leftarrow \text{empty}(c_0)$
  $j \leftarrow i$
  $j^* \leftarrow 0$
  while $j^* < c_0$ **do**
    $X_{max} \leftarrow \mathbf{X}_{grid}[j, 0]$
    for $m \leftarrow 1$ **to** $N_t - 1$ **do**
      if $\mathbf{X}_{grid}[j, m] > X_{max}$ **then**
        $X_{max} \leftarrow \mathbf{X}_{grid}[j, m]$
        if $X_{max} = f$ **then**
          Terminate **for** loop
        end if
      end if
    end for
    $g_{val} \leftarrow c^d[X_{max}]$
    $\mathbf{G}[j^*] \leftarrow g_{val}$
    $j \leftarrow j + g_{val}$
    $j^* \leftarrow j^* + g_{val}$
  end while
  for $m \leftarrow 0$ **to** $N_t - 1$ **do**
    $j \leftarrow i$
    $j^* \leftarrow 0$
    while $j^* < c_0$ **do**
      $\phi_{sum} \leftarrow 0$

```
            for n ← 0 to N_t − 1 do
                φ_sum ← φ_sum + X[j, n] ∗ Ψ[n, m]
            end for
            g_val ← g[j*]
            φ_sum ← φ_sum ∗ λ_m^{−1/2}
            for k ← j to j + g_val − 1 do
                Φ[k, m] = φ_sum
            end for
            j ← j + g_val
            j* ← j* + g_val
        end while
    end for
end for
```

Algorithm 4 Computing $\mathbf{\Phi} = \mathbf{X\Psi\Lambda}^{-1/2}$ – Method 2.

```
Φ ← empty(N_s, N_t)
c_0 ← c^d[0]
c_1 ← c^d[1]
c^d_{f−1} ← c^d[f − 1]
n_lev ← f + 1
for i ← 0 to N_s − 1 inc c_0 do
    G ← empty(n_lev, c_1, N_t)
    n_ℓ ← empty(n_lev, c_1)
    for n ← 0 to N_t − 1 do
        ℓ ← X_grid[i, n]
        if ℓ = 0 then
            G[0, 0, n_ℓ[0, 0]] ← n
            n_ℓ[0, 0] ← n_ℓ[0, 0] + 1
        else
            if f > 1 then
                j ← i
                j* ← 0
                while j* < c_1 do
                    ℓ ← X_grid[j, n]
                    if ℓ = f then
                        G[f, j*, n_ℓ[f, j*]] ← n
                        n_ℓ[f, j*] ← n_ℓ[f, j*] + 1
                        j ← j + c^d_{f−1}
                        j* ← j* + 1
                    else
                        G[ℓ, j*, n_ℓ[ℓ, j*]] ← n
                        n_ℓ[ℓ, j*] ← n_ℓ[ℓ, j*] + 1
                        j ← j + c^d[ℓ]
                        j* ← j* + c^d[ℓ + 1]
                    end if
                end while
            else
                G[1, 0, n_ℓ[1, 0]] ← n
                n_ℓ[1, 0] ← n_ℓ[1, 0] + 1
            end if
        end if
    end for
    for n ← 0 to N_t − 1 do
        H ← empty(c_0, n_lev)
        if n_ℓ[0, 0] > 0 then
            ℓ_sum ← 0
            for m ← 0 to n_ℓ[0, 0] − 1 do
                k ← G[0, 0, m]
                ℓ_sum ← ℓ_sum + X[i, k] ∗ Ψ[k, n]
            end for
            for m ← 0 to c_0 do
                H[m, 0] ← ℓ_sum
            end for
        end if
        if n_ℓ[0, 0] < N_t then
            if f > 2 then
                for L ← 1 to f − 2 do
                    j* ← 0
                    for j ← i to i + c_0 inc c^d[L] do
                        if n_ℓ[L, j*] > 0 then
