F2x - An Extensible Wrapper Generator for Fortran Modules

Michael Meinel
Intelligent and Distributed Systems
German Aerospace Center (DLR e.V.)
Berlin, Germany
michael.meinel@dlr.de

***Abstract*—F2x is a tool that wraps exiting Fortran codes to be used by other languages like Python. It uses a complete parser that captures many features of the Fortran language. The wrapping code is produced using template-based code generation. In this paper we present the F2x tool, compare it to the competitors, and show how it can be used to even more integration of Fortran into Python.**

***Index Terms*—Python, Fortran, Wrapper, Templates, Code generation, HPC**

1. Introduction

Fortran is still an important language for development of HPC codes. Due to its long history and broad set of features like OpenMP, OpenACC, and Co-Array Fortran and smart compilers it is also one of the best performing languages according to [1].

Compared to Fortran, the Python programming language is relatively new, especially to HPC development. Yet it already has a history in scientific programming. Tools and libraries like NumPy, SciPy, and Jupyter notebook make it a great environment for scientific applications and a cheap and valuable replacement for other languages and frameworks [2]. The growing importance of Python as HPC language is also reflected by the "Workshop on Python for High-Performance and Scientific Computing" (PyHPC) which takes place in context of the Supercomputing conference. In 2018 the 8th edition of the PyHPC Workshop was held. Especially the ability to integrate Python with C libraries makes it a preferred choice as high level integration language [3].

Python is often used as a high level language to coordinate different (HPC) codes. This includes lots of Fortran codes that are specifically written to do extensive arithmetic. Even if Python calls these Fortran codes by using files and sub processes a speedup is often achieved compared to pure Python implementations. However, it is still desirable to have a closer integration.

In 2009 Petersen published a paper about "F2PY: a tool for connecting fortran and python programs" [4]. Since then f2py has been included in the SciPy software bundle and has a very big user base. Despite its popularity, f2py is not actively developed anymore[1]. Only a subset of modern Fortran is supported by f2py and the code produced is highly platform-dependent.

Today, most probably all code that could be trivially wrapped with f2py has been handled. Some projects even restrict their Fortran usage to constructs that can be wrapped by f2py. Some legacy codes exist that are not easily wrapable with f2py. Other projects do not want to restrict the usage of Fortran features but still aim for Python interoperability.

F2x was developed to overcome the limitations of f2py while still being easy to use. The development started in 2014 and F2x is in active use in different Projects, e.g., at the German Space Operations Center [5], [6]. This allowed us to develop the software close to the requirement of the users. The design of F2x does not only focus on Python as a target but provides the flexibility to allow interactions with other languages.

The following paper is structured as follows:

- We describe the problem of integrating Fortran code with other languages and especially with Python in Section 2.
- We look at related software in Section 3. Especially the limitations of f2py are outlined to encourage the need for a new tool.
- We describe how F2x is implemented in Section 4.
- Finally we present how to use F2x and some benchmark results in Section 5.

2. Interaction with Fortran Code

In this section, we describe the problems that arise when you want to do calls from Python into Fortran code. First, we provide some background on how to expose an interface complying to C calling conventions from Fortran. Then we discuss some of the problems that needs to be handled when mixing different languages. Finally, the concrete case of mixing Python and Fortran is handled.

1. Despite a merge request for Python 3 support from December 2017, the activity log of the GitHub project at https://github.com/pearu/f2py dates the latest changes back to 2015.

2.1. Enforcing C Calling Conventions

Each Fortran compiler has its very own way of memory management as well as a somewhat special type system [7]. Due to this behavior, Fortran code is not ready for use from other languages in a generalized way. As an example, each compiler has its specifics on how to store dimensions of dynamic allocated arrays. Another problem is the translations of names called "mangling." Peterson [4] gives some insights on other problematic behavior of Fortran compilers. Consequently, a shim compatibility layer is required to allow interactions with Fortran code from other languages.

The most standardized and commonly supported interface for binary code interaction are C calling conventions. These conventions are also supported by many other languages like Python, Java, C# and so on. This makes C the optimal target for providing a compatibility layer. There are however at least two possibilities to approach the implementation of a C compatibility layer.

