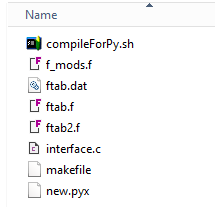
FTAB conversion to Python

# Example Code Overview:

You will need the following:



**Fortran**:

1. ftab.f (main ftab code)
2. ftab2.f (supplemental ftab code)
3. f\_mods.f (code with interface iso\_c\_binding modules)

**C**:

1. interface.c (routines that interface between fortran and python code)

**Cython/Python**:

1. new.pyx (new python code replacing legacy fortran code)

**Additional files**:

1. compileForPy.sh (script to create the ftab executable – ftab.exe)
2. makefile (used by compileForPy.sh)
3. ftab.dat (needed to run ftab.exe)

# Author’s Note

The conversion methodology outlined in this report was conducted as part of a research effort to evaluate approaches for modernizing legacy Fortran code. This work was conducted under an assumption that one of the easiest ways to update legacy Fortran code is to select targeted functions or subroutines and rewrite the selected sections using best practices for code development, documentation, and testing.

The benefits of this approach are that it allows for iterative and incremental (i.e., agile) development and testing. In other words, the original code would never be offline but would always be running with the latest updates. Under the methodology outlined below, developers would swap out legacy code with updated code during a series of sprints. At the end of each sprint, users would run updated code with the latest round of delivered changes. The expectation is that the software would continue to perform as expected. If not, it would be easy enough to trace and address the issue. This is as opposed to a major overhaul approach which would deliver an updated product many months down the line which would be difficult to troubleshoot if users experienced difficulties after delivery.

In this study, as a proof of concept, Python (with Cython) was selected for replacing a Fortran function which is called to sum numbers in an array and return the calculated value. Output from the original code was compared to that of the modified code. Results appeared equivalent. Thus, in that respect, this conversion methodology is deemed a tenable approach.

However, it should be noted, this report only evaluates the feasibility of converting Fortran to Python (with Cython as the intermediary). A number of considerations remain unevaluated. Any undertaking of this magnitude should begin with a thorough understanding of the function of the model being updated. This would include discussions with modelers with both thorough and historical knowledge of the model. Everyone should agree on any programming language(s) to which existing code would be migrated. Concerns about the migration should be brought up and addressed before determining an approach. Best practices for code development, documentation, and testing should be clearly defined. Modernization goals should include eliminating spaghetti code, identifying and correcting bugs in the code, and improving overall code maintainability, readability, and performance such that development and execution times are minimized as much as possible.

# Justification for Modernization using Cython:

There is a desire to update legacy Fortran to a more modern language. Python seems like a reasonable choice given the following considerations:

1. Python’s current popularity makes finding experienced developers easier
2. The relative ease of learning and writing Python code reduces development times
3. Python is a flexible, dynamically-typed, garbage-collected language, thus removing the need to deal with issues such as memory allocation, resizing arrays, specifying types or intents
4. Python interpreters make it easier to test small bits of code before implementing in larger code
5. Python has a wide community of developers which means a great selection of libraries (i.e., a solution to your problem probably already exists and is a Google search away)
6. Python interfaces very well with existing C/C++ code. This latter feature is important as it allows use of Fortran’s intrinsic iso\_c\_binding module (further discussion below)

Several options exist for transitioning from Fortran to Python. Three common options are:

1. Python’s C-API
2. F2PY
3. Cython

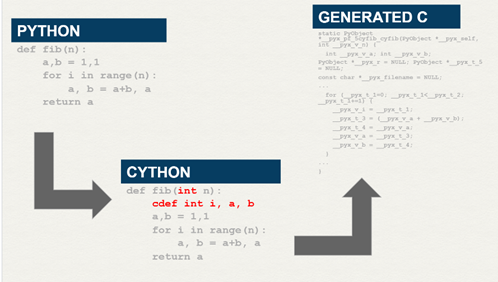
Option 1, though the most general way of transitioning, is the most work intensive as it requires writing a lot of C wrapper code. Option 2 uses F2PY to essentially automate the compilation of Fortran code into Python modules. Although F2PY is a popular tool with many benefits, it is **not** assessed in this report as ideally suited for converting Fortran to Python if more recent versions of Fortran have been used.

