xspline3d: A Python Library for MPI-Based Spline Interpolation Enforcing Global Continuity in Distributed 3D Volumes

Wenyang Zhao
RIKEN Center for Computational Science
Kobe, Hyogo, Japan
wenyang.zhao@riken.jp

Osamu Miyashita
RIKEN Center for Computational Science
Kobe, Hyogo, Japan
osamu.miyashita@riken.jp

Florence Tama*
Nagoya University
Nagoya, Aichi, Japan
RIKEN Center for Computational Science
Kobe, Hyogo, Japan
florence.tama@riken.jp

Abstract

Spline interpolation plays a fundamental role in scientific computing and data science. In recent years, the size of datasets has been increasing rapidly, requiring them to be partitioned across distributed memory. However, performing multivariate spline interpolation on large-scale, distributed, multidimensional data is non-trivial, as it requires the entire system to be solved globally to enforce global continuity. To address this challenge, we propose parallel algorithms to solve the global system progressively across dimensions with data redistribution between steps, implemented as a Python library `xspline3d`, the first publicly available library supporting MPI-based multivariate spline interpolation in distributed 3D volumes. This library effectively overcomes the single-node memory bottleneck associated with large-scale datasets. It is expected to evolve into a widely adopted utility library on HPC platforms.

Keywords

Spline interpolation, Large-scale, Parallel computing, Python library

1 Introduction

Interpolation is a fundamental operation in scientific computing and data science. It generates new data points from a discrete set of known points and is widely applied in tasks such as resampling discrete functions, refining sampling resolution, and aligning datasets on different coordinate grids. Interpolation can be either univariate in 1D space or multivariate in multidimensional spaces. A variety of interpolation methods are available, including nearest-neighbor, linear, Hermite, and spline. Among these, spline interpolation offers distinct advantages. It constructs a smooth interpolant, providing a stable and flexible approximation for discrete data sampled from an underlying smooth function. For a spline of degree d, continuity is enforced up to the $(d-1)$-th derivative. For example, a cubic ($d = 3$) spline ensures continuity of both its first and second derivatives, referred to as C^2 continuity. In contrast, cubic Hermite interpolation guarantees up to C^1 continuity, producing discontinuous slopes at interval boundaries. Linear interpolation provides only C^0 continuity, producing sharp corners between points. Nearest-neighbor interpolation offers no continuity, producing staircase artifacts. Figure 1 illustrates these differences using data points sampled from a modulated sine function. The cubic spline interpolant demonstrates superior smoothness and provides the closest approximation to the original function.

The advantages of univariate spline interpolation naturally extend to higher dimensions, making multivariate spline interpolation a widely used tool across diverse areas of scientific computing. For example, in medical imaging, it has been shown to outperform alternative methods in terms of interpolation accuracy [20, 33] for operations such as image rotation and subpixel translation in 3D medical images from computed tomography, magnetic resonance imaging, positron emission tomography, and similar techniques. It is also used for registering 3D images of nonrigid soft tissues [31, 32, 44, 45]. Furthermore, it has been found effective for solving 3D diffusion partial differential equations [36, 37] in optical diffusion tomography. In computational fluid dynamics, multivariate spline interpolation is widely employed [2], including simulations of convective turbulence [18] and reacting compressible fluid flows [25]. It is also employed to model particle transport in large-scale systems such as ocean currents [21, 23]. In structural biology, multivariate spline interpolation plays a key role in processing 2D and 3D images of biomacromolecules obtained from cryo-electron microscopy [14, 29, 38, 40] and X-ray free-electron lasers (XFEL) [22]. In aerospace engineering, multivariate spline interpolation generates smooth aerodynamic datasets, aiding the design and optimization of aerospace vehicles [24]. In gravitational-wave physics, it supports the modeling of high-dimensional parameter spaces for black hole systems [4, 35]. This is not an exhaustive list of applications. In summary, multivariate spline interpolation libraries are fundamental tools for a wide range of scientific computing research.

To date, a number of publicly or commercially available tools focusing on multivariate spline interpolation have been developed. For example, the classical Fortran package `FITPACK` [8] provides routines for spline fitting on 1D curves and 2D surfaces; these routines are also accessible through `SciPy`'s `interpolate` module, which provides a convenient wrapper. The MATLAB Spline Toolbox [6] offers functions for creating, evaluating, and manipulating splines for interpolation. Other recent open-source libraries include

*Corresponding author

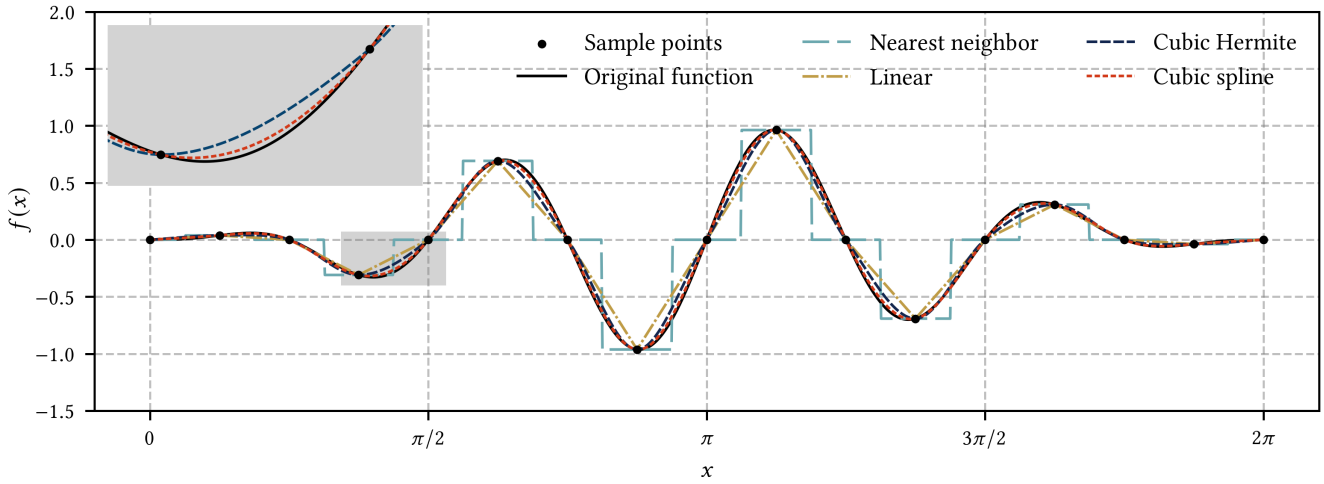

Figure 1: Comparison of interpolation methods on sample points from a modulated sine function. The plot is zoomed in near a trough to make it more visible that cubic spline interpolation approximates the original function more closely than cubic Hermite interpolation.

`bspline-fortran` [42], `splinter` [12] written in C++, `TPI` [30] written in Cython/C, and `ndsplines` [19] written in Python. Perhaps the most accessible tool for most users is the `RegularGridInterpolator` (hereafter referred to as `SciPy RGI`) [7] class in `SciPy`'s `interpolate` module.

All the tools mentioned above are designed for single-process execution on a single node. However, in recent years, the size of datasets in scientific computing has grown dramatically. They often exceed the memory capacity of a single computing node and have to be partitioned and distributed across multiple nodes. In this situation, directly applying single-process spline tools to each partition, constructing local spline interpolants independently on each node, and then stitching them together is mathematically invalid, because spline interpolant requires global continuity and must be constructed by solving the entire system in a unified manner. This raises an urgent need for new spline interpolation libraries that can handle large-scale, distributed, multidimensional datasets, support MPI-based parallelization, and operate efficiently on HPC platforms.

Our present work is primarily motivated by the need to extract 2D oriented images via spline interpolation from large-scale 3D volumetric data obtained from XFEL diffraction experiments on macromolecules [22]. Since no currently available tool can handle this task when the data is distributed across multiple nodes, we developed a new Python library, `xspline3d` [46], which supports trivariate spline interpolation in 3D volumes partitioned across multiple nodes using either 1D slab or 2D pencil decompositions. In this paper, we briefly review the mathematical background of multivariate spline interpolation, describe our MPI-based parallel algorithms, introduce the usage of `xspline3d`, and present benchmark results on multiple platforms.

