Fortran Interoperability with Python and C

Daniel Topa
daniel.topa@hii.com

Huntington Ingalls Industries
Mission Technologies

December 13, 2024

Abstract

Modern Fortran is a powerful language for high-performance computing (HPC) and numerical analysis, offering features specifically designed for scientific and engineering applications. Its support for advanced numerics, such as high-precision arithmetic and robust intrinsic functions, makes it ideal for solving complex mathematical problems. With built-in tools for vectorization and array operations, Fortran enables concise, efficient code for handling large datasets. Coarrays introduce a simple yet effective parallel programming model, allowing developers to harness distributed-memory architectures with ease. Additionally, modern Fortran integrates seamlessly with C and Python, ensuring interoperability and access to extensive ecosystems. These features, combined with decades of optimization for numerical workloads, which makes modern Fortran a top choice.

Contents

1 About Fortran

Modern Fortran is an ideal choice for computationally intensive applications due to its efficient handling of arrays in numerical computations, vector formulation, advanced parallelism through coarrays, and seamless integration with C and other languages. Its object-oriented features enable large-scale software design, while array manipulation and intrinsic functions provide concise and readable code for mathematical and scientific applications. Fortran is a robust, high-performance language for HPC, data analysis, and numerical modeling in the modern era.

1.1 Fortran: A Pioneering Language

The advent of commodity-priced personal computers and the democratization of the Internet. At times, Fortran has spurred new paradigms, at other times responded. The watershed change was captured in the Fortran 90 standard. IBM, once guardian of the language, saw their influence wane as the corporation stumbled in the personal computer market. They fought to maintain the simplicity of the language and lost to those who wanted to include object oriented programming features, setting a new mindset over the language. Even more revolutionary, was the remake of Fortran into a vector language. While Cray had supplied Fortran compilers with vector tools for Cray super computers, the new standard brought vector computing to the desktop, an astonishing and often overlooked breakthrough.

- **Fortran 90:**
  - Modular programming with `MODULE`.
  - Dynamic memory allocation and allocatable arrays.
  - Whole-array and elemental operations for numerical computations.
  - User-defined types and array slicing.
  - Enhanced control structures like `DO WHILE` and `SELECT CASE`.
- **Fortran 95:**
  - `FORALL` and `WHERE` constructs for parallelism.
  - `PURE` and `ELEMENTAL` procedures for functional programming.
- **Fortran 2003:**
  - Full object-oriented programming with type extension, polymorphism, and type-bound procedures.
  - C interoperability with the `ISO_C_BINDING` module and `BIND(C)` attribute.
  - Asynchronous and stream I/O for better data handling.
- **Fortran 2008:**[1]
  - Introduction of coarrays for native parallel programming.
  - `SUBMODULES` for modular program decomposition.
  - `DO CONCURRENT` for loop-level parallelism.
  - `BLOCK` construct for nested variable scoping.
- **Fortran 2018:**
  - Enhanced coarrays with teams and events for parallel synchronization.
  - Support for optional arguments and assumed-rank arrays in C interoperability.
  - `FAIL IMAGE` and error codes in `STOP` statements for debugging.
  - Bitwise operations and improved intrinsic functions.
- **Fortran 2023 (Upcoming):**
  - Improved object-oriented features, including finalization and procedure pointers.
  - Automatic deallocation of allocatable arrays.
  - Further enhancements to coarray parallelism and task-based concurrency.
  - Simplified syntax and improved performance diagnostics.

[1] The infamous DO LOOP is now extinct.

1.2 Why Fortran?

Perhaps a better question is Why aren't you using a vector language with intrinsic parallelism? We should rise beyond the cultural inertia of scalar and serial programming.

2 Calling Fortran from Python

Integrating Fortran with Python combines the computational efficiency of Fortran with the flexibility and ecosystem of Python, an approach particularly valuable in HPC and scientific workflows where computational efficiency and ease of scripting are paramount (See e.g. [2, 14, 1, 11]). This section demonstrates a practical approach to achieve this using PyBind11, highlighting examples and best practices.

