University of Waterloo

Faculty of Engineering

Using Cython to Create Fortran-Python Interfaces of Coastal Interpolation Tool

Meteorological Service of Canada

Dorval, Québec

Submitted by

Anirudh Kilambi

20816406

anirudh.kilambi@uwaterloo.ca

2B Chemical Engineering

Date of submission: November 25, 2021

Executive Summary

The purpose of this report is to determine the best solution for creating a Python interface for an interpolation tool written in Fortran and analyzing the effects using this interface has on performance of the tool. The original interpolation tool is a large program, with over 15 dependencies and thousands of lines of code. While changes to the underlying model had to be made, the goal of this task was not to improve on the accuracy of the tool. Rather it was to improve the ease of use of the tool, while minimizing the performance loss.

In order to determine the best path forward, various wrapping techniques were used and analyzed including ctypes (Python Software Foundation, n.d.), F2PY (The NumPy Foundation, n.d.), and Cython (Behnel et al., 2010) to determine if they had the feature support needed, while also didn't add too much overhead to the program. It was determined that Cython had the desired feature support needed, and did not greatly increase the runtime for the tool.

As the code is not yet complete, the final analysis on the performance will be done at a later date.

Table of Contents

List of Figures

List of Tables

1. Introduction

The Royal Meteorological Society of Canada (RMS), a division of Environment and Climate Change Canada, is responsible for disseminating meteorological data, weather forecasts and warnings, as well as observing and performing research on various subsets of environmental science such as oceanography, atmospheric science, and climate conditions.

In order to provide the most accurate information possible in this modern era, computer programs must be modified and developed to perform the necessary calculations. One of these programs is a coastal interpolation tool. This coastal interpolation tool is written in Fortran. Specifically, Fortran 90 and is responsible for interpolating a value for a destination grid within a source grid.

While the code is perfectly functioning, it can be confusing for someone who was not involved in the development to use. Additionally, the code was not written by software developers. Rather, it was written by scientists whose main goal was to ensure it works to the best of their ability. Because of this, best coding practices were not used, and the code could use some streamlining.

To improve the ease of use of the program, a python interface was desired to create a pseudo front-end of the application for operations to use.

The goals of this report is to determine the best method for creating the Python interface, as well as analyzing the performance drawbacks and possibly more long term solutions for improving the tool.

2. Background

2.1. Fortran90

In computing, Fortran (short for *For*mula *Tran*slation) is a veritable dinosaur. Initially conceived in 1957, the language is the oldest commercial programming language in the world. As a result of this age, it has slowly started to fall out of use. With the TIOBE programming community index placing it as the 19th most popular programming language as of November 2021 (Tiobe Software BV, 2021). However, while Fortran in its original state is rarely seen, modern Fortran (Fortran 90, 95 03, and 08) are still very popular in scientific circles. Especially when attempting to model large scale environmental processes.

There are two major compilers that can be used for Fortran compilation – the GNU Compiler Edition's *gfortran*, an open source and free compiler for Fortran code, as well as the Intel Fortran Compiler. The RMS utilizes, the Intel Fortran Compiler, and therefore it will be used for the testing done in this report.

Additionally, Fortran is simply a very fast programming language. For example, take a simple algorithm that derives digits of Pi using arbitrary precision libraries built into the programming language. Fortran is the 5th fastest programming language to complete this task, taking around 0.74 seconds (Gouy, 2018). Figure 1 shows the times for C, C++, Python, and Intel Fortran to complete the Pi Digits Test.