2 Background: Multivariate Spline Interpolation

Splines are frequently represented in B-form as a linear combination of B-spline basis functions rather than in the equivalent piecewise polynomial form (ppform). This preference is due to several advantages of the B-form, including higher computational efficiency, improved numerical stability, local control, and compact storage. In particular, B-splines can be naturally extended from 1D space to multi dimensions via tensor-product construction. Comprehensive introductions to B-spline basis functions, univariate B-spline interpolation, and the tensor-product multivariate extension can be found in de Boor's foundational work [5], as well as in standard textbooks [27, 34] published later. In this paper, to facilitate the presentation of our parallel algorithms, we briefly review the necessary mathematical background and formulate the equations directly relevant to this work.

2.1 Univariate B-spline interpolation

The univariate B-spline interpolant of order d in 1D space can be expressed as

$$f(x) = \sum_{i=1}^{N} c_i \, B_i^{(d)}(x) \tag{1}$$

where N is the number of data points and $B_i^{(d)}(x)$ denotes the i-th B-spline basis function of degree d, constructed using the Cox-de Boor recursion. To construct the interpolant that passes exactly through all data points, we need to construct N basis functions and correspondingly determine the N coefficients c_i.

The coefficients can be uniquely determined by solving a linear system

$$\mathbf{A}\boldsymbol{c} = \boldsymbol{f} \tag{2}$$

where $\boldsymbol{f}$ is a column vector whose i-th entry is the given value at the i-th data point, $\boldsymbol{c}$ is the column vector of unknown coefficients c_i, and $\mathbf{A}$ is the B-spline collocation matrix of size $N \times N$, with

entries $\mathbf{A}_{pq} = B_q^{(d)}(x_p)$, representing the value of the q-th B-spline basis function evaluated at the p-th data point.

Since each B-spline basis function is nonzero over at most $d+1$ neighboring intervals, the collocation matrix $\mathbf{A}$ is banded, with a bandwidth at most $d+1$. This structure allows the linear system to be efficiently solved using a banded LU decomposition followed by forward and backward substitution [11]. The computational complexity of this banded solver is $O(Nd^2)$.

2.2 Extension to multivariate B-spline interpolation via tensor products

B-spline interpolation can be naturally extended from 1D space to multidimensional spaces using a tensor-product apporach. Multivariate basis functions can be expressed as the tensor product of univariate basis functions. Since the focus of `xspline3d` is on 3D space, here we present the 3D formulation explicitly. Extensions to higher-dimensional spaces follow in an analogous manner.

The trivariate B-spline interpolant in 3D space can be expressed as

$$f(x,y,z) = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z} c_{ijk}\, B_i^{(d_x)}(x)\, B_j^{(d_y)}(y)\, B_k^{(d_z)}(z). \quad (3)$$

Here, the B-spline basis functions $B_i^{(d_x)}(x)$, $B_j^{(d_y)}(y)$, and $B_k^{(d_z)}(z)$ are independently constructed along the x-, y-, and z-axes, of orders d_x, d_y, and d_z, respectively. Similar to the 1D case, the $N_x \times N_y \times N_z$ coefficients c_{ijk} can be uniquely determined by solving a large linear system. Let $\mathbf{C} \in \mathbb{R}^{N_x \times N_y \times N_z}$ be the tensor of coefficients c_{ijk}, and let $\mathbf{F} \in \mathbb{R}^{N_x \times N_y \times N_z}$ be the tensor of given values at the grid points. Define $\mathbf{A}_x \in \mathbb{R}^{N_x \times N_x}$, $\mathbf{A}_y \in \mathbb{R}^{N_y \times N_y}$, and $\mathbf{A}_z \in \mathbb{R}^{N_z \times N_z}$ as the 1D B-spline collocation matrices along each axis. For example, the entry $(\mathbf{A_x})_{pq} = B_q^{(d_x)}(x_p)$ is the value of the q-th B-spline function evaluated at the p-th grid point x_p along the x-axis. Then, the large linear system can be written compactly as

$$(\mathbf{A}_x \otimes \mathbf{A}_y \otimes \mathbf{A}_z) \cdot \mathrm{vec}(\mathbf{C}) = \mathrm{vec}(\mathbf{F}) \quad (4)$$

where $\otimes$ denotes the Kronecker product and $\mathrm{vec}(\cdot)$ flattens a 3D tensor into a column vector. The system matrix $\mathbf{A}_x \otimes \mathbf{A}_y \otimes \mathbf{A}_z$ has size $(N_xN_yN_z) \times (N_xN_yN_z)$ and contains the values of all $N_xN_yN_z$ trivariate B-spline basis functions evaluated at the $N_xN_yN_z$ grid points.

Directly solving this large linear system using, for example, Gaussian elimination or LU decomposition has a computational complexity of $O\big((N_xN_yN_z)^3\big)$. However, although the system matrix $\mathbf{A}_x \otimes \mathbf{A}_y \otimes \mathbf{A}_z$ is no longer banded as in the 1D case, it remains sparse due to the local support of B-spline basis functions. This sparsity allows the use of efficient sparse iterative solvers, which may reduce the computational cost to some extent.

2.3 Dimension-by-dimension construction of tensor product B-spline interpolants

Equation (4) provides a concise and intuitive approach for computing the coefficients to construct the multivariate interpolant. However, as dimensionality increases, the size of the linear system grows rapidly, giving rise to the "curse of dimensionality" and significant computational and memory overhead. Furthermore, incorporating parallel algorithms to solve such large systems remains challenging. Fortunately, owing to the tensor-product structure of the B-spline interpolant, an elegant and computationally efficient alternative approach exists. Specifically, Eq. (4) can be reformulated such that the coefficients c_{ijk} are computed in three sequential steps, one along each axis.

First, construct 1D interpolants along the x-axis. For each fixed (j,k), solve a linear system similar to Eq. (2). In total, $N_y \times N_z$ systems are solved. The tensor of the first intermediate coefficients is denoted as $\mathbf{C}''$:

$$\mathbf{A}_x \cdot \mathbf{C}''_{:,j,k} = \mathbf{F}_{:,j,k} \quad (5a)$$

Second, using the first intermediate coefficients as the target function, construct 1D interpolants along the y-axis. For each fixed (i,k), solve for the second intermediate coefficients in $\mathbf{C}'$:

$$\mathbf{A}_y \cdot \mathbf{C}'_{i,:,k} = \mathbf{C}''_{i,:,k} \quad (5b)$$

Finally, using the second intermediate coefficients as the target function, construct 1D interpolants along the z-axis. For each fixed (i,j), solve for the target coefficients c_{ijk} in $\mathbf{C}$. In this way, the trivariate interpolant is constructed:

$$\mathbf{A}_z \cdot \mathbf{C}_{i,j,:} = \mathbf{C}'_{i,j,:} \quad (5c)$$

Analogous to Eq. (2), the collocation matrices $\mathbf{A}_x$, $\mathbf{A}_y$, and $\mathbf{A}_z$ in the three steps of Eqs. (5a)-(5c) are all banded. Consequently, the corresponding linear systems can be efficiently solved using a banded solver. The computational complexity of this dimension-by-dimension approach is approximately $O(N_xN_yN_zd^2)$, where d denotes the spline degree, identical across all three axes. This is significantly lower than the $O\big((N_xN_yN_z)^3\big)$ complexity of solving the full 3D system in Eq. (4) in a single step.

2.4 Evaluating the interpolant at query points

After all coefficients c_{ijk} in $\mathbf{C}$ are determined, the trivariate B-spline interpolant is fully constructed. To evaluate the interpolant at an arbitrary query point (x_q, y_q, z_q), one simply performs the forward computation in Eq. (3). Owing to the local support of B-spline basis functions, only a small subset of basis functions are nonzero at the query point. Specifically, if x_q lies within the μ-th interval between two adjacent knots along the x-axis, defined by $t_{x,\mu} \le x_q < t_{x,\mu+1}$, then at most the basis functions with indices from $\mu - d_x$ to μ contribute to the evaluation. The same holds for the y- and z-axes.

Consequently, Eq. (3) can be simplified to

$$f(x_q,y_q,z_q) = \sum_{i=\mu-d_x}^{\mu}\sum_{j=\nu-d_y}^{\nu}\sum_{k=\xi-d_z}^{\xi} c_{ijk}\, B_i^{(d_x)}(x_q)\, B_j^{(d_y)}(y_q)\, B_k^{(d_z)}(z_q) \quad (6)$$

where (μ,ν,ξ) denotes the indices of the intervals along the three axes that enclose the query point. This simplification reduces the number of terms in the summation from $N_x \times N_y \times N_z$ to $(d_x+1)\times(d_y+1)\times(d_z+1)$, thereby making the B-spline interpolant evaluation highly efficient.