2.1 Why PyBind11?

PyBind11 is a high-level library for binding C++ code to Python. Although designed for C++, it serves as an effective bridge between Python and Fortran by leveraging C++ wrappers for Fortran routines. Benefits include:

- Minimal overhead, allowing efficient execution of Fortran routines.
- High-level abstractions to simplify binding.
- Direct integration with Python's ecosystem.

2.2 Workflow Overview

The workflow to call Fortran from Python using PyBind11 involves three main steps:

1. Writing computational routines in Fortran and compiling them into a shared library.
2. Creating a C++ wrapper to expose the Fortran routines.
3. Using PyBind11 to bind the C++ wrapper to Python.

2.3 Example Implementation

2.3.1 Fortran Code

Here is a simple Fortran module that adds two arrays:

```
module myfortran
contains
    subroutine add_arrays(a, b, result, n)
        real, intent(in) :: a(:), b(:)
        real, intent(out) :: result(:)
        integer, intent(in) :: n
        integer :: i

        result = a + b

    end subroutine add_arrays
end module myfortran
```

Listing 1: Fortran routine to add arrays.

Compile the Fortran code into a shared library:

Listing 2: Fortran compilation into a shared object.

```
$ gfortran -shared -fPIC -o libmyfortran.so myfortran.f90
```

2.3.2 C++ Wrapper

Create a C++ wrapper to call the Fortran routine. Use the `extern "C"` directive to ensure compatibility:

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

extern "C" {
    void add_arrays_(float* a, float* b, float* result, int* n);
}

namespace py = pybind11;

void add_arrays(py::array_t<float> a, py::array_t<float> b, py::array_t<float> result) {
    auto buf_a = a.request();
    auto buf_b = b.request();
    auto buf_result = result.request();

    if (buf_a.size != buf_b.size || buf_a.size != buf_result.size) {
        throw std::runtime_error("Array sizes must match.");
    }

    int n = buf_a.size;
    add_arrays_(static_cast<float*>(buf_a.ptr), static_cast<float*>(buf_b.ptr),
                static_cast<float*>(buf_result.ptr), &n);
}

PYBIND11_MODULE(myfortran, m) {
    m.doc() = "Python interface for Fortran routines";
    m.def("add_arrays", &add_arrays, "Add two arrays using Fortran");
}
```

Listing 3: C++ wrapper for the Fortran routine.

Compile the C++ wrapper with PyBind11:

Listing 4: Compiling the PyBind11 wrapper.

```
c++ -O3 -Wall -shared -std=c++17 -fPIC $(python3 -m pybind11 --includes) \
    wrapper.cpp -o myfortran$(python3-config --extension-suffix) -L. -lmyfortran
```

2.3.3 Python Code

Finally, use the Fortran routine from Python:

```python
import numpy as np
import myfortran

a = np.array([1.0, 2.0, 3.0], dtype=np.float32)
b = np.array([4.0, 5.0, 6.0], dtype=np.float32)
result = np.empty_like(a)

