FLIT: A Generic Fortran Library based on Interfaces and Templates

Kai Gao*,[1] and Ting Chen[1]

[1]Earth and Environmental Sciences Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

1 Summary

We develop a generic Fortran library, `FLIT` (**F**ortran **L**ibrary based on **I**nterfaces and **T**emplates), to provide a number of useful functionalities for computational geophysics and beyond. These functionalities include several single/multi-dimensional array manipulation functions/subroutines, flexible parameter reading from textual file or command line arguments, signal and image filtering and processing, integral transforms, interpolation, statistical functions, and so on. These functionalities are not intrinsically available in the current Fortran standard. The most notable feature of `FLIT` is that we provide user-friendly interfaces for similar functionalities with different data types, attempting to demonstrate the possibility of accelerating scientific application development using modern Fortran. The package can be used as a robust Fortran library for developing sophisticated scientific applications in the fields of computational physics, computational and applied geophysics, signal and image processing, and so on.

2 Statement of need

Fortran has been a programming language of choice for developing scientific and engineering applications because of its simplicity, efficiency, and in particular, its capability in conveniently handling multi-dimensional arrays, ever since its birth in 1957. Although being gradually superseded by newer programming languages like Python, Julia, and modern C++, the modernized Fortran ($>$ Fortran 90) still anchors to a wide variety of computationally intensive applications. However, in many cases, one still need to develop low-level functions or subroutines to perform basic array operations from scratch. Many efforts emerge in the recent decade to provide a so-called standard library for Fortran (e.g., `stdlib` – `https://github.com/fortran-lang/stdlib`), covering array manipulations, string functions, hash table, along with many others, with varying levels of success and community acceptance.

In computational geophysics and many fields alike, one need to routinely manipulate various types of arrays to achieve efficient manipulation, interpolation, filtering, and/or other types of processing of digitized signals and images. Unlike those in the domain of, for instance, Python and MATLAB, many Fortran users still scramble to find a consistent, flexible, and efficient library that encloses these functionalities. One of the most critical issues people recognize for Fortran is its lack of convenient generic programming (e.g., something like C++'s template) to accommodate different types of input variables. There have been several choices for Fortran generic programming (e.g., `fypp` – `https://github.com/aradi/fypp`), with different degrees of maturity and convenience to use.

*Email: kaigao@lanl.gov

Our `FLIT` attempts to build a generic Fortran library by exploiting interfaces and templates for scientific applications, especially for computational physics and geophysics. It encloses a number of modules for array allocation and manipulation, signal and image interpolation, filtering, flexible parameter reading, statistical functions, Fourier transforms, linear algebra, and so on. In particular, we design user-friendly interfaces for these functions and subroutines, so that a user can easily build sophisticated computational functionalities based on the modules. In a departure from `fypp`'s syntax, in `FLIP` we use a more traditional approach based on macros (`#define`-`#undef`) and file inclusion (`#include`) to achieve generic programming for various types of functionality. We emphasize that, by `FLIT`, we are not attempting to replace the current community effort of constructing a Fortran standard library or anything alike; we are not attempting to create a *library-of-everything* either, which is probably too ambitious for modern scientific research with highly diverse needs from different fields.

`FLIT` is designed to be used by both researchers in computational geophysics and researchers in a much broader scope that need to routinely manipulate single/multi-dimensional arrays, use signal and image processing functionalities, interpolation, complex parameter input, and so on. We have used it as a base library to develop sophisticated seismic/electromagnetic wave modeling, traveltime tomography, full-wavefield imaging, and full-waveform inversion applications, as well as to generate high-fidelity synthetic data/images for training machine learning models. These works have generated in a number of publications, including Gao et al. (2022), Gao (2023), and Gao and Modrak (2024), to name a few. The generic functionalities of `FLIT` will likely enable more scientific applications not only in computational geophysics but also in other computational fields alike.

3 Design and methodology

While `FLIT` contains several modules to cover a variety of needs for Fortran users, we highlight the following modules that may be particularly related to computational geophysics applications. We implement most of the functionalities in `FLIT` using OpenMP (for shared-memory parallelism) and/or MPI (message passing interface, for distributed memory parallelism) wherever possible, to improve the computation efficiency and scalability.

3.1 Array manipulation

We design a number of basic array allocation and manipulation functions and subroutines, and provide unified interfaces for each of these functionalities for different data types, attempting to extend the intrinsic function/subroutine library provided by the current Fortran standard.