3 Design of the proposed parallel algorithms

Based on the well-established mathematical background of multivariate spline interpolation discussed above, we propose parallel algorithms to accommodate the increasing trend of distributing large datasets across multiple nodes. During the construction phase, the parallel algorithms integrate seamlessly with the dimension-by-dimension construction approach. During the evaluation phase, they effectively utilize the local support of B-spline basis functions. To our knowledge, such parallel algorithms have not been explicitly documented before.

3.1 Construction phase

The dimension-by-dimension approach for constructing multivariate B-spline interpolants offers an elegant framework that is well suited for parallel implementation. When a 3D volume is partitioned across multiple processes, each step in Eqs. (5a)-(5c) requires only that the axis along which the 1D interpolant is constructed be complete within each partition, while the interpolants at different locations on the other two axes are completely independent and can therefore be constructed sequentially or in parallel. This flexibility allows the 3D volume to be decomposed along a single axis, known as slab decomposition, or along two axes, known as pencil decomposition. When necessary, the partitioned volume can be redistributed across processes via all-to-all communication, effectively transposing the 3D volume and swapping a complete axis with a partitioned axis. Notably, this parallelization scheme is conceptually similar to those used in parallel 3D discrete transforms such as the Fast Fourier Transform (FFT), cosine/sine transforms, and Hartley transforms, where the multidimensional transforms are also performed dimension by dimension. Among these, parallel 3D FFT [1, 9, 10, 17, 26, 28, 39] is the most extensively studied and applied. It also supports both slab and pencil decompositions.

Algorithm 1 Construction of trivariate B-spline interpolant: Slab decomposition

Input: Values on grid points stored in $\mathbf{F}$, dimensions $N_x \times N_y \times N_z$, decomposed along the z-axis
Output: Coefficients stored in $\mathbf{C}$, dimensions $N_x \times N_y \times N_z$, decomposed along the x-axis

1: **Interpolant along x-axis:** Solve for the coefficients of $N_y \times N_z$ univariate B-spline interpolants along the x-axis (Eq. (5a)). Update $\mathbf{F}$ with $\mathbf{C}''$.
2: **Interpolant along y-axis:** Solve for the coefficients of $N_x \times N_z$ univariate B-spline interpolants along the y-axis (Eq. (5b)). Update $\mathbf{C}''$ with $\mathbf{C}'$.
3: **Redistribution:** Transpose between the z- and x-axes via all-to-all communication so that $\mathbf{C}'$ becomes decomposed along the x-axis.
4: **Interpolant along z-axis:** Solve for the coefficients of $N_x \times N_y$ univariate B-spline interpolants along the z-axis (Eq. (5c)). Update $\mathbf{C}'$ with $\mathbf{C}$.

Figure 2 presents the parallel algorithms for constructing a trivariate B-spline interpolant in a partitioned 3D volume. The complete sequential procedure for slab decomposition is given in **Algorithm 1**.

Algorithm 2 Construction of trivariate B-spline interpolant: Pencil decomposition

Input: $\mathbf{F}$, decomposed along the y- and z-axes
Output: $\mathbf{C}$, decomposed along the x- and y-axes

1: **Interpolant along x-axis**
2: **Redistribution:** Transpose between the y- and x-axes via all-to-all communication within z subgroups so that $\mathbf{C}''$ becomes decomposed along the x- and z-axes
3: **Interpolant along y-axis**
4: **Redistribution:** Transpose between the z- and y-axes via all-to-all communication within x subgroups so that $\mathbf{C}'$ becomes decomposed along the x- and y-axes
5: **Interpolant along z-axis**

For pencil decomposition, the steps for constructing univariate interpolants along each axis remain the same. The key difference is that it requires two redistribution steps. Each redistribution involves all-to-all communication restricted to subgroups of processes. For clarity, the complete sequential procedure is given in **Algorithm 2**.

Both slab decomposition and pencil decomposition are applicable to distributed-memory systems. Comparing **Algorithm 1** and **Algorithm 2**, the latter involves an additional all-to-all communication for data redistribution, resulting in a slightly higher processing cost. However, slab decomposition requires that the memory of a single node be sufficient to process an entire xy plane, whereas pencil decomposition only requires enough memory to process a single x-direction pencil. This allows pencil decomposition to process much larger 3D volumes.

Unlike most mainstream parallel FFT libraries [10, 17, 26, 28] which decompose the 3D volume as evenly as possible, `xspline3d` supports user-defined uneven partitioning for both slab and pencil decompositions. This flexibility allows users to tailor partition sizes according to available computational resources, such as memory capacity and processing power of individual processes. It helps maintain load balance and enhance parallel performance, particularly in heterogeneous systems where processes may have varying computational capabilities.

In addition, at each redistribution step where two axes are swapped, the uneven decomposition pattern along the source axis is propagated to the target axis, thereby preserving load balance consistently across redistributions. Specifically, for slab decomposition as described in **Algorithm 1**, before the redistribution in Step 3, the 3D volume is decomposed along the z-axis with process-specific partition sizes:

$$\sum_{i=1}^{N_p} n_z^{[i]} = N_z \tag{7a}$$

where N_p is the total number of processes. After the redistribution, the 3D volume becomes decomposed along the x-axis:

$$\sum_{i=1}^{N_p} n_x^{[i]} = N_x. \tag{7b}$$

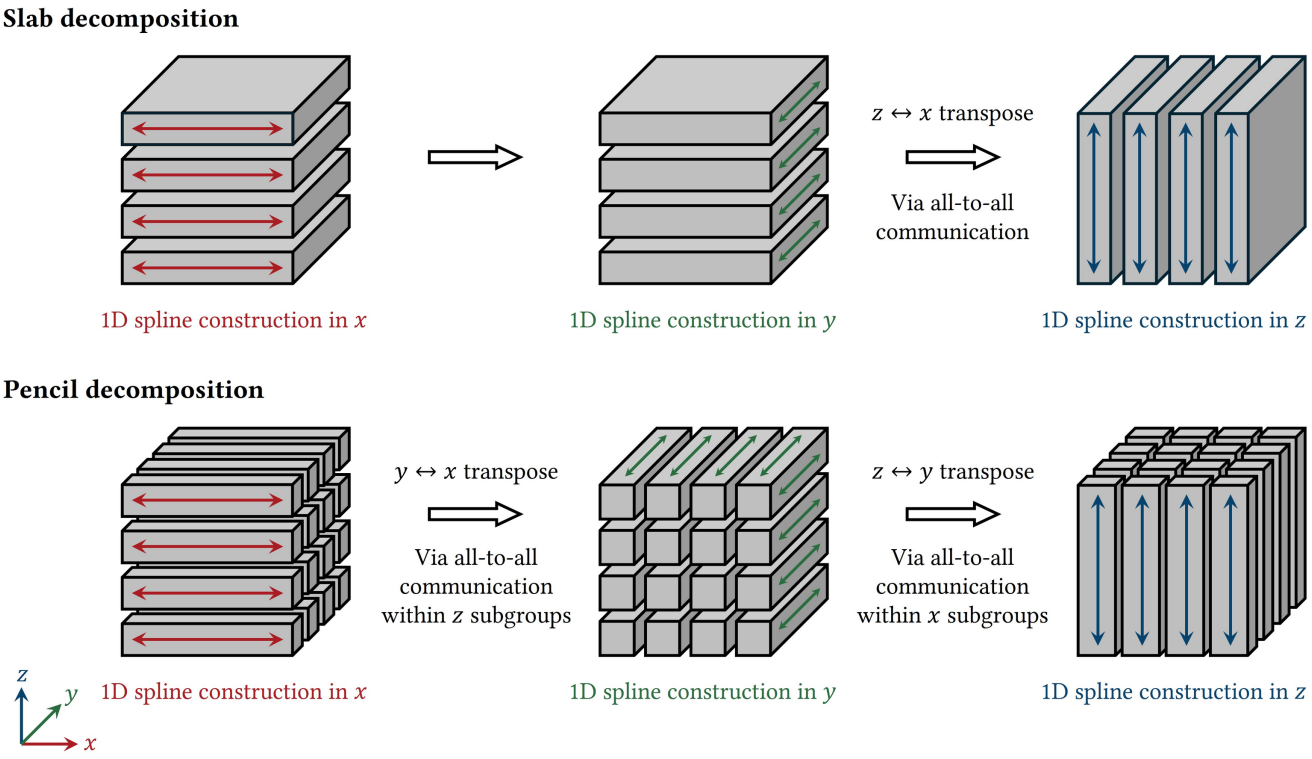

Figure 2: Parallel construction of a trivariate B-spline interpolant in a 3D volume subject to slab and pencil decompositions.

