SF2A 2025
A. Siebert, K. Baillié, M. Béthermin, F. Cantalloube, E. Josselin, N. Lagarde, J. Malzac, J. Richard, L. Selliez, O. Venot (eds)

HIGH-PERFORMANCE INTEGRAL TRANSFORMS FOR THE NEXT GENERATION OF ASTROPHYSICAL CODES

T. Maamaatuaiahutapu[2,1] and V. Reverdy[2,3,4]

Abstract. The Fast Fourier Transform and signal processing are central to the analysis of large data sets in astrophysics. In particular, modern astrophysics require the processing of increasingly large data. As the latter increase in size, so does the need for scalable, high-performance signal processing algorithms.

In this work, we delve into some details of our implementation strategy around the computation of multidimensional Fourier transforms. This is a dimension agnostic, modular and generic framework that makes use of the most recent additions to the C++ standard library. We detail how the separation of concerns and a multi-level design helps to achieve modularity and genericity while maintaining efficiency. Finally, each level of this design structure is naturally compatible with various forms of parallelism, allowing seamless integration in high-performance simulation codes.

Keywords: FFT, high-performance computing, computational astrophysics, numerical methods, C++

1 Introduction

Multidimensional Fourier transforms (MDFFTs) are at the foundation of numerical computations in astrophysics, imaging, and scientific computing (see Sugiyama et al. 2019; Kumar et al. 2019; Press 2007). As datasets grow in size, efficient and modular implementations become increasingly critical. Traditional approaches are often bundled into black-box libraries (Frigo & Johnson 2005), which limits flexibility and parallelization. In this work, we present a general scheme for multidimensional FFTs in modern C++, leveraging recent language features. Our approach separates the multidimensional iteration logic from the 1D FFT kernel, using recursive function objects and multi-dimensional iterators. This enables flexible, modular, and type-safe computations. This design naturally supports concurrency at multiple levels and forms a blueprint for extending the same structure to other integral transforms. The paper proceeds as follows: we first introduce the building blocks used for this scheme then describe the recursive procedure and patterns involved in the computation, discuss parallelism considerations, and finally conclude with potential extensions. In doing so, we follow the presentation style typical of the C++ standard, exposing the design as a set of composable and modular schemes.

2 Abstractions for multidimensional FFTs

MDFFTs are generally implemented as a series of one dimensional FFTs along each axis of a multidimensional array as illustrated in Fig. 1.

To express this generically, we introduce a set of abstractions that separate the multidimensional axis logic from the 1D-FFT kernel itself. At the lowest level, axis iterators allow traversal along arbitrary array dimensions. At the next level, axis wise function objects apply 1D FFTs across slices. Finally, a multidimensional driver composes these into a full N-dimensional transform. Our goal is to avoid a rigid architecture by leveraging modern language features that support generic and modular software design. We build on recent additions to the C++ standard (ISO/IEC 2024) like C++ concepts, `std::mdspan`, and `std::ranges` to express theses degrees of freedom in a type safe way that doesn't sacrifice performance.

[1] Université Claude-Bernard Lyon 1, 69622 Villeurbanne, FRANCE
[2] LAPP, Université de Savoie, IN2P3, 74940, FRANCE
[3] LUX, observatoire de Paris, Université PSL, Sorbonne Université, CNRS, 92190 Meudon, FRANCE
[4] NCSA, University of Illinois, Urbana, IL, USA

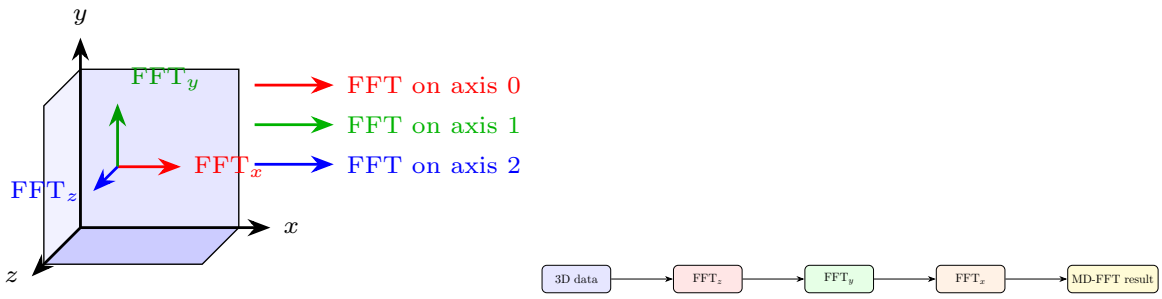

Fig. 1. Left: Application of one-dimensional FFTs along each axis of a three-dimensional array. The cube represents multidimensional input data, with independent FFTs performed on all slices aligned along the x, y and z directions. **Right:** Schematic decomposition of a three-dimensional FFT into successive one-dimensional transforms along each axis: the 3D input data are first transformed along z, then along y, and finally along x, yielding the full MD-FFT.