                            ℓ_sum ← 0
                            for m ← 0 to n_ℓ[L, j*] − 1 do
```

```
                    k ← G[L, j*, m]
                    ℓ_sum ← ℓ_sum + X[j, k] * Ψ[k, n]
                end for
                for m ← j* * c^d_{f−1} to j* * c^d_{f−1} + c^d[L] − 1 do
                    H[m, L] ← ℓ_sum
                end for
            end if
            j* ← j* + c^d[L + 1]
        end for
    end for
end if
if f > 1 then
    j* ← 0
    for j ← i to i + c_0 − 1 inc c^d[f − 1] do
        if n_ℓ[f − 1, j*] > 0 then
            L ← f − 1
            ℓ_sum ← 0
            for m ← 0 to n_ℓ[L, j*] − 1 do
                k ← G[L, j*, m]
                ℓ_sum ← ℓ_sum + X[j, k] * Ψ[k, n]
            end for
            for m ← j* * c_d[L] to (j* + 1) * c_d[L] − 1 do
                H[m, L] ← ℓ_sum
            end for
        end if
        if n_ℓ[f, j*] > 0 then
            for k ← j to j + c^d[f] − 1 do
                ℓ_sum ← 0
                for m ← 0 to n_ℓ[f, j*] − 1 do
                    p ← G[f, j*, m]
                    ℓ_sum ← ℓ_sum + X[k, p] * Psi[p, n]
                end for
                H[k − j + c^d_{f−1} * j*, f] ← ℓ_sum
            end for
        end if
        j* ← j* + 1
    end for
else
    for k ← i to i + c_0 − 1 do
        ℓ_sum ← 0
        for m ← 0 to n_ℓ[1, 0] − 1 do
            p = G[1, 0, m]
            ℓ_sum ← ℓ_sum + X[k, p] * Ψ[p, n]
        end for
        H[k − i, 1] ← ℓ_sum
    end for
end if
end if
for m ← 0 to c_0 − 1 do
    H_sum ← 0
    for L ← 0 to f do
        H_sum ← H_sum + H[m, L]
    end for
    Φ[m + i, n] ← H_sum * λ^{−1/2}[n]
end for
end for
end for
```

Algorithm 5 Reshaping ϕ_i into the original flowfield shape for a row–major code. The difference for a column–major order code is provided on the right hand side.

```
Φ_1D ← φ_i
if d = 1 then
    No reshaping required
else if d = 2 then
    for c in reversed c_ℓ excluding last element do
        n_xr ← size(Φ_1D, 0)                                   ▷ n_yr ← size(Φ_1D, 1)
        m_1 ← N_s/(c * n_xr)                                   ▷ m_1 ← N_s/(c * n_yr)
        m_2 ← N_s/c

        Φ_1D ← reshape(Φ_1D, (n_xr, m_1, c))                   ▷ Φ_1D ← reshape(Φ_1D, (c, m_1, n_yr))
        Φ_1D ← permute(Φ_1D, (1, 0, 2))                        ▷ Φ_1D ← permute(Φ_1D, (0, 2, 1))
        Φ_1D ← reshape(Φ_1D, (m_2, c))                         ▷ Φ_1D ← reshape(Φ_1D, (c, m_2))
```

```
        Φ_1D ← permute(Φ_1D, (1, 0))
    end for
    n_xr ← size(U_1D, 0)                                   ▷ n_yr ← size(U_1D, 1)
    m_1 ← N_s/(n_x * n_xr)                                 ▷ m_1 ← N_s/(n_y * n_yr)

    Φ_1D ← reshape(Φ_1D, (n_xr, m_1, n_x))                 ▷ Φ_1D ← reshape(Φ_1D, (n_y, m_1, n_yr))
    Φ_1D ← permute(Φ_1D, (1, 0, 2))                        ▷ Φ_1D ← permute(Φ_1D, (0, 2, 1))