To preserve the uneven decomposition pattern, the ratio of local partition sizes before and after redistribution is maintained approximately constant across all processes:

$$\frac{n_x^{[i]}}{n_z^{[i]}} \approx \frac{N_x}{N_z}, \quad \forall i \in [1, N_p]. \tag{7c}$$

Similarly, for pencil decomposition as described in **Algorithm 2**, the 3D volume is decomposed across $N_p = N_{p1} \times N_{p2}$ processes. Before the first redistribution in Step 2, the decomposition is

$$\sum_{i=1}^{N_{p1}} n_y^{[i]} = N_y, \quad \sum_{j=1}^{N_{p2}} n_z^{[j]} = N_z. \tag{8a}$$

After Step 2, the decomposition becomes

$$\sum_{i=1}^{N_{p1}} n_x^{[i]} = N_x, \quad \sum_{j=1}^{N_{p2}} n_z^{[j]} = N_z. \tag{8b}$$

Here, the decomposition along the z-axis, $\{n_z^{[j]}\}$, remains unchanged, and the uneven decomposition pattern is approximately preserved by

$$\frac{n_x^{[i]}}{n_y^{[i]}} \approx \frac{N_x}{N_y}, \quad \forall i \in [1, N_{p1}]. \tag{8c}$$

After Step 4, the decomposition further becomes

$$\sum_{i=1}^{N_{p1}} n_x^{[i]} = N_x, \quad \sum_{j=1}^{N_{p2}} n_{y'}^{[j]} = N_y. \tag{8d}$$

This time, the decomposition of $\{n_x^{[j]}\}$ remains unchanged, and the uneven decomposition pattern is approximately preserved by

$$\frac{n_{y'}^{[j]}}{n_z^{[j]}} \approx \frac{N_y}{N_z}, \quad \forall j \in [1, N_{p2}]. \tag{8e}$$

3.2 Evaluation phase

In the parallel implementation, the 3D array $\mathbf{C}$ is partitioned across processes. During the evaluation phase, according to Eq. (6), given a set of query points, each process only needs to compute for query points whose contributing basis function indices, such as $[\mu - d_x, \mu]$ along the x-axis, overlap with its assigned partition of $\mathbf{C}$. If the contributing indices of a query point span multiple processes, the partial results from these processes are accumulated to obtain the final evaluated value.

4 Usage of xspline3d

The `xspline3d` library is written entirely in Python. Since it only depends on five well-established and widely used libraries, which are `SciPy`, `NumPy`, `Numba`, `threadpoolctl`, and `mpi4py`, it can be readily deployed on a broad range of HPC systems.

In spline-related computation, xspline3d employs SciPy's interpolate module to perform some fundamental tasks, including configuring knot vectors, generating B-spline basis functions, and constructing 1D spline interpolants. Building on this foundation, xspline3d has its own scripts for the dimension-by-dimension trivariate interpolant construction as described in Eqs. (5a)-(5c), evaluation based on the local support of B-spline basis functions as described in Eq. (6), and its unique parallel algorithms as shown in Fig. 2.

xspline3d sets the spline order identical across all three axes: $d = d_x = d_y = d_z$. It internally supports splines of arbitrary order and externally exposes three options via an argument named method: "slinear" for $d = 1$, "cubic" for $d = 3$, and "quintic" for $d = 5$. xspline3d provides three interpolators: SingleProcessInterpolator (SPI) implementing single-process trivariate spline interpolation, SlabDecompInterpolator (SDI) supporting slab decomposition where the input 3D volume is partitioned along the z-axis, and PencilDecomp Interpolator (PDI) supporting pencil decomposition where the input 3D volume is partitioned along the y- and z-axes. Demo Python scripts using these three interpolators are shown below.

Listing 1: Single-process interpolation

```
interp = SingleProcessInterpolator(coordinates,
    data, method="cubic")
values = interp(points)
```

Here, coordinates is a sequence of three 1D arrays, ($[x_1, x_2, \ldots, x_{N_x}]$, $[y_1, y_2, \ldots, y_{N_y}]$, $[z_1, z_2, \ldots, z_{N_z}]$), which define the positions of the 3D rectilinear grid points. The positions along each axis do not need to be equally spaced, and the positions along different axes are independent. data is a 3D array consisting of values on the grid points. points is an array of shape $(N, 3)$, representing the coordinates of N query points. The interpolated values evaluated on the query points are returned in values as a 1D array.

Listing 2: Interpolation in slab-decomposed data

```
interp_slab = SlabDecompInterpolator(comm,
    coordinates, data_z, method="cubic")
if is_root:
    values = interp_slab(points)
else:
    _ = interp_slab(None)
```

Here, coordinates remains the same as in single-process interpolation. For user convenience, each rank is provided with the full coordinate information of the entire 3D volume, without rank-specific tailoring. comm is the MPI communicator. data_z is the local partition of data, obtained via slab decomposition along the z-axis. Users do not need to explicitly pass arguments regarding the slab decomposition layout to the interpolator. Instead, the interpolator automatically detects the layout based on the local shapes of data_z across all processes. During evaluation, only the root process receives all the query points and provides all the interpolation results. This root-driven I/O strategy ensures that all processes share an identical set of query points and avoids redundant storage of interpolation results. The same strategies of full-coordinate replication, automatic layout detection, and root-driven I/O also apply to pencil-decomposed interpolation.

Listing 3: Interpolation in pencil-decomposed data

```
interp_pencil = PencilDecompInterpolator(comm,
    coordinates, data_yz, method="cubic")
if is_root:
    values = interp_pencil(points)
else:
    _ = interp_pencil(None)
```

Here, data_yz is the local partition of data obtained via pencil decomposition along the y- and z-axes. In ascending order of MPI ranks, the partition indices along the z-axis should change more rapidly than those along the y-axis. The interpolator automatically detects the pencil decomposition layout according to the local shapes of data_yz across all processes.

5 Performance evaluation

We evaluated the performance of xspline3d on three platforms. The first is a workstation equipped with dual Intel Xeon Gold 6336Y processors (48 cores, 2.4 GHz) and 503 GB of memory, hereafter referred to as the Workstation. The second is the HOKUSAI BigWaterfall2 (HBW2) system maintained by RIKEN, which comprises 312 nodes, each with two Intel Xeon Max CPUs (112 cores, 1.9 GHz) and 128 GB of memory. The third is the supercomputer Fugaku, of which each node employs a Fujitsu A64FX CPU with 48 computing cores and 32 GB of memory.

For reference, we compared the performance of xspline3d with SciPy RGI, which is perhaps the most readily accessible tool for most users. Our tests cover SciPy versions from 1.13.0, released in April 2024, to the latest 1.16.1, current as of August 2025. Across these versions, the core functions related to multivariate spline interpolation in RGI have remained unchanged. In all tests, we used the default solver and solver arguments provided by SciPy RGI.

In all tests, the spline functions were set as cubic ($d = 3$).

5.1 Resident memory usage

We first measured resident memory usage. For multivariate spline interpolation, the memory bottleneck arises during interpolant construction, as the entire 3D volume must be processed simultaneously. In contrast, memory usage during evaluation is not a primary concern: If the set of query points is large, it can be divided into batches and processed sequentially, thereby avoiding peak memory pressure. Consequently, our resident memory usage tests focused on the construction phase. These tests were performed on the Workstation.

Figure 3 shows the peak resident memory of SciPy RGI and xspline3d during interpolant construction for 3D volumes of 256^3 and 512^3. SciPy RGI supports only single-process execution. For xspline3d, all its three interpolators, SPI, SDI, and PDI, show nearly identical memory usage for the same number of processes and are therefore not distinguished in the figure. Note that SPI is applicable only to a single process.

For xspline3d, the memory usage increases slightly with the number of processes. The rate of increase is consistent for two 3D volumes, independent of their size, suggesting that this increase is not due to data redundancy across processes but rather to per-process overhead.

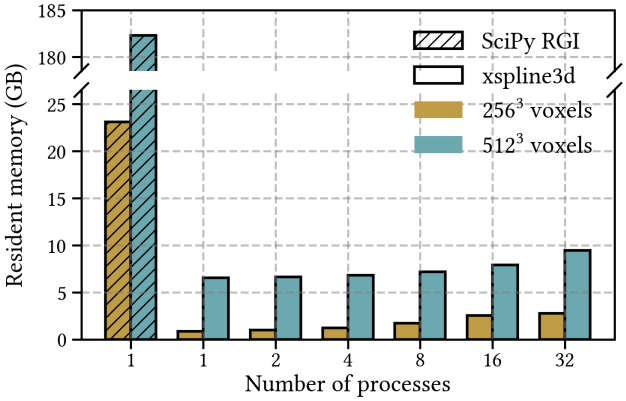

Figure 3: Peak resident memory of `Scipy RGI` and `xspline3d` during interpolant construction.