- Array allocation: The memory allocation function `allocate` in the current Fortran standard cannot be used on an already allocated array before deallocating the array. The restrict can cause lengthy codes. `FLIT` provides a number of array allocation methods for safely dealocation-allocation, e.g.,
  - `alloc_array(w, [lbound1, ubound1, ...], pad=l, source=s)` – a subroutine that encloses both `deallocate` and `allocate` based on specific lower and upper bounds, and also pad the array if needed. When `w` is already allocated, `alloc_array` will first deallocate the array and then allocate, ensuring memory safety;
  - `zeros(n1, n2, ...)` – a function that allocates an array of zeros. Based on Fortran's dynamic allocation mechanism, the resulting array will be automatically adjusted to the new dimensions, if it is already allocated;
  - `ones(n1, n2, ...)` – a function that allocates an array of ones;
  - `const(n1, n2, ...)` – a function that allocates an array of constant;
  - `trues(n1, n2, ...)`, `falses(n1, n2, ...)` – a function that allocates an array of Fortran `.true.` or `.false.`;

  - `zeros_like`, `ones_like`, `const_like`, `trues_like`, `falses_like` – functions that alloate an array like another array (i.e., without needing to specify dimensions);
  - `random(n1, n2, ..., dist, seed)` – a function that allocates an array filled with random values, where `dist` is the distribution of the random distribution, and `seed` is the seed for reproducibility. Without specifying the `seed`, the random values that fill the array will be different every time it is initialized, even in OpenMP or MPI parallel processes that run simultaneously.

  `FLIT` also provides convenient interfaces for generating regularly spaced array:
  - `linspace` – generating a regularly spaced array from a to b with n points, i.e.,

    $$x_i = a + (i-1)\frac{b-a}{n-1}, \quad (i = 1, 2, \cdots, n), \tag{1}$$

    where the dimension of the resulting array is n, with head and tail to be a and b, respectively.
  - `regspace` – generating a regularly spaced array from a with an interval of d, till b, i.e.,

    $$x_i = a + (i-1)d, \quad (i = 1, 2, \cdots, \text{ until } x_i \le b), \tag{2}$$

    where the dimension of the resulting array can be queried through `size(x)`; the head of the resulting array is a, the interval for two consecutive elements is d, and the tail of the resulting array is $\max(x_i) \le b$.

  Similar functions also include `logspace` for generating regularly spaced array on the logarithmic scale, and `meshgrid` for generating single- to multi-dimensional regularly spaced array.
- Array manipulation: `FLIT` provides a number of array manipulations, e.g.,
  - `crop` – cropping an array;
  - `flatten` – flatten a multi-dimensional array to a 1D array;
  - `flip` – flip an array with respect to one or multiple axes;
  - `pad` – pad an array on edges;
  - `permute` – permute an array, with the target order of dimensions specified as an integer, e.g., `order=321`, and `order=2143`. `FLIT` will interpret the integer associated with the parameter `order` digit by digit and as the target order. For instance, `321` means that after permutation, the third dimension will move to the first dimension, the second dimension keeps the same, while the first dimension will move to the third dimension;
  - `rot90` – rotate an array by 90 degrees;
  - `tile` – repeat an array along one or multiple axes;
  - `rescale` – linearly rescale an array to a specific range;
  - `randomize` – randomize an array with random values following some distribution;
  - `taper` – taper an array using different types of window functions. `FLIT` implements several different window functions, including step, linear, Parzen, Welch, sine, power of sine, Hann, Hamming, Blackman, Nuttall, Blackman-Nuttall, Blackman-Harris, Kaiser, and Gauss windows. Definition of these window functions can be found on the Wikipedia (https://en.wikipedia.org/wiki/Window_function).

3.2 Flexible parameter reading

Current Fortran standard provides a namelist-based parameter input method that is inherited from the Fortran 77 standard. However, in addition to the need of pre-defining a namelist in the code, such a parameter reading functionality falls short when a user demands more sophisticated parameter input, e.g., reading multiple values to initialize an array parameter, or reading a value to a parameter under certain conditions.

FLIT allows for parameter reading from three types of sources: from an American Standard Code for Information Interchange (more commonly known as ASCII) file, from command line arguments, and from an array of string. Currently, we provide the functionalities for reading integer (kind 2, 4, and 8), single-precision real numbers, double-precision real numbers, single-precision complex numbers, double-precision complex numbers, strings, and logical values. The suffix #type (as described below) corresponding to these types are int2, int4, int8, float, double, complex, dcomplex, string, and logical.