One approach uses the flexibility of C to mimic the behavior of Fortran compilers. This would result in an approach similar to f2py. However, this would be highly dependent on the target platform including the operating system and compiler suite. Yet, if you master the odds of this, you get very close integration without any loss of performance.

As another solution, Fortran provides the BIND(C) construct that was introduced with the Fortran 2003 standard [8]. This instructs the compiler to export interfaces in a way that it is accessible from C code. Instead of modelling the specifics of a compiler, the compiler produces binaries that are compatible with C. This results in a higher level of abstraction and platform independence. However, there are limitations which Fortran constructs may be used. Small pieces of Fortran code can accommodate for this. All major Fortran compilers support the BIND(C) feature. Hence, it is a well-supported way to achieve good portability and interoperability.

2.2. Interaction from Other Languages with C Code

Many languages have a C calling interface. This interface allows to interact with C libraries and covers almost all features that are known from C. Some examples of this functionality are the Java Native Interface (JNI), PInvoke for .Net, and many more. All of them support two important functionalities. First they provide means to create references to C functions that can be used from the target language. Second they provide a ruleset on how to access C data from the higher language. The process of translating data between C and other language representations is called "marshalling."

While most of the primitive data types can easily be marshalled between C and other languages, other constructs are a bit more complicated. Especially when more complex memory management like indirect pointers or dynamic arrays are used, the marshalling becomes non-trivial. This imposes a special problem when wrapping Fortran code. In Fortran all parameters are passed "by reference" by default, i.e., they are usually passed as pointers. Additionally, Fortran uses its own memory allocation techniques and array indexing. This makes it even harder to find generic solutions.

The current templates for F2x handle this problem by only exposing interfaces with primitive datatypes or opaque pointers. To access an element within an array or a field of a derive data type, getter and setter routines are generated. This approach allows flexible and portable code.

2.3. Interaction from Python with Fortran and C Code

There are several ways to call Fortran code from Python. The simplest is to call a Fortran executable from Python passing data using the file system. More sophisticated approaches use some C intermediate layer to call the Fortran routines. This intermediate layer could be anything from Python's own *ctypes* module, a solution like Cython, or a completely automated wrapper generator like Wrappy [9] or SWIG [10]. All have in common that you need to provide a C interface that can be accessed.

One solution is to write the accessing code by hand. This solution requires quite some knowledge and work. However, if you have enough insight, this most probably produces the results with best performance. For bigger code bases, an automated wrapper generator will be required. Automation also allows wrapping without too much insight into the details of the C compatibility layer.

3. Related Work

In the following section we will have a look on work that has been done already on the task of integrating Fortran code with other languages. First we have a look at f2py which does a similar job to F2x and is still the de-facto standard for integrating Fortran code into Python. We will also have a look at other wrapper generators that are available on the market. Finally, we present some work considering template-based code generation.

3.1. f2py

The focus of f2py is usability and performance. These goals are well achieved with the current implementation. However, the original author already identified some shortcomings of the tool which are partly due to the way it was implemented [4]:

- The generated interface code is highly dependent on the Fortran compiler is use.
- Due to direct usage of the Python C API, it is also dependent on the Python version in use. This was especially problematic when the API changed between Python 2 and Python 3. However, support for Python 3 has been added lately.
- Only primitives and primitive arrays are natively supported by f2py. To add support of other data

types, additional interface information or additional wrapping is needed.

Even though these restrictions might be a show stopper for some tools, they allow in return to have a fast and easy to use wrapping approach.

Another downside of f2py is its architecture. Before the work on F2x was started, some investigations were made how feasible it would be to extend f2py. However, while the compact code is well written and understandable, it was pretty hard to extend it e.g., to support derived types.

3.2. Other Automated Wrapper Generators

There are no alternative wrappers for Fortran code available. In contrast, there is a wide range of wrapper generators for C and C++ code available. The latter could be used to automatically generate wrappers that interact with Fortran code that is exposed using C calling conventions. Some popular implementations of automated C/C++ wrappers include SWIG [11] or Shiboken [12].