The nature of containers in C++ and programming in general can vary greatly in implementation. A container can be anything ranging from arrays, vectors, lists, tuples, etc, all of which have different numerical implementations and characteristics. To allow the use of many sequence formats, we use the `std::ranges` library. This is a generic abstraction of containers defined through concepts, which specifies an object type that supports iteration. This provides the flexibility to allow users to provide their own memory layout or container types without rewriting core algorithms.

When working with multidimensional arrays, flat iteration doesn't suffice. In general, describing multi-rank data generically is not a trivial pursuit. There are many ways to encode a multidimensional object into memory and the number of them increases with the rank. `std::mdspan` plays a critical role. While not as generic as `std::ranges` , it offers a non-owning multi-dimensional view over contiguous memory. A view provides a layout and structure over a chunk of contiguous memory. Through the view we can describe, modify and access data as if it had a multidimensional shape without worrying about the underlying memory. Although `std::mdspan` remains an experimental feature for several standard library implementations, it serves as a practical container placeholder until a fully standardised framework becomes available.

As this is somewhat restrictive, some flexibility is obtained through the use of concepts and we can extend the placeholder to include any structure that supports an `extent()` member function and access to its elements through the `operator[]` by the passing of a parameter pack.

In C++, a function object is any object that can be used with the function-call syntax by overloading the operator(). Unlike free functions, function objects can hold state. This makes them particularly suitable for recursive algorithms. Behaving both like a function and like an object, allows them to be manipulated as objects and encapsulate context. In the present work, we use this tool to wrap recursive traversal over multidimensional data. This separates the control logic (iteration and recursion) from the numerical kernel (the one-dimensional FFT), and provides a modular interface that can be reused or extended without modifying the core algorithm.

2.1 Axis-wise slices: multi-dimensional iterator

To apply 1D-FFTs along every axis we pass 1D slices or ranges of the input object, to the kernel. These slices are built using the concept `std::ranges::subrange` whose constructor takes an iterator as input. We therefore use a multi-dimensional iterator template that fixes all but one axis and traverses along the remaining dimension. The main aim of this iterator is to access, through a view, one-dimensional slices of the input array without copying or moving data in memory. This effectively saves us from redundant memory and computation loads.

To be passed to the subrange constructor, this iterator must be a random-access iterator, that is it must satisfy a list of set requirements to be standard compliant. The requirements for a random-access iterator are defined in the C++ standard (ISO/IEC 2024). In practice, most developers verify conformance against the concept constraint lists available in the C++ standard. The distinguishing feature of this iterator is that it accesses elements through the underlying multidimensional view instead of directly pointing to memory.

2.2 fft axis function object

The looping over axes is implemented as a recursive template. At each level of recursion, an index is either fixed (for all axes except the chosen one) or marked as a placeholder for iteration (on the chosen axis). Once all indices have been fixed, the recursion terminates and a one-dimensional slice is identified. The FFT kernel is then applied to this slice through the mdIterator abstraction. Recursion is necessary here to replace nested loops as, at coding time, the rank of the input array and the number of nested loops are unknown.

The recursive algorithm is wrapped into a templated function object which is charged with the computation of the FFT of all slices aligned in a given direction. This encapsulation makes the recursion reusable and composable: instead of calling the recursive procedure directly, users interact with a single object that manages the traversal internally. It also allows the recursion to be invoked in a uniform way across different axes without rewriting the control flow. This also allows users to write additional recursions independently of the general pattern and the FFT kernel.

At the highest level, the multi-dimensional FFT driver invokes these axis function objects in sequence to complete the multi-dimensional FFT.

2.3 General mdfft wrapper

The mdfft structure is a function object that applies the FFT across all dimensions. It uses parameter pack expansion to instantiate a function object for each axis I of the input object. This turns the multidimensional transform into a composition of axis-directional transforms. Here we use parameter packs over loops to do as much of this process at compile time and explicitly show the compiler the underlying code structure for it to easily optimise the code. Fig. 2 shows how nested function objects are structured according to this scheme. `mdfft()` calls `fft_axis()` for every axis I which itself handles the recursions for each slice.

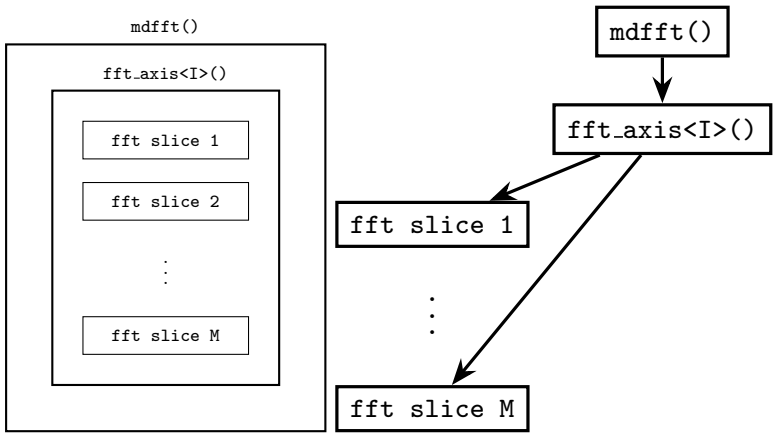

Fig. 2. Left: Structural nesting, showing how function objects and kernels are encapsulated within the driver. **Right:** Call hierarchy, where the top-level driver delegates to axis-wise function objects and eventually to the 1D kernel.