This assessment is based on several considerations. First, the tool has some limitations with regards to more modern Fortran (i.e., Fortran 90+). In general, F2PY can be used to wrap any Fortran 77 code to Python. However, for programs written using Fortran 90 or newer, the wrapping only works reliably on certain subsets of modern Fortran’s features [*Peterson, P., F2PY: A tool for connection Fortran and Python programs, Int. J. Computational Science and Engineering, Vol 4, 2009*; *Cross, M., Calling the NAG Fortran Library from Python using F2PY, NP3665, 2009*]. This could be problematic for Fortran code which has been developed over the years using different versions of Fortran (e.g., from Fortran 90 to 2008/2018), thus the code is a mix of constructs and syntax ([The Difference Between FORTRAN - Now And Then](https://www.electronicdesign.com/compilers/whats-difference-between-fortran-now-and-then)). Successful run performance can depend on compiler flags[[1]](#footnote-2) and there have been at least a few documented instances of problems using F2PY in place of some standard Fortran compilers [e.g., *Cross, M. 2009*]. To avoid potential errors during conversion, F2PY requires Fortran code that strongly types and specifies intents for all variables [*Shell, S., Principles of modern molecular simulation methods,* [*engineering.ucsb.edu*](https://sites.engineering.ucsb.edu/~shell/che210d/f2py.pdf) *UC Santa Barbara, 2019*]. Finally, because F2PY automates the conversion, details of what the tool does are not transparent to a user so it could be more difficult to debug or distinguish between compilation vs. coding errors.

A model that has been built and continually developed over decades will have had a number of developers with varying coding practices who will have worked on the code over the years, thus a conversion approach using Cython is favored by the author since it will allow for standardization and the cleaning of current spaghetti code. Additionally, conversion via Cython will hopefully limit problems that could arise due to differences in compiler flag options between ifort and F2PY.

The basic properties of Cython, Python, and Fortran that permit an iterative and incremental conversion delivery process are:

1. Python (the CPython implementation of it) is built on top of a C API
2. Cython makes use of CPython’s C API to translate Python code into the C equivalent of what the Python runtime would be doing and compiling this into machine code
3. As of 2003, mixed-language procedure calls between the programming languages C and Fortran have been standardized using Fortran’s intrinsic module, iso\_c\_binding. This ensures compatibility and interoperability between the two languages. This also means Python code (once automatically translated to C code by Cython) can be called by Fortran.
4. As a bonus, Cython is commonly used to speed up Python code. If we already have existing Cython code, this could be used to improve performance down the road if needed.



The basic steps of converting Fortran into Python are:

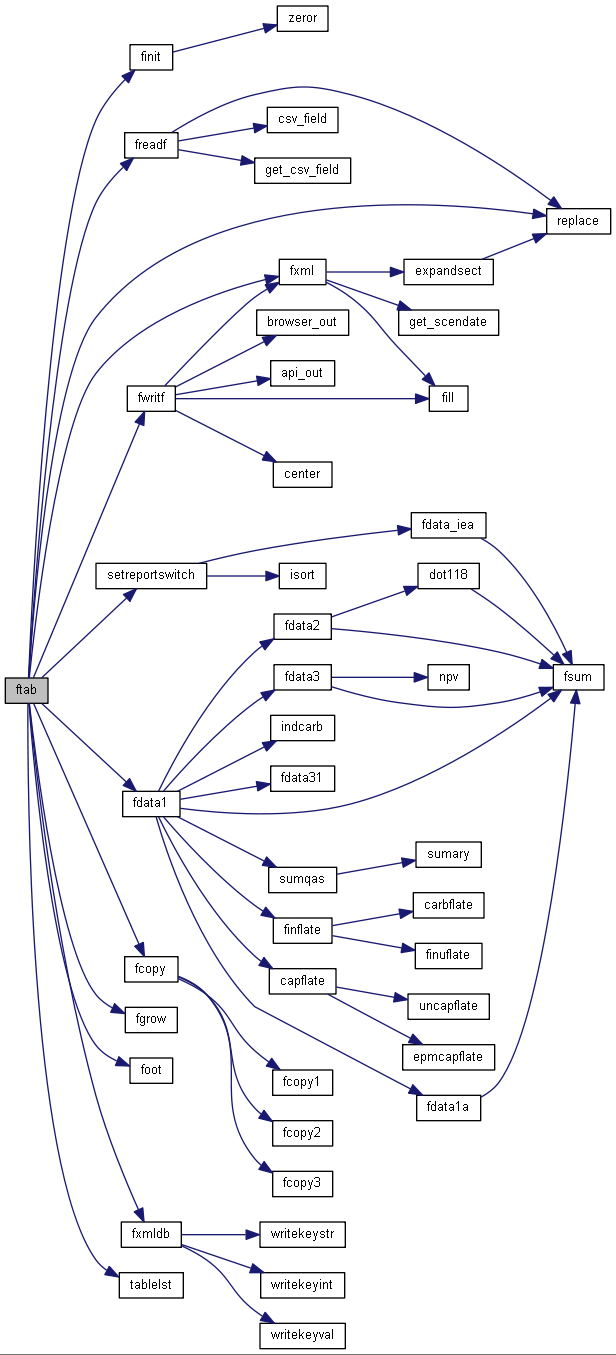
1. Identify a Fortran subroutine or function for conversion (more on selection approach in following sections)
2. Re-write the Fortran routine in Python
3. Use Cython to auto-translate the Python to its C equivalent
4. Call the C-translated Python code from Fortran
5. As you expand then number of routines written in Python, Python code can call other Python code without translation to C. Thus, theoretically you would only need some C interface code between the remaining Fortran and the new Python code. (Proof of concept for this was beyond the scope of this report.)

# Explanation of Methodology Used in Study:

## Call graph:

The generation and review of a call graph is a reasonable first step in converting a complex, legacy code from Fortran. In the following use case, Doxygen is used to generate a call graph of ftab.

A sample input file for Doxygen and instructions on using Doxygen are located here.



*Sample of call graph generated by Doxygen*

The above call graph was used to identify peripheral subroutines/functions since these could be easily swapped out with equivalent Python code.

## Modifying Fortran Code:

The function **fsum** was selected as the starting edge routine (see call graph above). The existing Fortran code was replaced with Python via the following elements:

1. Fortran interface iso\_c\_binding module (.f or .f90)
2. C interface allowing Fortran calls to Cython/Python routines (.c)
3. Cython/Python code that replace existing Fortran functions (.py or .pyx)

Minimal changes to the legacy code were required since most of the code changes are external to the original Fortran (i.e., the Fortran iso\_c\_binding module, C interface, and Cython/Python code are written in separate files that are linked to the legacy code during the compilation steps, more on this below).

Only two notable changes to the existing code were required. The first change was a **use** statement had to be added to the beginning of all programs, subroutines, or functions that call the new Cython/Python routine. The syntax for the use statement is

use *interface\_module\_name*

where ***interface\_module\_name*** is taken from the external Fortran file containing the iso\_c\_binding module.

For example, if your module file contains the following code:

module **f2csum**

interface

real (kind=c\_float) function c\_fsum(a,n) bind(c,name='c\_fsum')

use, intrinsic :: iso\_c\_binding

real (kind=c\_float) :: a(\*)

integer (kind=c\_int), value :: n

end function c\_fsum

end interface

end module f2csum

then you will add **use f2csum** to the top of the original Fortran code:

!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SUBROUTINE FDATA3(IS)

**use f2csum**

IMPLICIT NONE

The other required change was to replace the old function call with the new.

That is, the old Fortran function, FSUM, was replaced with C\_FSUM. As a convention, this study decided all new functions will begin with C\_ followed by the name of the original function that is being replaced. The *C\_* is used to indicate that a legacy Fortran function has been replaced by Cython/Python code via the intermediate call to the C interface code.