All of these C wrapper generators have in common that they expect some sort of C/C++ interface definition. This is usually included in a C header file and always specified using C syntax. For the use case targeted by F2x (i.e., wrapping Fortran code), usually there is no C interface definition available. This needs to be written by hand or could be generated based on the source code. The former approach would require too much manual work. For the latter approach, instead of generating a C interface definition the whole wrapper could be generated just as well.

3.3. Template-based Code Generation

Template-based code generation is often used in model driven software development. It helps derive different textual representations from an abstract model [13]. Source code is generated using templates that contain placeholders. These placeholders are filled with the corresponding parts of a model instance.

In context of F2x the model is represented by a reduced form of the abstract syntax tree as described in Section 4.3. This is transformed using different templates to generate the different code layers (e.g., C compatible layer, Python integration layer, etc.).

Lots of template systems are available that can be used for template-based code generation. Luhunu [14] provides a classification system and categorizes some Java specific implementations. F2x employs the Jinja2 template system [15]. This system is in broad use throughout the Python community. Jinja2 is mainly used for web-related applications generating HTML. However, it can be used for building templates for any text-based languages. According to the classifications of Luhunu [14] Jinja2 allows navigation, variable dependency and recursion. It also allows flow control constructs like conditionals and loops that can be used to implement polymorphism. Thus it is a very powerful implementation covering any scenarios required for complex code generation needs.

4. Automated Wrappers with F2x

In this section, we explain how F2x is implemented to allow automated wrapper generation for Fortran codes. First we give an overview of how the wrapping process is organized. Then we have a more detailed look at the parser, the model, and the code generator. Finally we explain how the Python interfaces to the Fortran code are implemented in the template.

4.1. How Does It Work?

F2x was developed to fill the gap left by f2py. It parses Fortran sources based on a LALR(1) grammar [16] and applies templates to transform the resulting syntax tree representation into executable wrapper code. It is implemented in Python.

The general workflow appears in Fig. 1. Internally, the code generation happens in thee steps:

1) The Fortran source code is preprocessed.
2) The preprocessed Fortran source code is parsed into some kind of tree representation called “Abstract Generator Tree” (AGT).
3) By applying the templates to the AGT, the actual wrapper code is generated.

The preprocessing step transforms the Fortran source to facilitate the parsing step. Most important in this step is that executable source lines are commented out. This helps to speed up the parsing process as no executable statements need to be parsed anymore. In consequence, also the grammar can be stripped. Preprocessing also reduces the complexity of the code and with it the intermediate abstract syntax tree. In consequence, also the Abstract Generation Tree can be create much faster as it is derived from the abstract syntax tree.

The parsing step reads the preprocessed source files and produces an abstract syntax tree for it. This tree could be already used as a model for code generation. However, the form and content of the abstract syntax tree is highly dependent on the parser and grammar used. To allow replacing the parser part, the parser specific abstract syntax tree is transformed into a generalized Abstract Generation Tree. All information about executable code is stripped from this tree. It contains only those information that are required to generate the wrapper code. This is mainly everything that describes the interfaces and parameter types.

In a final step, the AGT is taken and transformed by templates into wrapper code. Usually there is one template—possibly with child templates—for every layer of integration. Most of the logic required for code generation is also put into the templates.

Each source file can be augmented with a configuration file. This configuration may contain some additional information which are processed at the different stages. Some examples are replacing and ignoring complete lines, renaming variables, adapting the name of exported functions, etc.

