The Journal of Open Source Software

DL_PY2F: A library for Python-Fortran interoperability

You Lu [1¶] **and Thomas W. Keal** [1]

1 STFC Scientific Computing, Daresbury Laboratory, United Kingdom ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Rohit Goswami

Reviewers:

- @awvwgk
- @mgoonde

Submitted: 28 August 2025
Published: unpublished

Summary

Fortran is long established as one of the major programming languages for scientific software programs, but has only limited facilities for interoperating with other modern languages such as Python. `DL_PY2F` is an open-source library for the creation of modern interfaces and data structures in Python that can interoperate with existing scientific software written in Fortran and manipulate their data.

Statement of need

`DL_PY2F` was created to facilitate the redevelopment of the computational chemistry environment `ChemShell` (The authors of ChemShell, 2025), which contains a number of Fortran modules and interfaces to external Fortran software. The redevelopment involved a new Python-based user interface, a modern Python software architecture, and NumPy-based core data structures. A key criterion for the redeveloped `Py-ChemShell` package (Lu et al., 2018, 2023) was ensuring the direct accessibility of its core Python data structures in the Fortran modules and interfaces. `DL_PY2F` achieves this interoperability using the Python `ctypes` and `numpy.ctypeslib` libraries and relevant features in the Fortran 2003 standard, in particular `iso_c_binding`. In `Py-ChemShell`, class instances and `NumPy` array data such as molecular coordinates, energy gradients, and Hessian matrices, are mapped to Fortran pointers via memory addresses by `DL_PY2F`. A simple Fortran 2003 interface is all that is required to access the data. As a result, developers benefit from a seamless Python user interface experience with native data objects, while direct access to data is possible for numerically intensive computing tasks in the Fortran code.

While the `DL_PY2F` library is vital for the `Py-ChemShell` program, it has the potential to be applied in other situations in which Fortran code could benefit from integration into a wider Python software environment. Therefore, we have released `DL_PY2F` as an independent, general-purpose library for Python–Fortran interoperability.

Python-to-Fortran interoperability

`DL_PY2F` is intended for use with Python-based software packages where data are managed using Python/`NumPy` and need to be accessed for computations performed by Fortran code. At the ABI level, a pre-compiled shared object (dynamic library) containing a Fortran 2003 interface to the existing Fortran application is loaded using Python's `ctypes.CDLL`, as follows:

```
import ctypes, dl_py2f
libapp = ctypes.CDLL('/abc/def/libapp.so')
ierror = libapp.interface_app(dl_py2f.py2f(appObj))
```

Here, `appObj` is an instance, typically created by the user at runtime, of the new package's Python class `App` which inherits from `ctypes.Structure`, for example:

```
import ctypes, numpy
from . import callback
class App(ctypes.Structure):
    _kwargs = {
        'callback':callback.callback,
        'child'   :Child(),
        'coords'  :numpy.zeros(shape=(1000,3), dtype=numpy.float64),
        'npoints' :1000}
```

The instance's member attributes are declared in a Python dictionary `App._kwargs` (with default values) and these become visible to the Fortran application after `appObj` is read by the `dl_py2f.py2f` method provided at the API level. DL_PY2F supports a wide range of Python data types, with some illustrative examples given in the example above. Supported data types include 1- and 2-dimensional arrays (`numpy.ndarray`, `numpy.recarray`) and scalar values such as `int`, `float`, and `str` that are commonly needed in scientific computing. `dl_py2f.py2f` works recursively, so that `appObj` may contain unlimited levels of child instances (e.g., `appObj.child` in the above example). DL_PY2F provides utility tools to facilitate initialisation and enhancement of an instance. Please see the example application provided in the `DL_PY2F-example` repository for further details. Callback functions to facilitate two-way data communication are also supported by DL_PY2F.