# Compiling & Running on nem servers:

Open 2 command prompt windows

In one prompt, activate a virtual environment with Cython installed

(for information on virtual environments see Setting up a Python virtual environment)

In both prompts, navigate to location of code

## In the prompt where you activated the virtual environment

Enter the following commands into the prompt

"C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64\vcvars64.bat"

cython -3 new.pyx

Check the files new.c, new.h were created



If the .c and .h files were generated, then run the command below

cl -c -I Python\envs\pyf\Include interface.c new.c

You should see a message telling you the Microsoft C compiler is generating code…



The above step should have created the object files: interface.obj and new.obj



## In the second prompt you opened

Next create the remaining .obj files for the Fortran codes and compile all the .obj files to make the ftab executable.

This is done by running the script compileForPy.sh

. compileForPy.sh

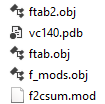
Processing ftab.f may take a few minutes.

The script has two parts. The first part creates the Fortran object files.

Once the first part of the script completes, you will see the message *Object files created!* printed to the screen



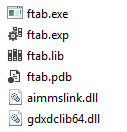
You should have the following files vc140.pdb, f\_mods.obj, f2csum.mod, ftab.obj and ftab2.obj



The second part of the script links the object files and creates the executable (ftab.exe). Once the Fortran files are processed you will see the message *\* Making ftab.exe \** printed to the screen.



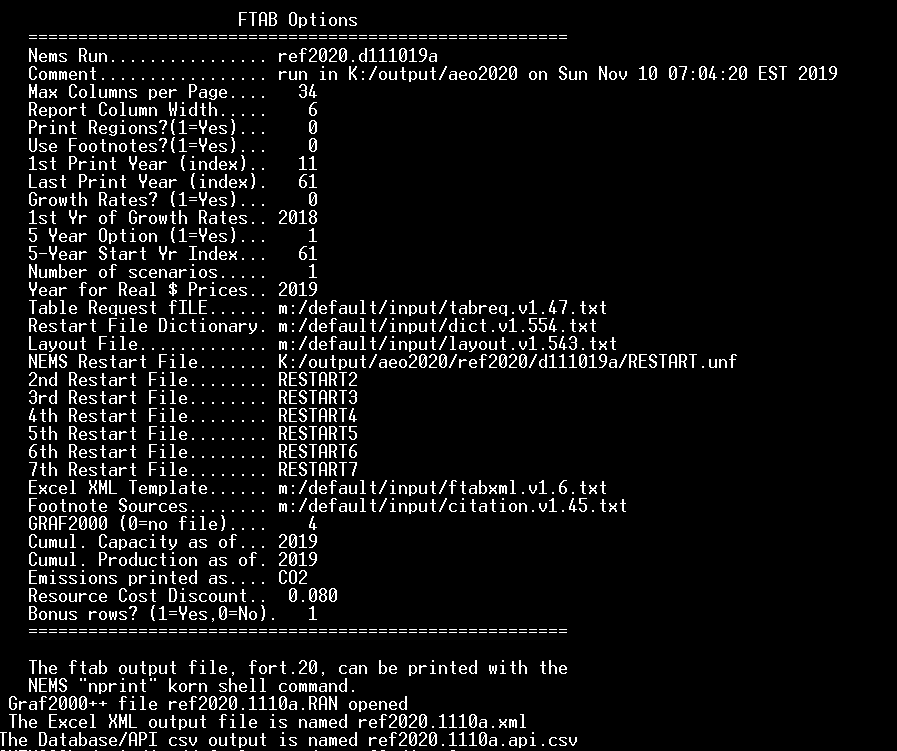
Once the script is complete you should see ftab.exe and the files needed to run it (ftab.exp, ftab.lib, ftab.pdb, aimmslink.dll, and gdxdclib64.dll.



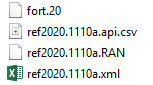
Execute ftab.exe and see if it runs/works. To do this you will need a copy of an ftab.dat file.

ftab.exe ftab.dat

It may take a minute to run. You should see text similar to below printed to the screen:



When done, you should have generated a fort.20, \*.api.csv, \*.RAN, and \*.xml (where \* = scenario name and date key of the run from which the ftab.dat was taken, e.g., *ref2020.1110a*).



You can diff the resulting fort.20 with the original to check everything looks good.

1. As of October 2019, the Fortran code under consideration was compiled using ifort version 18. It is unclear if the necessary ifort flags are available with F2PY. [↑](#footnote-ref-2)