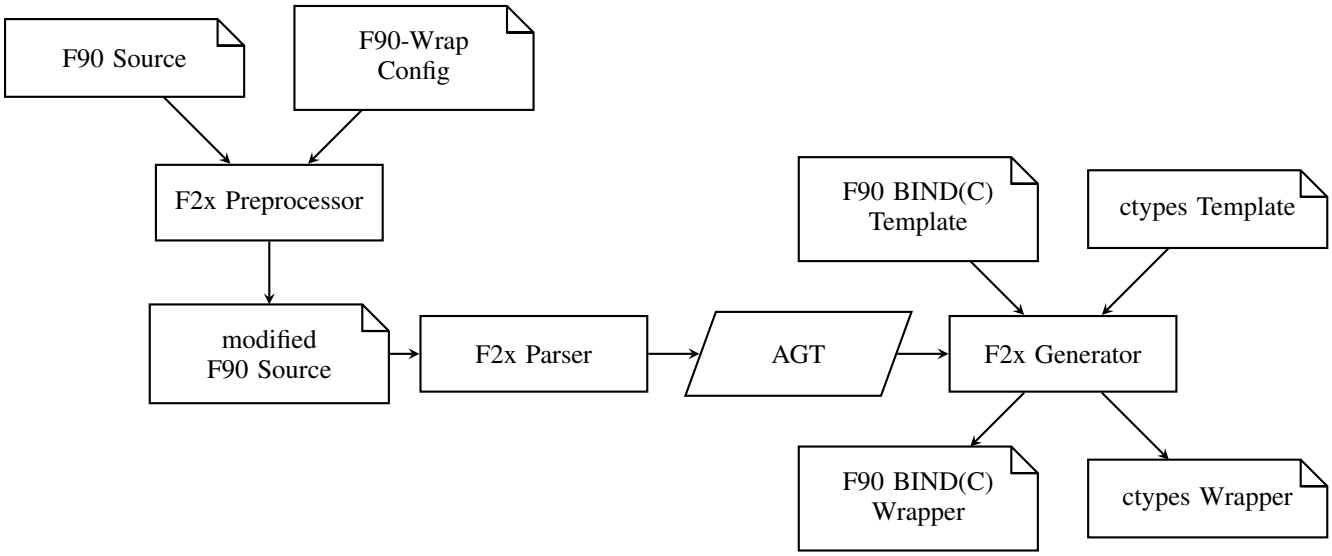

Figure 1. Example workflow of F2x

To allow easy customization and extension F2x was developed with high modularity in mind. One goal is to allow different target languages to be supported, so especially the code generation part needs to be easily extendable. To address this goal, a template-based approach was chosen for output generation.

The model used to instantiate the templates is an abstraction of the syntax tree produced by the parser component. This model reduces the complexity and provides a clean extension point allowing to replace the parser component with another parser that produces an equivalent model.

To allow F2x to be adapted, it consists of three main modules: Parser frontend with optional pre-processor, abstract model (Generation Tree), and template-based code generator. All of them are exchangeable and extensible to allow for experimentation and easy adoption.

4.2. Fortran Parser

As of this writing an almost complete Fortran grammar in some dialect of the Backus-Naur-Form is used. This grammar was adopted from Open Fortran Project [17]. Based on the grammar, plyplus [18] generates a LALR(1) parser. While this is a pretty robust way of processing the Fortran sources, it also has some drawbacks. A preprocessor is required to remove ambiguities. Due to the high flexibility and dynamic nature of plyplus, parser generation and parsing are also pretty slow. Some effort has been taken to reduce the grammar by stripping off execution parts of it (i.e., function and subroutine bodies).

As the plyplus project has been discontinued in favor of its successor, some alternative approaches are under investigation. One is to use the successor of plyplus. This was not very successful as the performance decreased even more. Another approach would be to generate a static parser using ANTLR [19]. Thanks to the high modularity of the F2x architecture, new parsers can easily be added as optional features for the software package.

4.3. Abstract Generation Tree

The Abstract Generation Tree is the central model used for generating interface code. As generation of interfaces does not need to know about the actual implementation behind the functions that will be exported, this information is not preserved by the parser. Basically, only the interface information about Fortran modules, subroutines and functions and their parameters is contained in the model.

The implementation of the AGT is not directly dependent on any information from the parser. This design allows to decouple the parsing step from the code generation. The following information is available in the tree.

- The **module** is the top level node of the AGT. It contains the name of the module to be exported as well as collections of imported modules, global variables, types, and exported functions and subroutines.
- The **types** collections stores information about derived types declared within this module. All kind of fields and nesting of derived types is supported. Especially, the type support is not limited to those types that are compatible with the BIND(C) feature (e.g., string arrays).
- The **globals** collection contains all global available variables and parameters. They can be of any type, however global arrays need to be of type POINTER or TARGET. In future implementations this might be removed with improved templates.
- Finally, the module has a collection of **methods** which combines all exported subroutines and functions.
- Each type has a collection of **fields**.