In comparison, `SciPy RGI` requires significantly more memory during interpolant construction, which is approximately 23 GB for a 256^3 volume and 184 GB for a 512^3 volume. In fact, this considerable memory requirement was one of our motivations for developing `xspline3d`, which reduces total memory usage by more than a factor of 25.

5.2 Single-process computational efficiency

We next evaluated the computational efficiency. Since `SciPy RGI` operates only in single-process mode, the comparison was limited to the single-process `SPI` in `xspline3d`. The evaluation reflects only the spline-related computations, excluding the additional overhead of inter-process communications. This comparison is also informative for the two multi-process interpolators, `SDI` and `PDI`, in `xspline3d`, as they share the same spline-related computation functions.

Tests were conducted on the Workstation and on a single node of Fugaku, in a 3D volume of 256^3 and evaluating 256^3 query points in a single batch. The results are shown in Fig. 4. Each test was repeated ten times, with the root-mean-square error (RMSE) indicated by error bars. These error bars are mostly invisible, reflecting the high stability of the test results. The same procedure of repeating each test ten times and calculating the RMSE also applies to Figs. 5 and 6, where the error bars are similarly negligible.

During the construction phase, `xspline3d SPI` achieves more than a 20-fold speedup over `SciPy RGI` on both platforms. This improvement arises from the different approaches to construct the multivariate tensor-product B-spline interpolant. `xspline3d` employes the dimension-by-dimension approach, as expressed in Eqs. (5a)-(5c), with a computational complexity of $O(N_x N_y N_z d^2)$. In contrast, `SciPy RGI` constructs the interpolant by solving a large linear system of size $(N_x N_y N_z) \times (N_x N_y N_z)$, as described in Eq. (4). Although the large linear system is sparse and can be solved using sparse iterative solvers, such as the Generalized Conjugate Residual with inner Orthogonalization and outer Truncation (GCROT(m,k)) algorithm [13] employed by `SciPy RGI` by default, the computational complexity remains significantly higher than that of the dimension-by-dimension approach. It is worth emphasizing that the sparse iterative solver provides only approximate solutions, whereas the banded solver employed in the dimension-by-dimension approach yields the exact solution. In other words, `xspline3d` provides higher interpolation accuracy than `SciPy RGI` in addition to higher computational efficiency. In practical implementation, this accuracy difference depends on the intrinsic shape of the underlying function as well as the tolerance parameters set for the sparse iterative solver.

In addition, during the construction phase, both `SciPy RGI` and `xspline3d SPI` provide limited multithreading support, and their construction times decrease slightly as the number of threads increases. For `xspline3d`, this multithreading support is not explicitly implemented in its Python scripts. Instead, it is managed by the `LAPACK gbsv` subroutine, which is wrapped and called by `SciPy` to solve for the B-spline coefficients of a mesh of 1D spline interpolants simultaneously.

During the evaluation phase, `xspline3d SPI` is approximately three times faster than `SciPy RGI`. Moreover, unlike `SciPy RGI`, which does not support multithreading, `xspline3d` evaluates the query points in parallel using `Numba`'s multithreaded `prange` loop.

5.3 Multi-process computational efficiency

To enable direct comparison with the single-process results in Fig. 4, the first set of tests on the multi-process interpolators `SDI` and `PDI` was also conducted on the Workstation and on a single node of Fugaku, in a 3D volume of 256^3 and evaluating 256^3 query points in a single batch. These tests employed a hybrid parallelization scheme combining multiprocessing and multithreading. For both `SDI` and `PDI`, each process was assigned an evenly decomposed partition of the 3D volume. In the case of `PDI`, the numbers of partitions along the y- and z-axes were set equal, and correspondingly its tests were performed only with process counts that are perfect squares, such as 1, 4, and 16.

The results are presented in Fig. 5. During both the construction and evaluation phases, and on both the Workstation and Fugaku, when the number of threads increases, the execution time slightly decreases. This is consistent with the multithreading behavior of `SPI` in Fig. 4. In addition, as the number of processes increases, the execution time reasonably decreases. The rate of reduction becomes smaller at larger process counts, likely due to the increasing cost of inter-process communication. Overall, `xspline3d` demonstrates improved performance with more threads and processes under a hybrid parallelization scheme, indicating its suitability for HPC platforms.

Subsequently, we tested on a larger 3D volume of 1024^3 partitioned across multiple nodes of HBW2 and Fugaku. During the construction phase, it requires at least 56 GB of memory. For evaluation, all query points were grouped into 16 batches, each containing 1024^2 points. This approximates the application scenario of extracting 2D slicing images one by one from a 3D volume. On HBW2, we configured 8 processes per node with a single thread per process, and allocated 128 GB of memory to each task. On Fugaku, since each node has 4 core-memory groups (CMGs), each integrating 12 computational cores and 8 GB High Bandwidth Memory (HBM2), we configured 4 processes per node with 12 threads per process, with each node providing 32 GB of memory to the task. The measured construction time and evaluation time are presented

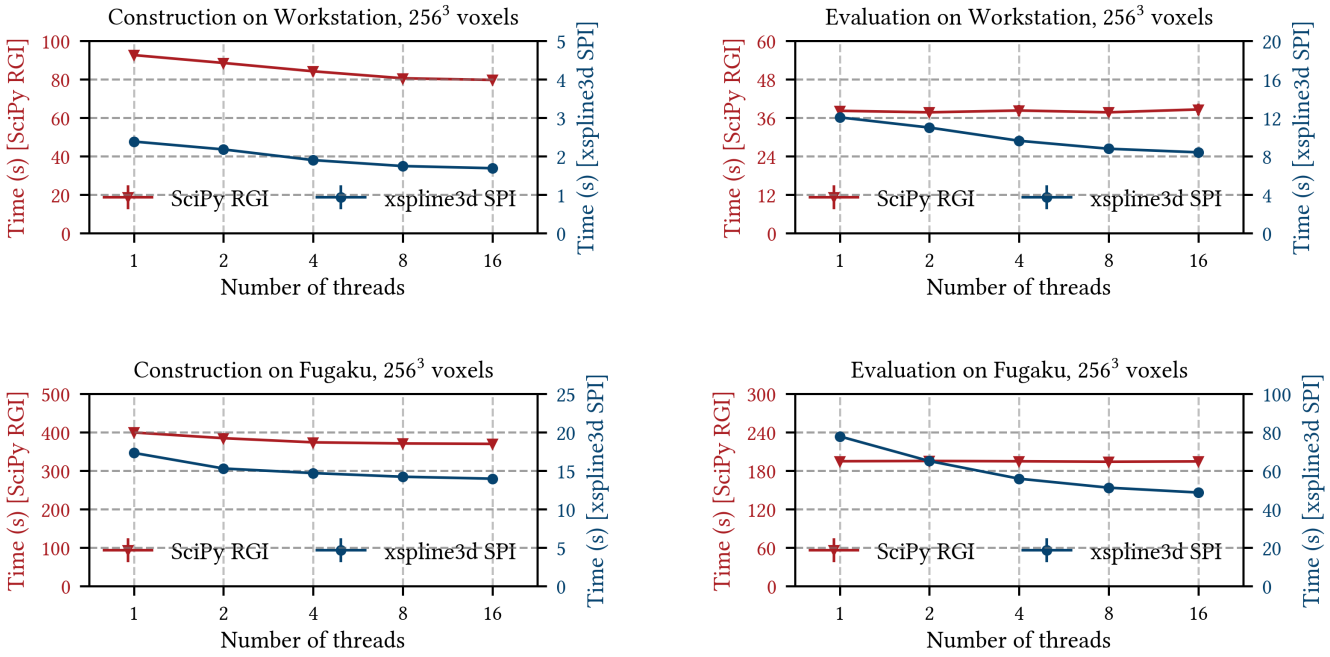

Figure 4: Single-process execution time of `SciPy RGI` and `xspline3d SPI`. Since their values differ by an order of magnitude, they are plotted on separate y-axes: `SciPy RGI` on the left and `xspline3d SPI` on the right.

in Fig. 6. An anomaly appears at 128 processes on HBW2, where the evaluation time increases unexpectedly. This is likely because, at this scale, the overhead of inter-process communication outweighs the gains from additional computational resources. In fact, this behavior is also influenced by the total number of query points and the batch size, which affect the balance between inter-process communication time and computation time.