On the Fortran side, the `interface_app` function receives the passed-in `appPtr` – a pointer to the Python object – and bookkeeps it in a `dictType` instance `PyApp`, which is a linked list with support for child instances (e.g., `PyChild` in the code example):

```
module AppModule
    use iso_c_binding
    use DL_PY2F, only: PyType, ptr2dict
    abstract interface
        integer(c_long) function callback() bind(c)
            use iso_c_binding
        endfunction callback
    endinterface
    type(dictType)     , pointer, public :: PyApp
    procedure(callback), pointer, public :: PyCallback
    contains
    function interface_app(appPtr) bind(c) result(ireturn)
        implicit none
        type(PyType)           , intent(in) :: appPtr
        type(c_funptr)                      :: pyfuncPtr
        type(PyType)   , pointer            :: childPtr
        real(kind=8)   , pointer            :: coords(:,:)
        integer                             :: npoints
        allocate(PyApp, source=ptr2dict(appPtr))
        call PyApp%get('child', childPtr)
        allocate(PyChild, source=ptr2dict(childPtr))
        allocate(coords(3,npoints))
        call PyApp%get('coords', coords)
        call PyApp%get('callback', pyfuncPtr)
        call c_f_procpointer(pyfuncPtr, PyCallback)
        ! run the application
        call my_app(npoints, coords)
        call PyApp%set('coords', coords)
        deallocate(PyApp, PyChild, coords)
    endfunction interface_app
endmodule AppModule
```

A great advantage of DL_PY2F for the application developers is that the attributes of the Python instance are conveniently retrieved by querying their names in a dictionary-like way. Two type-bound procedures get and set grant read and write access, respectively. For arrays, no copies are made because both operations act through memory addresses and values are thus changed in place and reflected on the Python side (if mutable). Callback functions can also be invoked, as follows:

```
subroutine get_something(coords, npoints)
    use AppModule, only: PyApp, PyCallback
    real(kind=8)          , intent(in) :: coords(3,npoints)
    real(kind=8), pointer              :: buffer(:,:)
    call PyCallback()
    allocate(buffer(3,npoints))
    call PyApp%get('coords', buffer)
    coords = buffer
    deallocate(buffer)
endsubroutine
```

DL_PY2F's Python-to-Fortran interoperability has been comprehensively tested using both GNU and Intel compilers.

Fortran-to-Python interoperability

While the Python-to-Fortran interoperability described above is recommended for new or redeveloped Python projects, other applications may benefit from interoperability using Python wrappers around their existing Fortran codes. For this DL_PY2F offers an ABI-style second method based on analysis of the symbols in a pre-compiled shared object and parsing of the Fortran module files, which are assumed to be kept at compiletime. In this method Python's dot syntax may be used to access Fortran entities, for example, in a Python function invoked by the application at runtime:

```
import dl_py2f
libapp = dl_py2f.DL_DL('/abc/def/libapp.so')
libapp.moddir = '/abc/def/modules'
libapp.modules.my_mod.b.coords[1,2]  = 1.2345
libapp.modules.my_mod.b.a[2,:].ibuff = 2025
```

given that the original application's Fortran code contains:

```
module my_mod
    type type_a
        integer :: ibuff
    endtype type_a
    type type_b
        type(type_a)                   :: a(5,6)
        real(kind=8)     , allocatable :: coords(:,:)
    endtype type_b
    type(type_b) :: b
endmodule my_mod
```

All Python attributes, including arrays of numbers and derived-type instances, are automatically instantiated as soon as an instance of class dl_py2f.DL_DL is created and a path to the module files is specified. The seamless access to Fortran data empowered by DL_PY2F will be particularly useful for machine-learning enhanced scientific computing, and is currently being trialled with the established computational chemistry codes DL-FIND (Kästner et al., 2009) and DL_POLY (Devereux et al., 2025). Note that this second method for Fortran-to-Python interoperability in DL_PY2F is still undergoing testing and validation, and is currently limited to use with the GNU compiler gfortran, as the proprietary .mod file format used by the Intel compiler (Green, 2024) is not yet supported.