- The functions and subroutines also carry a child collection of **arguments** that are passed to the method.

Figure 2 shows the class diagram with all nodes types that define an AGT.

4.4. Template-based Code Generator

The Abstract Generation Tree is the model that is used for the code generation. It is passed to the code generator that applies the templates to the model. F2x uses the Jinja2 library for template processing. Different Jinja2 templates exist that implement the different wrapping layers and stategies.

Jinja2 allows to define macros and import other templates. These features are used to structure and facilitate the maintenance of the templates. Yet, due to the many possibilities that Fortran allows, the templates are still rather complex.

The generator supports loading of bundled templates as well as custom, user-defined templates. This design allows to create own templates or extend existing templates in an easy way. The same mechanism can also be used to interface with other languages not yet supported by F2x.

4.5. Python Interfacing Strategies

As described in 2, the primary goals for development of the F2x generation templates were to be portable between different compilers and to restrict the usage of modern Fortran features as much as possible. Hence, there are two layers of generated interface code.

To allow usage of arrays and derived types, a whole bunch of getter and setter routines are generated. These routines use standard compliant Fortran that is exported using the BIND(C) feature to allow access using C calling convention. This makes the usage of complex data structures easily available to other languages without depending on knowledge about the internals of different compilers. However, this approach also requires to generate a thin wrapper layer around the actual functions that should be exported to convert between opaque C datatypes and their internal Fortran counterparts.

While this C layer is enough to allow access to all the Fortran functionality, the exported interfaces are not very easy to use and require some special attention at some places. To allow a better interaction of the developer with the code, a second layer is generated that wraps the exported C interfaces in easy to use Python objects and functions. Consequently, all exported types and functions are available in a pythonic way.

Listings 1 and 2 show a simple example of a Fortran code and how it can be accessed from Python.

Listing 1. An example Fortran code

```
MODULE EXAMPLE
    IMPLICIT NONE
    PRIVATE

    TYPE, PUBLIC :: POLYNOM
        INTEGER :: DEGREE
        REAL(8), DIMENSION(:), &
            ALLOCATABLE :: COEFF
    END TYPE

    PUBLIC EVAL_AT

CONTAINS

    FUNCTION EVAL_AT(P, X)
        REAL(8) :: EVAL_AT
        TYPE(POLYNOM), INTENT(IN) :: P
        REAL(8), INTENT(IN) :: X

        INTEGER :: I
        REAL(8) :: RES

        RES = 0.0_8
        DO I = 0, P%DEGREE
            RES = RES &
                + P%COEFF(I + 1) * (X ** I)
        END DO
        EVAL_AT = RES
    END FUNCTION

END
```

Listing 2. Usage of Fortran code from Python

```
from example_glue import POLYNOM, EVAL_AT

# p: f(x) = 3.4 * x^3 + 1.2 * x.
p = POLYNOM(DEGREE=3,
            COEFF=[0, 1.2, 0, 3.4])
c = EVAL_AT(p, 5.6)

# Update offset of p.
p.COEFF[0] = -c
assert abs(EVAL_AT(p, 5.6)) < 0.0001
```

5. Application of F2x

This section concentrates on the application of F2x to exiting Fortran code. First we show how to wrap a Fortran module. After that we give an overview of implemented templates that can be used. Then we show some benchmarks benchmarks to evaluate the performance of F2x compared to f2py. Finally we give some remarks about how F2x is used within the German Aerospace Center and how it is available.