Beyond the 1024^3 volume, on Fugaku we also successfully constructed large-scale trivariate spline interpolants in volumes of 2048^3 and 4096^3, which require at least 448 GB and 3,584 GB of memory, respectively. In these tests, we employed the same hybrid parallelization configurations as described above. The allocated resources and corresponding construction time are summarized in Table 1.

Overall, these tests demonstrate the feasibility of applying `xspline3d` to large-scale 3D volumes distributed across multiple nodes on various HPC platforms, effectively overcoming the critical single-node memory bottleneck. Notably, the validated large-scale volume of 4096^3 is becoming increasingly common in modern scientific computing, such as in high-resolution computed tomography [3, 43].

6 Discussion

All benchmark tests in this work employed cubic splines ($d = 3$). In fact, the cubic order in spline interpolation is commonly used because it offers a good balance between smoothness and computational cost. In many documents, cubic spline interpolation is sometimes loosely called "cubic interpolation", which can easily be confused with other cubic methods, such as cubic Hermite interpolation (sometimes referred to as cubic Hermite spline interpolation), cubic convolution interpolation [15], and tricubic interpolation [16, 41]. Non-expert users should carefully distinguish which "cubic" method they are actually using and understand its interpolation properties. The main difference between these methods lies in the continuity order they guarantee. While most other cubic methods ensure only C^1 continuity, cubic spline interpolation guarantees C^2 continuity, providing higher smoothness. This improves the accuracy of derivative approximations and enhances the reliability of subsequent computations. Furthermore, cubic Hermite interpolation is mostly applied in 1D space and is rarely extended to higher-dimensional spaces, and tricubic interpolation is restricted to 3D space.

In practical engineering applications, a reasonable compromise for performing spline interpolation in partitioned multidimensional volumes is to introduce overlapping halo regions between partitions. The interpolant is then constructed independently on each partitioned volume and subsequently concatenated. This approach avoids the need to solve a global system in distributed memory environments, and the magnitude of interpolation artifacts decreases with larger halo regions. However, it suffers from several limitations. First, as the number of partitions increases, the proportion of overlapping halo regions grows, resulting in redundant computations and reduced overall efficiency. Second, numerical artifacts tend to be more pronounced at regular partition boundaries, potentially introducing structured errors. This is particularly problematic for iterative algorithms, where such artifacts may accumulate or amplify over iterations. Third, the magnitude of these artifacts strongly

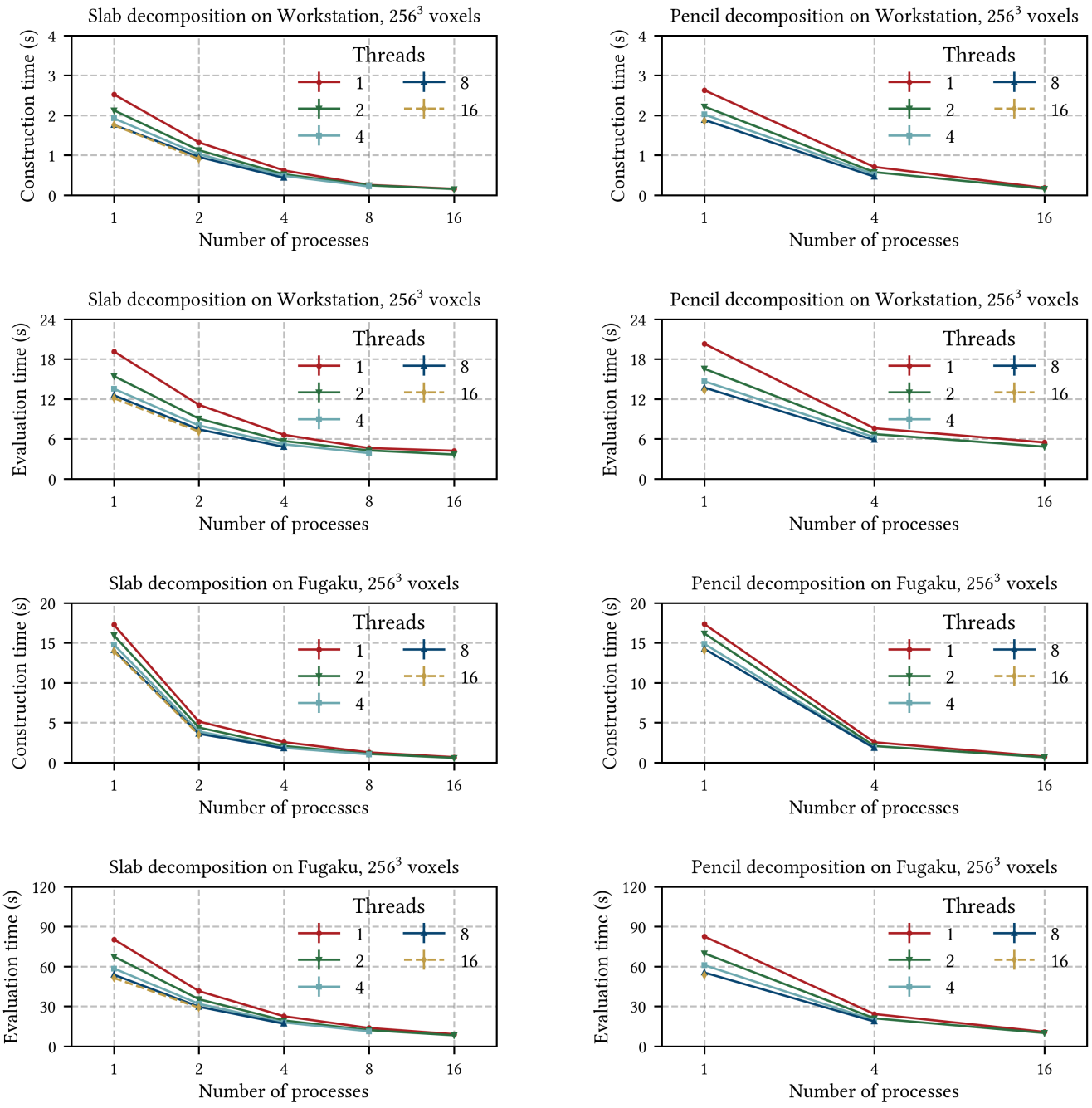

Figure 5: Execution time of the two `xspline3d` multi-process interpolators on the Workstation and a single Fugaku node under a hybrid multiprocessing-multithreading scheme.

depends on the properties of the underlying function, making it difficult to predict their impact across diverse application scenarios. In contrast, the parallelization approach in `xspline3d` is mathematically rigorous and theoretically free from approximation error. It also operates more efficiently by avoiding overlapping halo regions and the associated computational overhead.

In `xspline3d`, the interpolation task is divided into two phases: construction and evaluation. Once the multivariate spline interpolant is constructed, it can be efficiently reused to evaluate an arbitrary number of query points. On the other hand, there exists an alternative approach that does not explicitly construct the global multivariate interpolant. Instead, for each query point, it