    Φ_1D ← reshape(Φ_1D, (n_y, n_x))
    Φ_1D ← permute(Φ_1D, (1, 0))
else if d = 3 then
    for c in reversed c_ℓ excluding last element do
        n_xr ← size(Φ_1D, 0)                               ▷ n_yr ← size(Φ_1D, 1)
        n_yr ← size(Φ_1D, 1)                               ▷ n_zr ← size(Φ_1D, 2)
        m_1 ← N_s/(c * n_xr * n_yr)                        ▷ m_1 ← N_s/(c * n_yr * n_zr)
        m_2 ← N_s/(c^2 * n_xr)                             ▷ m_2 ← N_s/(c^2 * n_zr)
        m_3 ← N_s/c^2

        Φ_1D ← reshape(Φ_1D, (n_xr, n_yr, m_1, c))         ▷ Φ_1D ← reshape(Φ_1D, (c, m_1, n_yr, n_zr))
        Φ_1D ← permute(Φ_1D, (0, 2, 1, 3))
        Φ_1D ← reshape(Φ_1D, (n_xr, m_2, c, c))            ▷ Φ_1D ← reshape(Φ_1D, (c, c, m_2, n_zr))
        Φ_1D ← permute(Φ_1D, (1, 0, 2, 3))                 ▷ Φ_1D ← permute(Φ_1D, (0, 1, 3, 2))
        Φ_1D ← reshape(Φ_1D, (m_3, c, c))                  ▷ Φ_1D ← reshape(Φ_1D, (c, c, m_3))
        Φ_1D ← permute(Φ_1D, (2, 1, 0))
    end for
    n_xr ← size(Φ_1D, 0)                                   ▷ n_yr ← size(Φ_1D, 1)
    n_yr ← size(Φ_1D, 1)                                   ▷ n_zr ← size(Φ_1D, 2)
    m_1 ← N_s/(n_x * n_xr * n_yr)                          ▷ m_1 ← N_s/(n_z * n_yr * n_zr)
    m_2 ← N_s/(n_x * n_y * n_xr)                           ▷ m_2 ← N_s/(n_z * n_y * n_zr)

    Φ_1D ← reshape(Φ_1D, (n_xr, n_yr, m_1, n_x))           ▷ Φ_1D ← reshape(Φ_1D, (n_z, m_1, n_yr, n_zr))
    Φ_1D ← permute(Φ_1D, (0, 2, 1, 3))
    Φ_1D ← reshape(Φ_1D, (n_xr, m_2, n_y, n_x))            ▷ Φ_1D ← reshape(Φ_1D, (n_z, n_y, n_zr, m_2))
    Φ_1D ← permute(Φ_1D, (1, 0, 2, 3))                     ▷ Φ_1D ← permute(Φ_1D, (0, 1, 3, 2))
    Φ_1D ← reshape(Φ_1D, (n_z, n_y, n_x))
    Φ_1D ← permute(Φ_1D, (2, 1, 0))
end if
```

Algorithm 6 Computing $\mathbf{A} = \mathbf{X}^T \mathbf{\Phi}$.

```
A ← empty(N_t, N_t)
g ← empty(N_s)
i ← 0
c^d_{f−1} ← c^d[f − 1]
while i < N_s do
    X_max ← X_grid[i, 0]
    for m ← 0, N_t − 1 do
        if X_grid[i, m] > X_max then
            X_max ← X_grid[i, m]
            if X_max = f then
                Terminate for loop.
            end if
        end if
    end for
    if X_max = f then
        g[i] ← 1
        i ← i + c^d_{f−1}
    else
        g[i] ← c^d[X_max]
        i ← i + g[i]
    end if
end while
for m ← 0, N_t − 1 do
    for n ← 0, N_t − 1 do
        i ← 0
        a_sum ← 0
        while i < N_s do
            if g[i] = 1 then
                for j ← i to i + c^d_{f−1} − 1 do
                    a_sum ← a_sum + X[j, m] * Φ[j, n]
                end for
                i ← i + c^d_{f−1}
            else
                a_sum ← a_sum + g[i] * X[i, m] * Φ[i, n]
```

$i \leftarrow i + g[i]$
end if
end while
$\mathbf{A}[m, n] \leftarrow a_{\text{sum}}$
end for
end for

References

[1] S.B. Pope, Turbulent Flows, IOP Publishing, 2001.