myfortran.add_arrays(a, b, result)
print("Result:", result)
```

Listing 5: Calling the Fortran routine in Python.

2.4 Alternatives to PyBind11

While PyBind11 is versatile, other options exist for calling Fortran from Python:

- **CFFI** C Foreign Function Interface: Simplie, flexibile tools to call C functions and write inline C code in Python [4, 3].
- **Chasm:** Lightweight tool for creating bindings between Python and Fortran, emphasizing simplicity and ease of use.
- **ctypes:** Built-in Python library for calling functions in shared C libraries (.so – Linux, .dylib – macOS, DLLs – Windows,).
- **Cython:** Provides fine-grained control over bindings but involves more boilerplate code than PyBind11 [5, 8, 6, 7].
- **pyfort:** Handles boilerplate code to bridge the Python C API and Fortran, but limited compatibility (Some Fortran 90) [12].
- **F2PY:** A Fortran-to-Python interface generator that is part of NumPy. Ideal for pure Fortran-Python workflows, [9, 10].

Feature	Best For	URL
ctypes	Small projects, quick integrations	`https://docs.python.org/3/library/ctypes.html`
CFFI	Complex C libraries	`https://cffi.readthedocs.io/`
F2PY	Interfacing Fortran	`https://numpy.org/doc/stable/f2py/`
Cython	High-performance Python-C	`https://cython.org/`
Pyfort	Legacy Fortran libraries	`https://pyfortran.sourceforge.net/pyfort/`
Chasm	Simpler bindings	`https://example.com/chasm`

Table 1: Condensed Comparison of Python-C/Fortran Integration Tools (*Created by Achates, ChatGPT 4o).*

3 Fortran Interoperability With C

The interactions between Fortran and C are an area of tremendous academic and practical interest, e.g. [13, 15, 1]. A summary of Fortran language features follows.

3.1 Fortran and C Interoperability

Fortran 2018 introduced significant advancements in interoperability with the C programming language, marking a major breakthrough in combining the strengths of these two languages. This section highlights the key features of Fortran-C interoperability as defined in the Fortran 2018 standard.

3.2 Key Features of Interoperability

Fortran 2018 provides robust mechanisms to facilitate seamless integration between Fortran and C, ensuring efficient and consistent data exchange and procedure calling conventions. The following

are the main features of Fortran-C interoperability:

3.2.1 Interoperable Data Types

The standard defines a set of data types that are compatible between Fortran and C. These data types ensure seamless translation and interpretation of data structures when shared across both languages.

3.2.2 The `ISO_C_BINDING` Module

The `ISO_C_BINDING` intrinsic module introduces named constants and derived types that map Fortran types to their C counterparts. This module ensures consistent interpretation and compatibility of data types across both languages.

3.2.3 Procedure Interoperability

Fortran procedures can be made accessible to C, and vice versa, by using the `BIND(C)` attribute. This attribute specifies the linkage convention and optionally the external name to be used in the C environment, enabling seamless procedure calls between the languages.

3.2.4 Interoperability of Global Data

Global variables can be shared between Fortran and C by applying the `BIND(C)` attribute. This allows both languages to access and modify the same global data structures, ensuring consistency across the codebase.

3.2.5 Interfacing with C Pointers

The `ISO_C_BINDING` module includes derived types such as `C_PTR` and `C_FUNPTR`, which facilitate the interaction with C pointers and function pointers. These types ensure that pointer operations remain compatible and error-free.

3.2.6 Interoperability of Arrays

Guidelines are provided in the Fortran 2018 standard for handling array descriptors. These guidelines ensure that arrays, whether passed by Fortran or C, are correctly interpreted and manipulated across the languages.

3.3 Historical Impact

The introduction of these features in Fortran 2018 marked a significant milestone in scientific and engineering computing. By enabling seamless interoperability with C, Fortran retained its dominance in numerical computing while leveraging the extensive ecosystem of C libraries and tools. This advancement greatly simplified the process of integrating Fortran with modern software stacks.

3.4 Further Enhancements with TS 29113