Comparison with other tools

A number of tools have been developed to facilitate coupling of Python and Fortran code. A major category of these are interface generators, such as `F2PY` (Peterson, 2009) and its extensions, e.g. `f90wrap` (Kermode, 2020) and `Scikit-build-core`. These tools serve as builders which process Fortran source files and write out Python extension modules (via an intermediate C layer), and they put stress on calling specific Fortran functions/subroutines from Python. Such use cases often demand editing the original Fortran code, which, however, could be inconvenient or even unfeasible. By comparison, `DL_PY2F` is intended for invoking a whole library via a call to the Fortran application's main routine, and is designed to only need a minimal interace and not require modifications to the original Fortran source code. `DL_PY2F`' also supports more of the Fortran language: it is not limited to Fortran standards beyond 95 and supports Fortran derived types. `gfort2py` is an ABI tool that works similarly to the Fortran-to-Python interoperability in `DL_PY2F`, also without modification to the source code, while restricted to use of the `gfortran` compiler. In contrast to both `F2PY` and `gfort2py`, `DL_PY2F` does not provide compile tools which are only suitable for small pieces of Fortran code. Furthermore, `DL_PY2F` does not intervene in the Fortran applications' procedures, which normally involve a few parameters and are not compiled as exported symbols in shared objects. Instead, we focus on enabling modifications of application behaviour by manipulating computation data through inserting Python methods. An alternative route to realising Python-to-Fortran data binding would be to manually implement the mechanism based on the Python `ctypes` and `NumPy`'s `ctypeslib` modules. Such a challenging task might be indirectly assisted by tools such as `CFFI` which calls a Fortran-bound C code/library at the ABI/API level or `SWIG+Fortran` which generates Fortran 2003 wrappers to existing C/C++ libraries that are then used by Python. `DL_PY2F` provides a more convenient solution by automating this complex mechanism, and is portable across modern Fortran compilers.

Obtaining `DL_PY2F`

`DL_PY2F` is an open-source library released under GNU Lesser General Public License v3.0. It is available for download from the repository. There is also a comprehensive example demonstrating how to use `DL_PY2F` in an application project. `DL_PY2F` has also been published and deployed in a launchpad.net PPA and can be installed as a system package for Debian-type systems.

Acknowledgements

The `DL_PY2F` library was created during the redevelopment of ChemShell as a Python-based package, which was funded by EPSRC under the grant EP/K038419/1. Ongoing support for the development of `DL_PY2F` as part of ChemShell is provided under EPSRC grants EP/R001847/1 and EP/W014378/1, and the Computational Science Centre for Research Communities (CoSeC), via the support provided to the Materials Chemistry Consortium. We acknowledge helpful discussions and suggestions for improvement from Paul Sherwood, Joseph Thacker, Thomas Durrant, and Maitrayee Singh.

References

Devereux, H. L., Cockrell, C., Elena, A. M., Bush, I., Chalk, A. B. G., Madge, J., Scivetti, I., Wilkins, J. S., Todorov, I. T., Smith, W., & Trachenko, K. (2025). DL_POLY 5:

Calculation of system properties on the fly for very large systems via massive parallelism. *arXiv Preprint arXiv:2503.07526*. https://doi.org/10.48550/arXiv.2503.07526

Google Scholar. (2025). *Papers citing Py-ChemShell*. https://scholar.google.com/scholar?cites=7067225450638589990

Green, R. (2024). *Intel® fortran compiler module .mod files version compatibility*. https://community.intel.com/t5/Blogs/Tech-Innovation/Tools/Intel-Fortran-Compiler-Module-mod-Files-Ve post/1600674

Kästner, J., Carr, J. M., Keal, T. W., Thiel, W., Wander, A., & Sherwood, P. (2009). DL-FIND: An open-source geometry optimizer for atomistic simulations. *The Journal of Physical Chemistry A*, *113*(43), 11856–11865. https://doi.org/10.1021/jp9028968

Kermode, J. R. (2020). f90wrap: An automated tool for constructing deep python interfaces to modern fortran codes. *J. Phys. Condens. Matter*. https://doi.org/10.1088/1361-648X/ab82d2

Lu, Y., Farrow, M. R., Fayon, P., Logsdail, A. J., Sokol, A. A., Catlow, C. R. A., Sherwood, P., & Keal, T. W. (2018). Open-source, Python-based redevelopment of the ChemShell multiscale QM/MM environment. *Journal of Chemical Theory and Computation*, *15*(2), 1317–1328. https://doi.org/10.1021/acs.jctc.8b01036

Lu, Y., Sen, K., Yong, C., Gunn, D. S. D., Purton, J. A., Guan, J., Desmoutier, A., Nasir, J. A., Zhang, X., Zhu, L., Hou, Q., Jackson-Masters, J., Watts, S., Hanson, R., Thomas, H. N., Jayawardena, O., Logsdail, A. J., Woodley, S. M., Senn, H. M., … Keal, T. W. (2023). Multiscale QM/MM modelling of catalytic systems with ChemShell. *Physical Chemistry Chemical Physics*, *25*(33), 21816–21835. https://doi.org/10.1039/D3CP00648D

Peterson, P. (2009). F2PY: A tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, *4*(4), 296–305.

The authors of ChemShell. (2025). *ChemShell: Multiscale computational chemistry*. https://chemshell.org