[2] I. Babuška, M. Suri, The p and h-p versions of the finite element method, basic principles and properties, SIAM Rev. 36 (4) (1994) 578–632.
[3] W. Shyy, H. Udaykumar, M. Rao, R. Smith, Computational fluid dynamics with moving boundaries, AIAA J. 36 (2) (1998) 303–304.
[4] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, et al., A survey of high level frameworks in block-structured adaptive mesh refinement packages, J. Parallel Distrib. Comput. 74 (12) (2014) 3217–3227.
[5] X. Cai, H. Marschall, M. Wörner, O. Deutschmann, Numerical simulation of wetting phenomena with a phase-field method using OpenFOAM®, Chem. Eng. Technol. 38 (11) (2015) 1985–1992.
[6] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, M. Zingale, AMReX: a framework for block-structured adaptive mesh refinement, J. Open Sour. Softw. 4 (37) (2019) 1370.
[7] G. Berkooz, P. Holmes, J.L. Lumley, The proper orthogonal decomposition in the analysis of turbulent flows, Annu. Rev. Fluid Mech. 25 (1) (1993) 539–575.
[8] K. Taira, M.S. Hemati, S.L. Brunton, Y. Sun, K. Duraisamy, S. Bagheri, S.T. Dawson, C.-A. Yeh, Modal analysis of fluid flows: applications and outlook, AIAA J. (2019) 1–25.
[9] K. Taira, S.L. Brunton, S.T. Dawson, C.W. Rowley, T. Colonius, B.J. McKeon, O.T. Schmidt, S. Gordeyev, V. Theofilis, L.S. Ukeiley, Modal analysis of fluid flows: an overview, AIAA J. (2017) 4013–4041.
[10] P. Holmes, J.L. Lumley, G. Berkooz, C.W. Rowley, Turbulence, Coherent Structures, Dynamical Systems and Symmetry, Cambridge University Press, 2012.
[11] T. Colonius, J. Freund, POD analysis of sound generation by a turbulent jet, in: 40th AIAA Aerospace Sciences Meeting & Exhibit, 2002, p. 72.
[12] S. Kostka, A.C. Lynch, B.C. Huelskamp, B.V. Kiel, J.R. Gord, S. Roy, Characterization of flame-shedding behavior behind a bluff-body using proper orthogonal decomposition, Combust. Flame 159 (9) (2012) 2872–2882.
[13] F. Fang, C. Pain, I. Navon, M. Piggott, G. Gorman, P. Allison, A. Goddard, Reduced-order modelling of an adaptive mesh ocean model, Int. J. Numer. Methods Fluids 59 (8) (2009) 827–851.
[14] F. Fang, C. Pain, I. Navon, G. Gorman, M. Piggott, P. Allison, P. Farrell, A. Goddard, A POD reduced order unstructured mesh ocean modelling method for moderate Reynolds number flows, Ocean Model. 28 (1–3) (2009) 127–136.
[15] J. Du, F. Fang, C.C. Pain, I. Navon, J. Zhu, D.A. Ham, Pod reduced-order unstructured mesh modeling applied to 2d and 3d fluid flow, Comput. Math. Appl. 65 (3) (2013) 362–379.
[16] G.F. Barros, M. Grave, A. Viguerie, A. Reali, A.L. Coutinho, Dynamic mode decomposition in adaptive mesh refinement and coarsening simulations, arXiv preprint, arXiv:2104.14034, 2021.
[17] J.S. Hale, E. Schenone, D. Baroli, L.A. Beex, S.P. Bordas, A hyper-reduction method using adaptivity to cut the assembly costs of reduced order models, Comput. Methods Appl. Mech. Eng. 380 (2021) 113723.
[18] M.S. Day, J.B. Bell, Numerical simulation of laminar reacting flows with complex chemistry, Combust. Theory Model. 4 (4) (2000) 535.
[19] K.T. Mandli, C.N. Dawson, Adaptive mesh refinement for storm surge, Ocean Model. 75 (2014) 36–50.