For ASCII-file-based parameter input, we provide the following interfaces:

- Read a single value to a parameter: call readpar_#type(parameter_file, parameter_name, target_variable, default_value, required), where parameter_file is the ASCII file that holds parameters, parameter_name is the name of the parameter, target_variable is the target variable for storing the input parameter, default_value is the default value of the parameter if it cannot be found in the parameter file, and required is a Fortran logical variable stating that if the parameter is required. For required parameter, if it cannot be found in the parameter file, then the program will give error and stop.
- Read multiple values from file to an array variable: call readpar_n#type (parameter_file, parameter_name, target_variable, default_value, required). The function can read the parameter values in a style like var = 1, 2, 3, 4 for integer or float parameter, var = abcd, efgh, x, y, z for a string parameter, or var = y, n, n, y for a logical parameter.
- Read a value from file to a variable under a condition: call readpar_x#type (parameter_file, parameter_name, target_variable, default_value, condition, required). For instance, for a parameter named par, given a specification in file_parameter like par = 0:0, 10:1, 11∼20:2 (which means when condition = 0, par = 0, condition = 10, par = 1, and when 11 ≤ condition ≤ 20, par = 2, where we use ∼ to represent a range), then call reapar_xfloat (file_parameter, 'par', par, 1, 3.0) (i.e., the condition condition = 3.0) will result par = 0.3, because FLIT will linear interpolate between the known values 0 and 1 to obtain the value for par when condition = 3 (unspecified in the parameter file). For integer, logical, and string parameters where linear interpolation does not apply, FLIT uses nearest interpolation to obtain the value for the parameter under a condition. For instance, for a parameter named par, given a specification in file_parameter like par = 0:0, 10:1, 11∼20:2, then call reapar_xint (file_parameter, 'par', par, 1, 3.0) (i.e., the condition condition = 3.0) will result par = 0, because 3.0 is closer to 0 than to 10. The design of this parameter input method is inspired by Halliburton's Landmark ProMAX (see, for example, https://esd.halliburton.com/support/LSM/GGT/ProMAXSuite/ProMAX/5000/5000_8/Help/promax/user_int.pdf), a famous seismic data processing software that has been superseded by newer products from Halliburton. In seismic processing, one often needs to specify the value of a parameter based on the value or a variable (i.e., the condition here); sometimes the value of another variable is given as a range, similar with the way we design here (i.e., ∼). We believe such a parameter reading method will be a useful feature for developing some computational applications, yet is not available in the current Fortran standard.

Similarly, for reading parameters from command line arguments, FLIT provides three functionalities, getpar_#type, getpar_n#type, and getpar_x#type. In addition, for reading parameters from an array of strings, FLIT provides three functionalities, parsepar_#type, parsepar_n#type, and parsepar_x#type. Syntax of these functions are similar with readpar, with different sources: for getpar, the source is essentially command line arguments, while for parsepar, the source is an array of strings.

It is worthwhile to mention that FLIT allows for flexible complex number parameter reading. For instance, with `FLIT` it is possible to read a complex number to a parameter like `var = 1.0 + 2.0i` or `var = 2i + 1`, both representing a complex number $1 + 2i$. This allows for parameter reading in a more concise and natural manner instead of Fortran's native `(1.0, 2.0)` syntax.

3.3 Image and signal processing

Some of the image and signal filters provided by `FLIT` include

- Gaussian smoothing: Gaussian filtering is one of the most frequently used filters in image processing and in many geophysical applications. We implement several methods of Gaussian filters: the Deriche recursive filter (Deriche, 1990), discrete cosine transform (DCT) (Sugimoto and Kamata, 2013), and kernel convolution based on Fourier transform.
- Laplacian filtering: Filtering out low-wavenumber/frequency components. Details can be found on Wikipedia (https://en.wikipedia.org/wiki/Discrete_Laplace_operator);
- Mean filtering: Smoothing using mean of the windowed array;
- Median filtering: Smoothing using median of the windowed array;
- LOWESS filtering: Locally weighted scatterplot smoothing (LOWESS) is a popular non-parametric tool in regression analysis, but could also be viewed as a filter for smoothing out outliers for a given data with local polynomials. We implement LOWESS filters for 1D, 2D, and 3D data. Details can be found on Wikipedia (https://en.wikipedia.org/wiki/Local_regression);
- Fourier filtering: Filtering using

$$s'(t) = \Re\left[\mathcal{F}^{-1}\left[K(\omega)\mathcal{F}\left[s(t)\right]\right]\right], \tag{3}$$