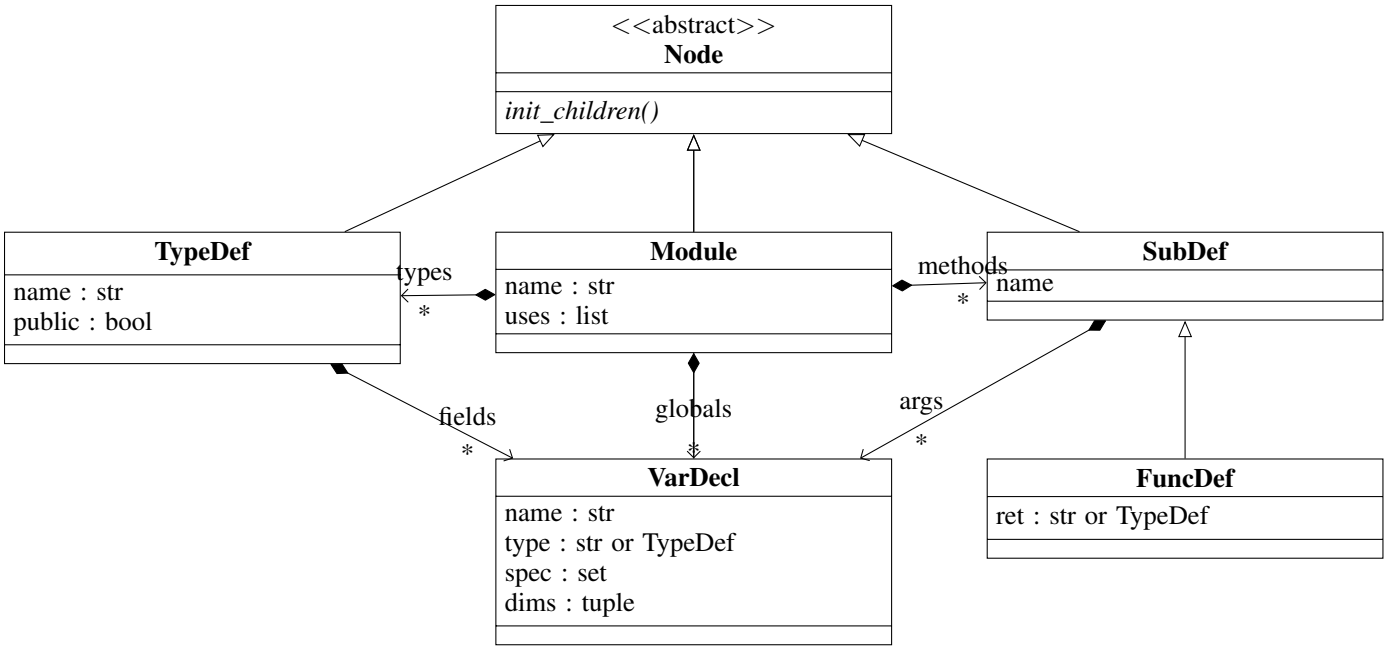

Figure 2. Class diagram for the AGT.

5.1. Wrapping Process

F2x has no support for a one step wrapping process like f2py yet. However, it only requires invocation of a single command line tool. The rest should be familiar to a Fortran developer. The whole process can easily be automated and integrated into any build process.

To wrap a simple Fortran module like the one in Listing 1 you need to do the following steps:

1) Generate the wrapper code using F2x. Usually this is as easy as running the F2x command line tool with appropriate arguments. It does not need any customization of the Fortran code or creation of extra interface definition files, although some mechanism for fine tuning exists. However, due to the incorporated parser technique, the runtime is pretty slow.

```
$ F2x -t @bindc/_wrap.f90.t \
  -t @ctypes/_wrap.py.t example.f90
```

2) Compile the Fortran code and the wrapper modules into a library. This should not be to complicated for any experienced Fortran developer as it does not require any special knowledge. The resulting library can in turn be used by any language that is able to access libraries with C calling conventions. It only needs to be regenerated when the Fortran code changes or if a new platform should be supported.

```
$ gfortran -shared -fPIC \
  -o libexample.so example.f90 \
  example_wrap.f90
```

3) The library and the Python modules need to be packaged and distributed.

This process can easily be incorporated into any common build system. However, especially when the Cython templates are available it would be pretty useful to also provide some extensions to the Python distutils framework to allow even easier creation of wrapped Python packages.