The Technical Specification TS 29113, titled *Further Interoperability of Fortran with C*, builds upon the Fortran 2018 standard to offer extended capabilities and clarifications. Notable additions include support for assumed-shape arrays and optional dummy arguments in interoperable procedures. For more details, refer to `https://j3-fortran.org/doc/year/12/12-119.pdf`.

Fortran and C interoperability continues to empower developers to combine the computational efficiency of Fortran with the versatility of C, enabling powerful solutions for modern scientific and engineering challenges.

A Fortran's `iso_c_binding`

Fortran 2018 introduced significant advancements in interoperability with the C programming language, marking a major breakthrough in combining the strengths of these two languages. This section highlights the key features of Fortran-C interoperability as defined in the Fortran 2018 standard the following is excerpted and adapted from *Modern Fortran Explained: Incorporating Fortran 2023, 3rd Edition* by Metcalf, Reid, Cohen, and Bader (2024).

A.1 Key Features of Interoperability

Fortran 2018 provides robust mechanisms to facilitate seamless integration between Fortran and C, ensuring efficient and consistent data exchange and procedure calling conventions. The following are the main features of Fortran-C interoperability:

A.1.1 Interoperable Data Types

The standard defines a set of data types that are compatible between Fortran and C. These data types ensure seamless translation and interpretation of data structures when shared across both languages.

A.1.2 The `ISO_C_BINDING` Module

The `ISO_C_BINDING` intrinsic module introduces named constants and derived types that map Fortran types to their C counterparts. This module ensures consistent interpretation and compatibility of data types across both languages.

A.1.3 Procedure Interoperability

Fortran procedures can be made accessible to C, and vice versa, by using the `BIND(C)` attribute. This attribute specifies the linkage convention and optionally the external name to be used in the C environment, enabling seamless procedure calls between the languages.

A.1.4 Interoperability of Global Data

Global variables can be shared between Fortran and C by applying the `BIND(C)` attribute. This allows both languages to access and modify the same global data structures, ensuring consistency across the codebase.

Type	Named Constant	C Type or Types
integer	c_int	int
	c_short	short int
	c_long	long int
	c_long_long	long long int
	c_signed_char	signed char, unsigned char
	c_size_t	size_t
	c_int8_t	int8_t
	c_int16_t	int16_t
	c_int32_t	int32_t
	c_int64_t	int64_t
	c_int_least8_t	int_least8_t
	c_int_least16_t	int_least16_t
	c_int_least32_t	int_least32_t
	c_int_least64_t	int_least64_t
	c_int_fast8_t	int_fast8_t
	c_int_fast16_t	int_fast16_t
	c_int_fast32_t	int_fast32_t
	c_int_fast64_t	int_fast64_t
	c_intmax_t	intmax_t
	c_intptr_t	intptr_t
real	c_float	float
	c_double	double
	c_long_double	long double
complex	c_float_complex	float _Complex
	c_double_complex	double _Complex
	c_long_double_complex	long double _Complex
logical	c_bool	_Bool
character	c_char	char

Table 2: Named constants for interoperable kinds of intrinsic Fortran types. Adapted from Metcalf, Cohen, and Reid, Table 19.1.

A.1.5 Interfacing with C Pointers

The `ISO_C_BINDING` module includes derived types such as `C_PTR` and `C_FUNPTR`, which facilitate the interaction with C pointers and function pointers. These types ensure that pointer operations remain compatible and error-free.

A.1.6 Interoperability of Arrays

Guidelines are provided in the Fortran 2018 standard for handling array descriptors. These guidelines ensure that arrays, whether passed by Fortran or C, are correctly interpreted and manipulated across the languages.

A.2 Historical Impact

The introduction of these features in Fortran 2018 marked a significant milestone in scientific and engineering computing. By enabling seamless interoperability with C, Fortran retained its dominance in numerical computing while leveraging the extensive ecosystem of C libraries and tools. This advancement greatly simplified the process of integrating Fortran with modern software stacks.

A.3 Further Enhancements with TS 29113