  where $K(\omega)$ is an arbitrary kernel/window in the wavenumber or frequency domain. Different choices of $K(\omega)$ can result in filters of lowpass, highpass, bandpass, bandstop, etc. Multi-dimensional version of the Fourier filtering is defined and implemented similarly;
- Dip filtering: Filtering based on local dip of a multi-dimensional image. The local dip or slope is computed based on the generalized structure tensor (Bakker, 2002);
- Windowed RMS filtering: Balance the amplitude of a single/multi-dimensional array using windowed root-mean-square (RMS):

$$s_i' = \frac{s_i}{\sqrt{\sum_{j=-w}^{w} s_{i+j}^2 + \epsilon}}, \tag{4}$$

  where w is the half length of the window;
- Nonlinear anisotropic diffusion filtering (AnDF): We implement a conjugate-gradient (CG) based AnDF solver and a fast explicit diffusion (FED) based solver (Weickert et al., 2016) for both 2D and 3D cases, both relying on computing general structure tensors, and both with some improvements over the original algorithms. For FED-based AnDF, we also provide the MPI version to accelerate the computation for 3D images;
- Total variation and total generalized variation filtering: Total variation (TV) filter is an important filter in image processing and more recently in seismic tomography and full-waveform inversion. We implement isotropic TV filtering and total generalized p-variation (TGpV) filtering. For isotropic TV, we follow the algorithm provided by Goldstein and Osher (2009). For TGpV, we implement the algorithms in Knoll et al. (2011). `FLIT` also provides the MPI version of the 3D filters.

In addition, we implement single- and multidimensional convolutions based on both discrete algorithm and Fourier-transform-based algorithm, with a unified interface like

$$s' = \mathrm{conv}(s, K, \mathrm{method}), \tag{5}$$

where K is the discrete kernel specified by the user. When `method = 'same'`, the resulting image s' has the same dimension with s.

3.4 Miscellaneous functionalities

`FLIT` also contains a number of functionalities that cannot be simply categorized in the above three major modules. All these functionalities are not included in the current Fortran standard, including statistical functions, integral transforms (Fourier, Hilbert, DCT, etc.), differentiation operators, geometry (point, line, surface, rotation, etc.), and so on. Users can refer to the source codes for details.

4 Acknowledgments

The work was supported by Los Alamos National Laboratory (LANL) Laboratory Directory Research and Development (LDRD) project 20240322ER. LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration (NNSA) of the U.S. Department of Energy (DOE) under Contract No. 89233218CNA000001. The research used high-performance computing resources provided by LANL's Institutional Computing program.

References

Bakker, P., 2002, Image structure analysis for seismic interpretation: Ph.D. dissertation of Delft University of Technology, doi: citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ab677525a49b902a42e3e3b42e779dcbcb957b1f.

Deriche, R., 1990, Fast algorithms for low-level vision: IEEE Transcations on Pattern Analysis Machine Intelligence, **12**, no. 1, 78–87, doi: 10.1109/34.41386.

Gao, K., 2023, Iterative multitask learning and inference from seismic images: Geophysical Journal International, **236**, no. 1, 565–592, doi: 10.1093/gji/ggad424.

Gao, K., C. Donahue, B. G. Henderson, and R. T. Modrak, 2022, Deep-learning-guided high-resolution subsurface reflectivity imaging with application to ground-penetrating radar data: Geophysical Journal International, **233**, no. 1, 448–471, doi: 10.1093/gji/ggac468.

Gao, K., and R. T. Modrak, 2024, Machine learning inference of random medium properties: IEEE Transactions on Geoscience and Remote Sensing, **62**, 1–13, doi: 10.1109/TGRS.2024.3367541.

Goldstein, T., and S. Osher, 2009, The split Bregman method for L1-regularized problems: SIAM Journal on Imaging Sciences, **2**, no. 2, 323–343, doi: 10.1137/080725891.

Knoll, F., K. Bredies, T. Pock, and R. Stollberger, 2011, Second order total generalized variation (TGV) for MRI: Magnetic Resonance in Medicine, **65**, no. 2, 480–491, doi: 10.1002/mrm.22595.

Sugimoto, K., and S.-i. Kamata, 2013, Fast Gaussian filter with second-order shift property of DCT-5: 2013 IEEE International Conference on Image Processing, 514–518, doi: 10.1109/ICIP.2013.6738106.

Weickert, J., S. Grewenig, C. Schroers, and A. Bruhn, 2016, Cyclic schemes for PDE-based image analysis: International Journal of Computer Vision, **118**, no. 3, 275–299, doi: 10.1007/s11263-015-0874-1.