This approach offers several key benefits. First, it achieves genericity, as the implementation works for arrays of arbitrary rank determined at compile-time, without requiring separate code paths for each dimensionality. Second, it promotes code reuse by relying on the same core 1D FFT engine used in simpler, lower-dimensional contexts, ensuring consistency and reducing maintenance overhead. Third, it maintains efficiency by operating directly on mdspan views, which avoids costly reshaping or copying of data. Finally, the design is naturally extensible as the recursive structure and abstraction layers make it easy to incorporate parallel FFT variants.

3 Parallelism

Parallel computing is a key paradigm to accelerating the computation of FFTs. An advantage of the modular structure is that parallelism can be applied at multiple levels.

At the kernel level, one-dimensional FFTs are naturally receptive to parallelisation. In our scheme, each 1D transform operates on a range provided by a view, which may be contiguous or strided in memory. Thread-level parallelism can be applied across elements of the slice regardless of its layout. However, achieving peak SIMD

performance requires contiguous memory: strided slices either prevent vectorization or force the use of slower gather/scatter instructions. Only specific axes in the layout are contiguous depending on the memory pattern used (typically only one dimension of an array is contiguous). Within this framework, thread-level parallelism is most natural given the memory restrictions.

Furthermore, at the slice level, the multidimensional FFT operates on 1D ranges along a fixed axis. Each slice is represented as a view into the input object. Because slices along the same axis are independent, thread-level parallelism can be applied: multiple slices can be processed concurrently, each invoking the 1D FFT kernel. This slice-level parallelism builds naturally on the recursive function object structure, as the recursion enumerates slices along the target axis, providing a modular and efficient mechanism for concurrency. In particular, this structure allows for the concurrent existence of multiple algorithms. Conditional application of these algorithms can be guided by C++ concepts, enabling elaborate parallel processing schemes when appropriate.

Direct parallelism across FFT axes is not possible, because the transform along one axis depends on the complete set of values along that axis. As a result, the next axis cannot begin until all slices of the previous axis have been processed. However, in three or more dimensions, the recursion computes one hyperplane at a time. By making the recursion hyperplane-aware, it is possible to begin computing FFTs along the next axis for portions of a hyperplane that have already been transformed. This approach avoids memory conflicts and respects the mathematical dependencies of the FFT. The procedure can be applied recursively until the final hyperplanes are two-dimensional, enabling finer-grained concurrency within multidimensional transforms.

Finally, because of its recursive slice-based structure, the MDFFT can be applied to sub-arrays of a larger multidimensional arrays with careful memory management. In this case, each sub-array would be transformed independently using the existing MDFFT scheme, and the results would be recombined with appropriate twiddle factors to implement a full multidimensional divide-and-conquer FFT, similar to the Cooley & Tukey (1965) approach. This approach would enable fine-grained parallelism across slices, hyperplanes, or sub-arrays, while respecting data dependencies along each axis.

4 Conclusions

The scheme we have outlined achieves a high degree of generality with minimal code duplication. It allows the FFT logic to scale naturally from one to arbitrary dimensions while remaining the core of the structure. Dimensionality logic and numerical transform kernels are separated and arranged as pluggable, composable components. The structure is naturally compatible with parallelism, supporting parallel computation at almost all levels. It also provides a blueprint for generalising other separable integral transforms (wavelets, Hankel transforms) by replacing the one-dimensional engine while preserving the recursive and iterator-based logic. Similarly, optimisations such as SIMD or other types of parallelism can be integrated at the kernel level without disrupting the high-level abstraction.

This type of strategy suits well the modular and performance needs of astrophysical codes. However, many optimisations related to parallel computing remain to be implemented by hand no matter how well the compiler is guided.

In summary, the proposed scheme demonstrates how modern C++ abstractions, such as concepts, mdspan, function objects, and iterators can enable modular, efficient, and reusable signal processing code. It reflects a broader philosophy: the more generic the interface, the more reusable and future-proof the system.

We thank the organisers of the SF2A 2025 for the opportunity to present this work. This research was carried out as part of a Master's internship at LAPP and financially supported by Université Savoie Mont Blanc.

References

Cooley, J. W. & Tukey, J. W. 1965, Mathematics of Computation, 19, 297

Frigo, M. & Johnson, S. G. 2005, Proceedings of the IEEE, 93, 216

ISO/IEC, ed. 2024, ISO/IEC 14882:2024 – Programming Languages – C++ (Geneva, Switzerland: International Organization for Standardization (ISO)), standard document, latest version of C++ standard

Kumar, G. G., Sahoo, S. K., & Meher, P. K. 2019, Circuits, Systems, and Signal Processing, 38, 5665

Press, W. H. 2007, Numerical recipes 3rd edition: The art of scientific computing (Cambridge university press)

Sugiyama, N. S., Saito, S., Beutler, F., & Seo, H.-J. 2019, Monthly Notices of the Royal Astronomical Society, 484, 364