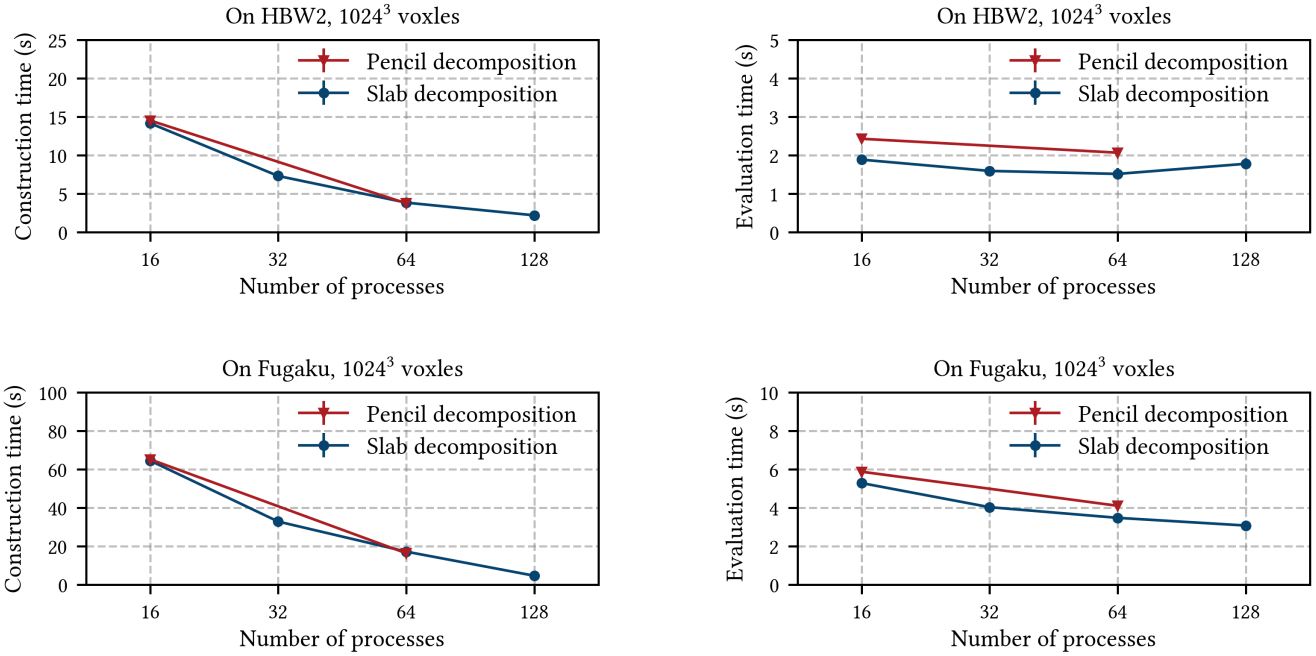

Figure 6: Execution time of the two `xspline3d` multi-process interpolators on multiple nodes of HBW2 and Fugaku.

Table 1: Construction time of large-scale trivariate spline interpolants on Fugaku

Volume size	Number of processes	Construction time (s)	
		Slab decomposition	Pencil decomposition
2048^3	256	42.0	40.0
2048^3	1024	6.7	5.6
4096^3	1024	115.0	106.0
4096^3	4096	29.5	25.7

performs 1D interpolations sequentially along each axis. For example, given 3D grid data points $f(x_i, y_j, z_k)$, the interpolation at a query point (x_q, y_q, z_q) proceeds as follows. First, $N_y \times N_z$ 1D interpolations along the x-axis are performed to obtain values on the yz-plane passing through the query point, $f(x_q, y_j, z_k)$. Next, on this plane, N_z 1D interpolations along the y-axis yield values on the line passing through the query point, $f(x_q, y_q, z_k)$. Finally, a single 1D interpolation along the z-axis determines the interpolated value at the query point, $f(x_q, y_q, z_q)$. It is easy to prove that this approach yields theoretically identical results. In fact, it was historically adopted by `SciPy RGI` from version 1.9.0, released in July 2022, to version 1.12.0, released in January 2024, and has been retained as a legacy method in subsequent versions. However, unlike the approach in `xspline3d`, it can only precompute $N_y \times N_z$ 1D interpolants. At evaluation, each query point requires constructing an additional $(N_z + 1)$ 1D interpolants, which cannot be reused for other scattered query points. As a result, although the precomputation burden is lighter, evaluating a large number of query points is time-consuming. This approach can also be implemented in parallel via slab decomposition along either the y- or z-axis. After precomputing the $N_y \times N_z$ 1D spline interpolants along the x-axis and transforming them from B-form into ppform, an all-to-all communication redistributes the 3D volume so that it is decomposed along the x-axis. Computation on query points can then be performed in parallel. Although we have fully validated this parallel approach, we did not include it in the current version of `xspline3d` because its suitable scenarios, where only a very small number of query points need to be interpolated in a large-scale 3D volume, are uncommon in scientific computing.

7 Conclusion

In the present work, we have developed `xspline3d`, a Python library for MPI-based parallel trivariate B-spline interpolation in large-scale 3D datasets partitioned across multiple processes and distributed memory via slab and pencil decompositions. We have evaluated its performance on three platforms, an individual workstation, HBW2 Intel Xeon cluster, and the supercomputer Fugaku. Compared to the widely used `SciPy RGI` whose spline interpolation is restricted to single-process, shared-memory execution, `xspline3d`

achieves significantly higher computational efficiency and substantially lower memory usage. Its computational efficiency can be further enhanced under a hybrid parallelization scheme combining multiprocessing and multithreading. In our tests, by utilizing the vast distributed memory on Fugaku, `xspline3d` successfully performed spline interpolation in 3D volumes as large as 4096^3, which requires 512 GB of memory for storage and at least 3.5 TB for interpolation. It is also capable of handling even larger volumes. At present, driven by our needs in structural biology and by common application scenarios in other scientific fields, the initial release of `xspline3d` supports interpolation only in 3D space. In the future, it can be readily extended to higher-dimensional spaces when necessary. Given that multivariate spline interpolation is a fundamental operation in many areas of scientific computing and data science, and that the size of datasets has been increasing dramatically in recent years, `xspline3d`, owing to its high efficiency, low memory usage, and MPI-based parallelization, has the potential to evolve into a popular utility library on many HPC platforms.

Acknowledgments

This work was in part supported by FOCUS for Establishing Supercomputing Center of Excellence. The computer resources were provided by RIKEN Advanced Center for Computing and Communication (HOKUSAI BigWaterfall2 project RB240025) and RIKEN Center for Computational Science (Fugaku supercomputer).

References

[1] Orlando Ayala and Lian-Ping Wang. 2013. Parallel implementation and scalability analysis of 3D Fast Fourier Transform using 2D domain decomposition. *Parallel Comput.* 39, 1 (Jan. 2013), 58–77. doi:10.1016/j.parco.2012.12.002

[2] Olivier Botella and Karim Shariff. 2003. B-spline Methods in Fluid Dynamics. *International Journal of Computational Fluid Dynamics* 17, 2 (Jan. 2003), 133–149. doi:10.1080/1061856031000104879

[3] Peng Chen, Mohamed Wahib, Xiao Wang, Takahiro Hirofuchi, Hirotaka Ogawa, Ander Biguri, Richard Boardman, Thomas Blumensath, and Satoshi Matsuoka. 2021. Scalable FBP decomposition for cone-beam CT reconstruction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, New York, NY, 1–16. doi:10.1145/3458817.3476139

[4] Roberto Cotesta, Sylvain Marsat, and Michael Pürrer. 2020. Frequency-domain reduced-order model of aligned-spin effective-one-body waveforms with higher-order modes. *Physical Review D* 101, 12 (June 2020), 124040. doi:10.1103/PhysRevD.101.124040

[5] Carl de Boor. 2001. *A Practical Guide to Splines* (revised ed.). Applied Mathematical Sciences, Vol. 27. Springer, New York.

[6] Carl de Boor. 2004. *Spline Toolbox for Use with MATLAB: User's Guide* (3rd. ed.). The MathWorks, Natick, MA.

[7] SciPy Developers. 2025. *RegularGridInterpolator documentation*. Retrieved August 24, 2025 from https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RegularGridInterpolator.html

[8] Paul Dierckx. 1993. *Curve and Surface Fitting with Splines*. Oxford University Press, Oxford. doi:10.1093/oso/9780198534419.001.0001

[9] P. Dmitruk, L.-P. Wang, W. H. Matthaeus, R. Zhang, and D. Seckel. 2001. Scalable parallel FFT for spectral simulations on a Beowulf cluster. *Parallel Comput.* 27, 14 (Dec. 2001), 1921–1936. doi:10.1016/S0167-8191(01)00120-X

[10] Matteo Frigo and S. G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (Feb. 2005), 216–231. doi:10.1109/JPROC.2004.840301

[11] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations* (4th. ed.). Johns Hopkins University Press, Baltimore, MD.