The Technical Specification TS 29113, titled *Further Interoperability of Fortran with C*, builds upon the Fortran 2018 standard to offer extended capabilities and clarifications. Notable additions include support for assumed-shape arrays and optional dummy arguments in interoperable procedures. For more details, refer to `https://j3-fortran.org/doc/year/12/12-119.pdf` [.]

A.4 Additional Concepts from Chapter 20

The following are additional interoperability concepts excerpted and adapted from *Modern Fortran Explained, 5th Edition* by Metcalf, Reid, Cohen, and Bader (2024):

A.4.1 Interoperability of Derived Types

Derived types can be made interoperable with C structures by using the `BIND(C)` attribute. For example, a Fortran type can interoperate with a C structure if they have the same number of components, and each component is interoperable. Components must be declared in the same order.

A.4.2 Procedure Interoperability

Fortran procedures can interoperate with C function prototypes if they declare arguments and results that match the corresponding C types. A Fortran explicit-shape or assumed-shape array can interoperate with a C array.

A.4.3 Assumed-Type Dummy Arguments

The assumed-type syntax `TYPE(*)` allows interoperability with C types where no type information is specified. Assumed-type dummy arguments can be scalar, assumed-shape, or rank-agnostic arrays.

A.4.4 Optional Arguments

Fortran allows optional arguments to interoperate with C. If an argument is not present, Fortran passes a null pointer to the C procedure, enabling flexible interoperability with optional parameters in C.

A.4.5 Enumerations

Enumerations declared with `BIND(C)` enable Fortran to interoperate with C `enum` types. For example, a C enumeration can map directly to a Fortran kind value.

A.5 Final Notes

Fortran and C interoperability takes a "best of both worlds" approach by combining the computational strengths of Fortran with the extensive library ecosystem of C, enabling robust solutions for scientific and engineering challenges.

References

1 Sylwester Arabas et al. “Formula translation in Blitz++, NumPy and modern Fortran: A case study of the language choice tradeoffs”. In: *Scientific Programming* 22.3 (2014), pp. 201–222.

2 PyHPC Workshop Committee. *PyHPC 2016 Slides Handout*. Presentation slides. Slides from the PyHPC 2016 workshop, focusing on Python for High-Performance Computing. 2016. URL: https://mbdevpl.github.io/pyhpc2016_slides_handout.pdf (visited on 12/12/2024).

3 CFFI Developers. *CFFI Documentation*. Accessed: 2024-12-11. 2024. URL: https://cffi.readthedocs.io/en/latest/.

4 CFFI Developers. *CFFI GitHub Repository*. Accessed: 2024-12-11. 2024. URL: https://github.com/cffi/cffi.

5 Cython Developers. *Cython Documentation*. Accessed: 2024-12-11. 2024. URL: https://docs.cython.org/en/latest/.

6 Cython Developers. *Cython Official Website*. Accessed: 2024-12-11. 2024. URL: https://cython.org/.

7 Cython Developers. *Cython Tutorial*. Accessed: 2024-12-11. 2024. URL: https://docs.cython.org/src/tutorial/cython_tutorial.html.

8 Cython Developers. *Cython: Getting Started Guide*. Accessed: 2024-12-11. 2024. URL: https://docs.cython.org/en/latest/src/quickstart/index.html.

9 NumPy Developers. *F2PY Documentation (Stable)*. Accessed: 2024-12-11. 2024. URL: https://numpy.org/doc/stable/f2py/.

10 NumPy Developers. *F2PY Usage Guide*. Accessed: 2024-12-11. 2024. URL: https://numpy.org/doc/2.1/f2py/usage.html.

11 P.F. Dubois and T.-Y. Yang. “Extending Python with Fortran”. In: *Computing in Science & Engineering* 1.5 (1999), pp. 66–73. DOI: 10.1109/5992.790589. URL: https://ieeexplore.ieee.org/document/790589.

12 Paul F. Dubois. *PyFort Reference Manual*. Comprehensive guide for PyFort functionality and usage. n.d. URL: https://pyfortran.sourceforge.net/pyfort/pyfort_reference.htm (visited on 12/12/2024).

13 Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. “Automated Fortran–C++ Bindings for Large-Scale Scientific Applications”. In: *Computing in Science & Engineering* 22.5 (2020), pp. 84–94. DOI: 10.1109/MCSE.2019.2924204.

14 Hans Petter Langtangen. “Fortran Programming with Numerical Python Arrays”. In: *Python Scripting for Computational Science* (2006), pp. 443–473.

15 Alexander Pletzer et al. “Exposing Fortran Derived Types to C and Other Languages”. In: *Computing in Science & Engineering* 10.4 (2008), pp. 86–92. DOI: 10.1109/MCSE.2008.94.