5.2. Implemented Templates

Right now, there are already a bunch of templates for code generation available.

bindc This set of templates produces the C compatible wrapper code that is required to make the Fortran code available for other languages. It is part of the base F2x distribution and implements all features that are available for F2x.

ctypes This is a set of templates that uses the interfaces provided by the *bindc* templates to make the exported Fortran code available to Python. It uses the ctypes package to access the C layer. While this is known to be very slow, this approach was used as reference implementation as it does not require any additional Python packages to be installed. These templates are also part of the F2x distribution and implement everything that is supported by F2x.

cerr This is one of the extended templates specially created for the needs of the GSOC. It provides another thin C layer between the wrappers generated by *bindc* and *ctypes*. The primary goal

was to provide a clean C stack layer that is used as a return point to enable treating errors using setjmp/longjmp. As it is a pretty special use case, it is not included in the F2x distribution.

cython These templates are meant to be a faster replacement of the *ctypes* implementation. They are under heavy development and do not support the full feature set of F2x yet. However, first benchmarks have shown that this approach provides a much better performance than the current implementation.

The base set of *bindc* and *ctypes* templates makes F2x reliably usable but with limited performance. The Cython template will show a direct improvement once they are done. However, it is also possible to write your own templates.

5.3. Benchmarks

If Fortran code is called from Python using the F2x ctypes wrapper approach it is a lot slower than using f2py. However, using the Cython approach to access the C interface, the result show even better performance compared to f2py while still allowing more Fortran features to be used.

Table 1 displays the benchmark results. For each implementation and benchmark, the absolute runtime (*abs.*) and the relative runtime compared with f2py (*rel.*) are given. Figure 3 shows a plot of the relative results.

For this benchmark some Fortran routines with very common interfaces have been wrapped. f2py and F2x with different templates have been used. All wrapping was done using the default settings of the respective tool. Compilation was done using the GNU Fortran compiler version 7.3.1 without special optimization flags. The host system was running SuSE Leap 42.3 on a x86_64 architecture with Python version 3.4.6.[2]

The content of the Fortran routines were kept simple to minimize their impact on the results. The measurement was done using the Python timeit module with default settings (i.e., 1,000,000 calls to the benchmark routines each).

The overall runtime of the benchmarks were normalized to the performance shown by *f2py*. F2x with ctypes-templates (*F2x + ctypes*) was expected to be slow. The performance of F2x with Cython-templates (*F2x + Cython*) had the same magnitude as *f2py* and was even a little better. For some special cases an even better performance could be generated by specifically crafted templates.

5.4. Usage in Projects

F2x was used in projects from the beginning of its development at the German Aerospace Center for different projects. Especially the German Space Operations Center uses it for several applications [5], [6].

Unfortunately a direct comparison of F2x and f2py for these applications is not possible due to the restrictions of f2py. However, some attempts were made to manually adapt Fortran codes in use to be wrapable by f2py. The need to adapt production codes for making it accessible from Python was drastically reduced with the introduction of F2x.

2. These benchmarks can be reproduced using the code at https://doi.org/10.5281/zenodo.1405459.

TABLE 1. RUNTIME OF THE DIFFERENT BENCHMARKS (ABSOLUTE IN SECONDS, AND RELATIVE TO F2PY).

	primitives		**array out**		**huge array**	
Implementation	abs.	rel.	abs.	rel.	abs.	rel.
f2py	3.6	1.0	1.16	1.0	0.82	1.0
F2x + Cython	3.06	0.85	1.37	1.18	1.1	1.33
F2x + ctypes	12.84	3.57	16.22	13.96	15.84	19.21

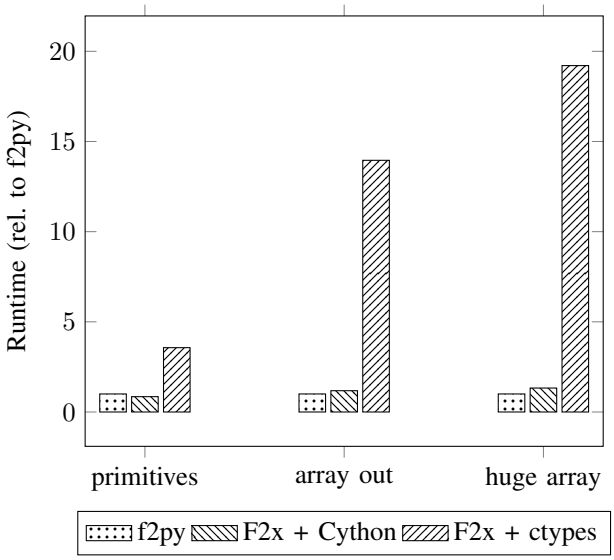

Figure 3. Benchmark results comparing *f2py* with different *F2x* templates.