[12] Bjarne Grimstad et al. 2015. *SPLINTER: a library for multivariate function approximation with splines*. Retrieved August 24, 2025 from http://github.com/bgrimstad/splinter

[13] Jason E. Hicken and David W. Zingg. 2010. A Simplified and Flexible Variant of GCROT for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific Computing* 32, 3 (June 2010), 1672–1694. doi:10.1137/090754674

[14] S. Jonić, C. O. S. Sorzano, P. Thévenaz, C. El-Bez, S. De Carlo, and M. Unser. 2005. Spline-based image-to-volume registration for three-dimensional electron microscopy. *Ultramicroscopy* 103, 4 (July 2005), 303–317. doi:10.1016/j.ultramic.2005.02.002

[15] R. Keys. 1981. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, 6 (Dec. 1981), 1153–1160. doi:10.1109/TASSP.1981.1163711

[16] F. Lekien and J. Marsden. 2005. Tricubic interpolation in three dimensions. *Internat. J. Numer. Methods Engrg.* 63, 3 (March 2005), 455–471. doi:10.1002/nme.1296

[17] Ning Li and Sylvain Laizet. 2010. 2DECOMP & FFT—A Highly Scalable 2D Decomposition Library and FFT Interface. In *Cray User Group 2010 Proceedings*. Cray User Group, Edinburgh, 1–13. https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/16-18Thursday/17B-CUG2010-paper-Ning_Li.pdf

[18] Andrei V. Malevsky. 1996. Spline-Characteristic Method for Simulation of Convective Turbulence. *J. Comput. Phys.* 123, 2 (Feb. 1996), 466–475. doi:10.1006/jcph.1996.0037

[19] Benjamin Margolis and Kenneth Lyons. 2019. ndsplines: A Python Library for Tensor-Product B-Splines of Arbitrary Dimension. *Journal of Open Source Software* 4, 42 (Oct. 2019), 1745. doi:10.21105/joss.01745

[20] Erik H. W. Meijering, Wiro J. Niessen, and Max A. Viergever. 2001. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis* 5, 2 (June 2001), 111–126. doi:10.1016/S1361-8415(00)00040-2

[21] Jenny Margareta Mørk, Tor Nordam, and Siren Rühs. 2025. *Handling discontinuities in numerical ODE methods for Lagrangian oceanography*. egusphere:2025-2109 doi:10.5194/egusphere-2025-2109

[22] Miki Nakano, Osamu Miyashita, Slavica Jonic, Changyong Song, Daewoong Nam, Yasumasa Joti, and Florence Tama. 2017. Three-dimensional reconstruction for coherent diffraction patterns obtained by XFEL. *Journal of Synchrotron Radiation* 24, 4 (July 2017), 727–737. doi:10.1107/S1600577517007767

[23] Tor Nordam and Rodrigo Duran. 2020. Numerical integrators for Lagrangian oceanography. *Geoscientific Model Development* 13, 12 (Dec. 2020), 5935–5957. doi:10.5194/gmd-13-5935-2020

[24] Wendy A. Okolo, Benjamin W. Margolis, Sarah N. D'souza, and Jeffrey D. Barton. 2020. Pterodactyl: Development and comparison of control architectures for a mechanically deployed entry vehicle. In *AIAA Scitech 2020 Forum*. Orlando, FL, 1012. doi:10.2514/6.2020-1012

[25] Nathaniel Overton, Xinfeng Gao, and Stephen Guzik. 2017. Spline Interpolation for A Fourth-Order Adaptive Mesh Refinement Algorithm on Arbitrary Mapped Grids. In *23rd AIAA Computational Fluid Dynamics Conference*. Denver, CO, 3107. doi:10.2514/6.2017-3107

[26] Dmitry Pekurovsky. 2012. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM Journal on Scientific Computing* 34, 4 (Aug. 2012), C192–C209. doi:10.1137/11082748X

[27] Les Piegl and Wayne Tiller. 1997. *The NURBS Book* (latest ed.). Springer, Berlin. doi:10.1007/978-3-642-59223-2

[28] Michael Pippig. 2013. PFFT: An Extension of FFTW to Massively Parallel Architectures. *SIAM Journal on Scientific Computing* 35, 3 (May 2013), C213–C236. doi:10.1137/120885887

[29] Ali Punjani, John L. Rubinstein, David J. Fleet, and Marcus A. Brubaker. 2017. cryoSPARC: algorithms for rapid unsupervised cryo-EM structure determination. *Nature Methods* 14, 3 (March 2017), 290–296. doi:10.1038/nmeth.4169

[30] Michael Pürrer. 2018. *TPI: Tensor Product Interpolation Package for Python*. Retrieved August 24, 2025 from https://github.com/mpuerrer/TPI

[31] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. 1999. Nonrigid registration using free-form deformations: application to breast MR images. *IEEE Transactions on Medical Imaging* 18, 8 (Aug. 1999), 712–721. doi:10.1109/42.796284

[32] Eduard Schreibmann, George T. Y. Chen, and Lei Xing. 2006. Image interpolation in 4D CT using a BSpline deformable registration model. *International Journal of Radiation Oncology, Biology, Physics* 64, 5 (April 2006), 1537–1550. doi:10.1016/j.ijrobp.2005.11.018

[33] Friederike A. Schulte, Floor M. Lambers, Thomas L. Mueller, Martin Stauber, and Ralph Müller. 2014. Image interpolation allows accurate quantitative bone morphometry in registered micro-computed tomography scans. *Computer Methods in Biomechanics and Biomedical Engineering* 17, 5 (2014), 539–548. doi:10.1080/10255842.2012.699526

[34] Larry Schumaker. 2007. *Spline Functions: Basic Theory* (3rd. ed.). Cambridge University Press, Cambridge. doi:10.1017/CBO9780511618994

[35] Yoshinta Setyawati, Michael Pürrer, and Frank Ohme. 2020. Regression methods in waveform modeling: a comparative study. *Classical and Quantum Gravity* 37, 7 (April 2020), 075012. doi:10.1088/1361-6382/ab693b

[36] Dmytro Shulga, Oleksii Morozov, and Patrick Hunziker. 2017. A Tensor B-Spline Approach for Solving the Diffusion PDE With Application to Optical Diffusion Tomography. *IEEE Transactions on Medical Imaging* 36, 4 (April 2017), 972–982. doi:10.1109/TMI.2016.2641500

[37] Dmytro Shulga, Oleksii Morozov, and Patrick Hunziker. 2018. Solving 3-D PDEs by Tensor B-Spline Methodology: A High Performance Approach Applied to Optical Diffusion Tomography. *IEEE Transactions on Medical Imaging* 37, 9 (Sept.

2018), 2115–2125. doi:10.1109/TMI.2018.2819901

[38] C.O.S. Sorzano, R. Marabini, J. Velázquez-Muriel, J.R. Bilbao-Castro, S.H.W. Scheres, J.M. Carazo, and A. Pascual-Montano. 2004. XMIPP: a new generation of an open-source image processing package for electron microscopy. *Journal of Structural Biology* 148, 2 (Nov. 2004), 194–204. doi:10.1016/j.jsb.2004.06.006

[39] Daisuke Takahashi. 2010. An Implementation of Parallel 3-D FFT with 2-D Decomposition on a Massively Parallel Cluster of Multi-core Processors. In *Parallel Processing and Applied Mathematics*. Springer, Berlin, 606–614. doi:10.1007/978-3-642-14390-8_63

[40] Dimitry Tegunov and Patrick Cramer. 2019. Real-time cryo-electron microscopy data preprocessing with Warp. *Nature Methods* 16, 11 (Oct. 2019), 1146–1152. doi:10.1038/s41592-019-0580-y

[41] Paul Walker, Ulrich Krohn, and David Carty. 2019. ARBTools: A Tricubic Spline Interpolator for Three-Dimensional Scalar or Vector Fields. *Journal of Open Research Software* 7, 1 (April 2019), 12. doi:10.5334/jors.258

[42] Jacob Williams. 2015. *bspline-fortran: Multidimensional B-Spline Interpolation of Data on a Regular Grid.* Retrieved August 24, 2025 from https://github.com/jacobwilliams/bspline-fortran

[43] Du Wu, Peng Chen, Xiao Wang, Issac Lyngaas, Takaaki Miyajima, Toshio Endo, Satoshi Matsuoka, and Mohamed Wahib. 2024. Real-time High-resolution X-Ray Computed Tomography. In *Proceedings of the 38th ACM International Conference on Supercomputing*. ACM Press, New York, NY, 110–123. doi:10.1145/3650200.3656634

[44] Youbing Yin, Eric A. Hoffman, and Ching-Long Lin. 2009. Mass preserving nonrigid registration of CT lung images using cubic B-spline. *Medical Physics* 36, 9 (Sept. 2009), 4213–4222. doi:10.1118/1.3193526

[45] Orestis Zachariadis, Andrea Teatini, Nitin Satpute, Juan Gómez-Luna, Onur Mutlu, Ole Jakob Elle, and Joaquín Olivares. 2020. Accelerating B-spline interpolation on GPUs: Application to medical image registration. *Computer Methods and Programs in Biomedicine* 193 (Sept. 2020), 105431. doi:10.1016/j.cmpb.2020.105431

[46] Wenyang Zhao, Osamu Miyashita, and Florence Tama. 2025. *xspline3d: MPI-based spline interpolation in segmented 3D volumes.* Retrieved August 24, 2025 from https://github.com/TamaLab/xspline3d