[20] N. Wimer, C. Lapointe, J. Christopher, S. Nigam, T. Hayden, A. Upadhye, M. Strobel, G. Rieker, P. Hamlington, Scaling of the puffing Strouhal number for buoyant jets and plumes, J. Fluid Mech. 895 (2020).
[21] C. Lapointe, N.T. Wimer, J.F. Glusman, A.S. Makowiecki, J.W. Daily, G.B. Rieker, P.E. Hamlington, Efficient simulation of turbulent diffusion flames in openfoam using adaptive mesh refinement, Fire Saf. J. 111 (2020) 102934.
[22] J. Ahrens, B. Geveci, C. Law, Paraview: an end-user tool for large data visualization, in: The Visualization Handbook, 2005, p. 717.
[23] M.J. Turk, B.D. Smith, J.S. Oishi, S. Skory, S.W. Skillman, T. Abel, M.L. Norman, yt: a multi-code analysis toolkit for astrophysical simulation data, Astrophys. J. Suppl. Ser. 192 (1) (2010) 9.
[24] D.F. Muñoz, D. Yin, R. Bakhtyar, H. Moftakhari, Z. Xue, K. Mandli, C. Ferreira, Inter-model comparison of delft3d-fm and 2d hec-ras for total water level prediction in coastal to inland transition zones, J. Am. Water Resour. Assoc. (2021).
[25] P. Krah, T. Engels, K. Schneider, J. Reiss, Wavelet adaptive proper orthogonal decomposition for large scale flow data, arXiv preprint, arXiv:2011.05016, 2020.
[26] M. Ohlberger, F. Schindler, Error control for the localized reduced basis multiscale method with adaptive on-line enrichment, SIAM J. Sci. Comput. 37 (6) (2015) A2865–A2895.
[27] M. Yano, A minimum-residual mixed reduced basis method: exact residual certification and simultaneous finite-element reduced-basis refinement, ESAIM Math. Model. Numer. Anal. 50 (1) (2016) 163–185.
[28] C.W. Rowley, T. Colonius, R.M. Murray, Model reduction for compressible flows using POD and Galerkin projection, Physica D 189 (1–2) (2004) 115–129.
[29] L. Sirovich, Turbulence and the dynamics of coherent structures. I. Coherent structures, Q. Appl. Math. 45 (3) (1987) 561–571.
[30] S.S. Skiena, The Algorithm Design Manual: Text, vol. 1, Springer Science & Business Media, 1998.
[31] S.J. Chapman, Fortran 90/95 for Scientists and Engineers, McGraw-Hill Higher Education, 2004.
[32] A.V. Aho, J.D. Ullman, Foundations of Computer Science, Computer Science Press, Inc., 1992.
[33] N. Halko, P.-G. Martinsson, J.A. Tropp, Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions, SIAM Rev. 53 (2) (2011) 217–288.
[34] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, Intel math kernel library, in: High-Performance Computing on the Intel® Xeon Phi™, Springer, 2014, pp. 167–188.
[35] J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.
[36] N.T. Wimer, M.S. Day, C. Lapointe, M.A. Meehan, A.S. Makowiecki, J.F. Glusman, J.W. Daily, G.B. Rieker, P.E. Hamlington, Numerical simulations of buoyancy-driven flows using adaptive mesh refinement: structure and dynamics of a large-scale helium plume, Theor. Comput. Fluid Dyn. 35 (1) (2021) 61–91.
[37] V. Strassen, Gaussian elimination is not optimal, Numer. Math. 13 (4) (1969) 354–356.
[38] J. Huang, T.M. Smith, G.M. Henry, R.A. van de Geijn, Strassen's algorithm reloaded, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2016, p. 59.
[39] S.L. Brunton, J.N. Kutz, Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control, Cambridge University Press, 2019.
[40] A.R. Benson, G. Ballard, A framework for practical parallel fast matrix multiplication, in: ACM SIGPLAN Notices, vol. 50, ACM, 2015, pp. 42–53.
[41] D. Salomon, A Guide to Data Compression Methods, Springer Science & Business Media, 2013.