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: 10.21105/joss.08992

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @mgoonde
- @jameskermode

Submitted: 28 August 2025
Published: 21 July 2026

Summary

Fortran is long established as one of the major programming languages for scientific software, 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 application's 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: dictType, PyType, ptr2dict
    abstract interface
        integer(c_long) function callback() bind(c)
            use iso_c_binding
        endfunction callback
    endinterface
    type(dictType)     , pointer, public :: PyApp, PyChild
    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(c_long)                     :: ireturn
        allocate(PyApp, source=ptr2dict(appPtr))
        call PyApp%get('child', childPtr)
        allocate(PyChild, source=ptr2dict(childPtr))
        call PyApp%get('callback', pyfuncPtr)
        call c_f_procpointer(pyfuncPtr, PyCallback)
        call PyApp%get('coords', coords) ! writable handle to the NumPy array
        call my_app(coords)              ! run the application
        call PyApp%finalise()
        call PyChild%finalise()
        deallocate(PyApp, PyChild)
        nullify(coords)                  ! do not deallocate: Python owns the data
        ireturn = 0
    endfunction interface_app
```

```
endmodule AppModule
```

A great advantage of DL_PY2F for application developers is that the attributes of the Python instance are conveniently retrieved by querying their names in a dictionary-like way. The type-bound accessor get moulds a handle of the target Python object with read and write (if mutable in Python) access. For arrays, no copies are made because access is via memory addresses; values are thus changed in place and reflected on the Python side. We also provide a safe mode in which the developer must make a copy first and explicitly use a set function to change the Python values:

```
integer               :: npoints
real(kind=8), pointer :: buffer(:,:)
call PyApp%get('npoints', npoints)
allocate(buffer(3,npoints))
call PyApp%get('coords', buffer, readonly=.true.) ! a copy without write access
call do_something(buffer)
call PyApp%set('coords', buffer)
deallocate(buffer)
```

Callback functions can also be invoked, as follows:

```
subroutine get_something()
    use AppModule, only: PyCallback
    call PyCallback()
endsubroutine
```

DL_PY2F's Python-to-Fortran interoperability has been comprehensively tested using the GNU, Intel, Flang/Clang++, and NVIDIA HPC[1] 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
```

[1] Currently, with NVIDIA HPC compilers, arrays must be retrieved and altered using the get and set methods, respectively, in the readonly=.true. mode due to a compiler bug in nvfortran.

All Python attributes, including arrays of numbers and derived-type instances, are automatically exposed as Python objects 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 CASTEP (Clark et al., 2005), 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 currently supports the GNU compiler gfortran, the Intel compiler ifx, the LLVM compiler Flang, and the NVIDIA HPC compiler nvfortran.

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 f90wrap (Kermode, 2020). These tools process the Fortran source files and write out a wrapper layer, namely Python extension modules, plus, in the case of f90wrap, additional Fortran wrapper files, through which Python calls Fortran procedures and accesses Fortran data. The original sources are left untouched, although the generated wrapper layer must be compiled and maintained alongside the application. While F2PY on its own does not handle derived types, f90wrap extends it with support for modern Fortran features, including nested and allocatable derived types and zero-copy NumPy access to Fortran arrays, and since v0.3.0 it can alternatively generate a direct C layer without involving F2PY. By comparison, DL_PY2F generates no code at any stage and drives the whole Fortran application through a single entry point. Its Python-to-Fortran direction addresses a workflow the generator tools are not designed for, in which the primary data structures are created and owned in Python/NumPy and the Fortran side queries them by name at runtime. The application developer hand-writes this entry point as one small Fortran 2003 interface (such as `interface_app` above); no wrapper layer needs to be generated, built, or maintained. DL_PY2F also provides some capabilities beyond the generator tools, for example multi-dimensional arrays of derived types (see `b.a[2,:].ibuff` above), and callbacks held as ordinary object attributes, so that Fortran retrieves a Python callable by name at runtime just like any other datum. gfort2py (Farmer, 2017) is an ABI-level runtime tool that works similarly to the Fortran-to-Python layer of DL_PY2F: it likewise operates directly on an existing shared library and its module files without amendment to the source code, but is restricted to the gfortran compiler, whereas the Fortran-to-Python layer of DL_PY2F supports four compiler families (GNU, Intel, LLVM, and NVIDIA HPC). 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 invokes 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.

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, which also includes a comprehensive reference manual. There is also a comprehensive example demonstrating how to embed DL_PY2F in an application project. DL_PY2F has been published and deployed in a launchpad.net PPA and can be installed as a system package for Debian-based systems and is likewise available on PyPI. Please note that the PPA distribution works only with applications compiled with gfortran due to the pre-compiled Fortran module file we shipped.

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

Clark, S. J., Segall, M. D., Pickard, C. J., Hasnip, P. J., Probert, M. I. J., Refson, K., & Payne, M. C. (2005). First principles methods using CASTEP. *Zeitschrift für Kristallographie – Crystalline Materials*, *220*(5-6), 567–570. https://doi.org/10.1524/zkri.220.5.567.65075

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

Farmer, R. (2017). *gfort2py: Library to allow calling Fortran code from Python*. https://doi.org/10.5281/zenodo.846302

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

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*, *32*, 305901. 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. https://doi.org/10.1504/IJCSE.2009.029165

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