5.5. Availability

F2x is licensed under the Apache Software License 2.0 and available on GitHub[3]. It already supports functions, subroutines, derived data types, arrays, and globals. New users and contributors are welcome.

6. Conclusions

We presented F2x as new tool to make Fortran code accessible from Python (and other languages). We outlined which obstacles we wanted to master that are not covered by existing work. Then we explained how F2x work to provide the functionality of automated Fortran wrapper generation. Finally we presented how F2x can be used in practice.

F2x is available as Open Source software. It is already used for production systems in different projects. Many features are already implemented while some are still missing. Usually it is already comparable to f2py and under certain circumstance can also outperform f2py.

3. https://github.com/DLR-SC/F2x

Development is ongoing and next to improvements of the templates convenience features are to be added. These include a faster, more robust parser and better integration with the Python build process. The goal is to allow a one-shot module generation similar to f2py.

References

[1] E. Loh, "The ideal hpc programming language," *Queue*, vol. 8, no. 6, pp. 30:30–30:38, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1810226.1820518

[2] T. E. Oliphant, "Python for scientific computing," *Computing in Science Engineering*, vol. 9, no. 3, pp. 10–20, May 2007.

[3] F. Perez, B. E. Granger, and J. D. Hunter, "Python: An ecosystem for scientific computing," *Computing in Science Engineering*, vol. 13, no. 2, pp. 13–21, March 2011.

[4] P. Peterson, "F2PY: a tool for connecting fortran and python programs," *IJCSE*, vol. 4, no. 4, pp. 296–305, 2009. [Online]. Available: https://doi.org/10.1504/IJCSE.2009.029165

[5] M. Weigel, M. Meinel, and H. Fiedler, "Processing of optical telescope observations with the space object catalogue bacardi," in *25th International Symposium on Space Flight Dynamics ISSFD*, Oktober 2015. [Online]. Available: https://elib.dlr.de/101947/

[6] H. Fiedler, J. Herzog, M. Prohaska, T. Schildknecht, and M. Weigel, "Smartnet(tm) - status and statistics," in *International Astronautical Congress*, Oktober 2017. [Online]. Available: https://elib.dlr.de/115884/

[7] J. Appleyard, "Fortran compiler comparisons 2016-2017," *SIGPLAN Fortran Forum*, vol. 35, no. 3, pp. 10–28, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022868.3022869

[8] "Information technology – programming languages – fortran – part 1: Base language," International Organization for Standardization, Standard, 2004.

[9] G. S. Couch, C. C. Huang, and T. E. Ferrin, "Wrappy – a python wrapper generator for c++ classes," in *O'Reilly Open Source Convention Python Conference Proceedings*, 1999. [Online]. Available: http://conferences.oreilly.com/

[10] D. M. Beazley and P. S. Lomdahl, "Feeding a large-scale physics application to python," in *6th International Python Conference*, San Jose, CA, October 1997. [Online]. Available: http://www.swig.org/papers/Tcl96/tcl96.html

[11] "Swig." [Online]. Available: https://www.swig.org

[12] "Pyside shiboken." [Online]. Available: https://wiki.qt.io/PySide_Shiboken

[13] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[14] L. Luhunu and E. Syriani, "Survey on template-based code generation," in *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.*, 2017, pp. 468–471. [Online]. Available: http://ceur-ws.org/Vol-2019/posters_3.pdf

[15] "Jinja." [Online]. Available: https://jinja.pocoo.org

[16] F. L. Deremer, "Practical translators for lr(k) languages," MIT, Cambridge, MA, USA, Tech. Rep., 1969.

[17] "Open fortran parser." [Online]. Available: http://fortran-parser.sourceforge.net/

[18] "plyplus." [Online]. Available: https://github.com/erezsh/plyplus

[19] "Antlrv4." [Online]. Available: http://www.antlr.org/