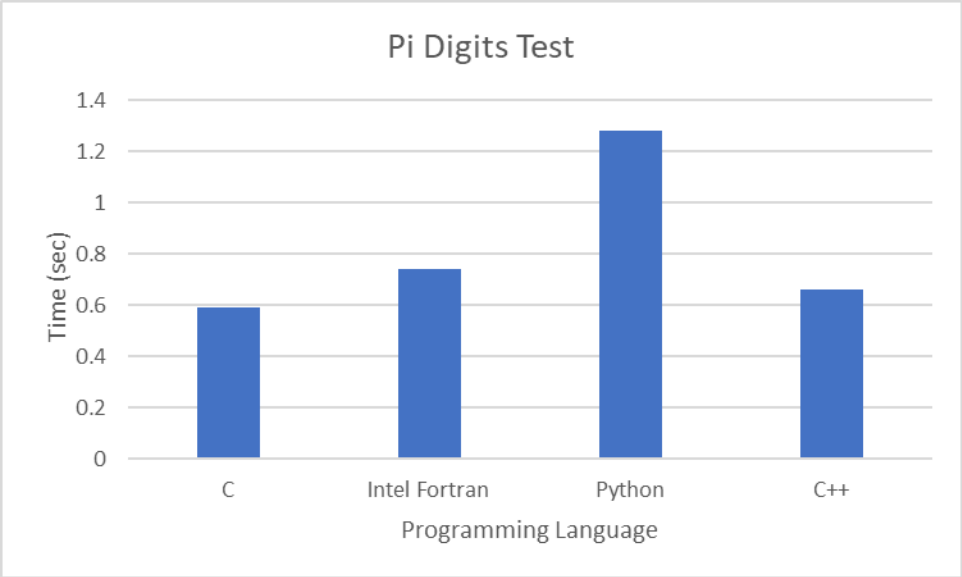

Figure 1: Time Elapsed to Complete Pi Digits Test

As can be seen, C and C++ are 10 – 20% faster than Fortran, however Python (the most popular programming language in the world as of Nov 2021 [7]) is approximately 73% slower than Fortran. Despite its age, Fortran is still amongst the fastest programming languages.

2.2. Coastal Interpolation Tool

The main coastal interpolation tool is a script dependent on 20 other Fortran files that perform various different tasks such as variable initialization, and algorithm processing. The tool can interpolate fields through different methods – bilinear interpolation, bicubic interpolation, and nearest neighbor interpolation. The goal of this interpolation tool is to be able to determine the value of a function of space (longitude and latitudes) and some other variable (e.g., temperature). If given a source grid of longitudes and latitudes and the corresponding temperature value for these coordinates, it will be possible to determine the temperature values for any point within the 2D space created by the coordinates. Temperature is not the only option, as it can also be used to determine other atmospheric conditions

2.2.1. Bilinear Interpolation

Bilinear interpolation is used for interpolation functions of two variables on a rectilinear grid (Wicklin, 2020). If the value of a function is known at four different points, it is possible to

determine the value of any other position within the grid. This is done by providing weights depending on how close the function is to the known points of the grid.

This can be seen through an example of the unit grid:

The unit grid has four corners:

- $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$
- At each of these points the temperature is known:
    - $T(0,0) = T_{00}$
    - $T(0,1) = T_{01}$
    - $T(1,0) = T_{10}$
    - $T(1,1) = T_{11}$

For any point within the original grid, the following formula can be used (Equation 1) [9]:

$$T(x,y) = [T_{00} * (1-x) * (1-y)] + [T_{10} * x * (1-y)] + [T_{01} * (1-x) * y] + [T_{11} * x * y]$$

Therefore, for point $(0.5, 0.5)$, the following formula can be used (Equation 2):

$$T(0.5{,}05) = [T_{00} * (1-0.5) * (1-0.5)] + [T_{10} * 0.5 * (1-0.5)] + [T_{01} * (1-0.5) * 0.5] + [T_{11} * 0.5 * 0.5]$$

The distribution of points for the unit grid can be seen below (Figure 2) [9]:

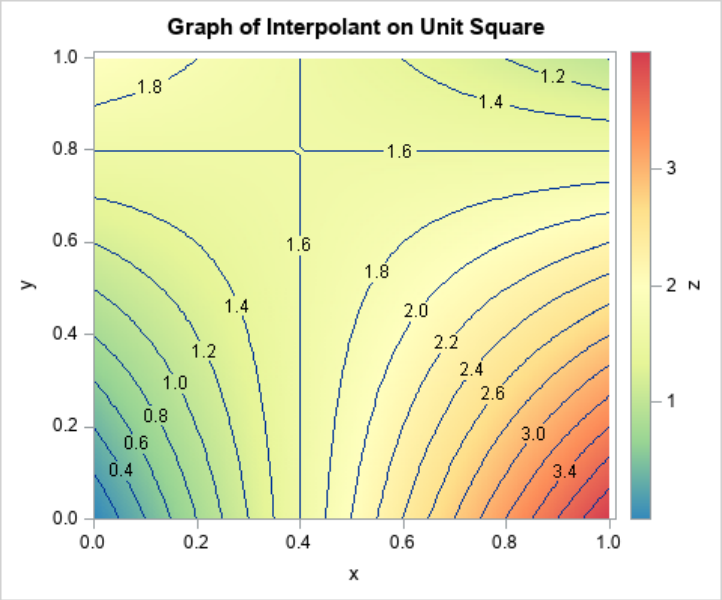

Figure 2: Contour Graph of Interpolated Points Using Bilinear Interpolation

2.2.2. Bicubic Interpolation

Bicubic interpolation can be used for interpolating values on a 2D regular grid (Cmglee, 2020). However, it is much more difficult to use as it requires more information than Bilinear interpolation does.

Say there is a function $f(x, y)$. In order to successfully use bicubic interpolation, the function values at each corner of the grid must be known, as well as their partial derivatives f_x, f_y, f_{xy}. At its simplest, the problem that must be solved is below (equation 3):

$$p(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j$$

Where the 16 coefficients that match $p(x, y)$ must be found.

2.2.3. Nearest Neighbor Interpolation

Nearest neighbor interpolation is the simplest interpolation method available in this tool. The algorithm essentially selects the closest known point to the unknown and assigns the value to it, creating a piecewise function.

Figure 3 summarizes each of these interpolation methods below (figure 3) [2]:

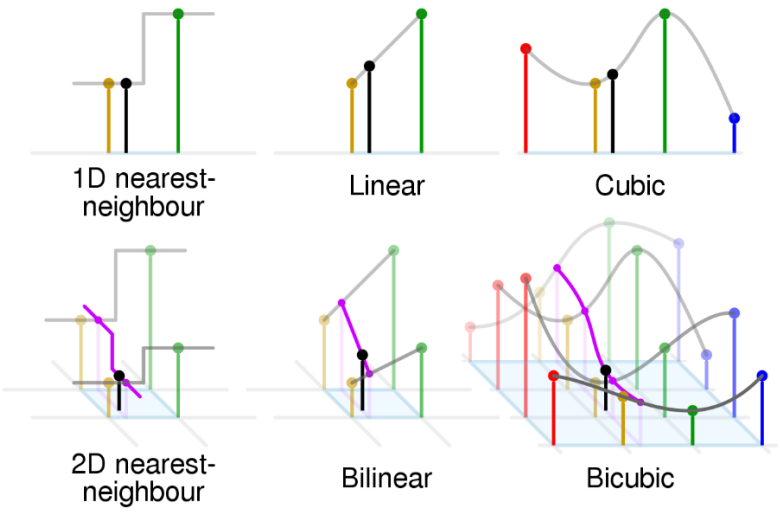

Figure 3: Summary of Interpolation Methods Used by Tool

2.3. Wrapping Fortran Code Using Cython

2.3.1. Fortran – Python workflow

In order to call Fortran code from Python, there are 2 major steps:

1. Create a C interface for the Fortran code
2. Create a Cython interface on the C interface

Cython was originally designed to help create C – Python interfaces. However, due to C being based on Fortran, there exists a built-in method to interface Fortran and C. By invoking the internal "iso_c_binding" module in Fortran (Sun Microsystems Inc., 2005), it is possible to simply call Fortran code from C. This effectively creates a C wrapper for the Fortran code. It then becomes possible to create the Python wrapper through Cython.

The coastal interpolation tool consists of 22 Fortran files that take care of various jobs, such as variable initialization, and manipulation, and performing functions. For the sake of this task, there is one major file, "cstintrp_py.f90" (Dupont, n.d.). Within this file, there exists a subroutine, "cstintrp_from_call" from which the interpolation tool is run. The subroutine takes the following inputs:

1. Source grids in the x, and y directions (longitudes and latitudes). These grids are 2D arrays, and must be the same size and shape. If the longitude array is a 4x4 array, than the latitude array must also be 4x4.
2. A final source grid that provides the variable that needs to be interpolated (i.e., temperature, atmospheric conditions).
3. Destination grids in the x, and y directions. These will be provided to interpolation functions later in the subroutine to determine the desired output variable values.

4. Lengths of each x and y grid.

By providing these inputs, the subroutine can then call other subroutines that exist in other files and provide the outputs. However, only code that interacts externally must be wrapped. Because these inputs are called from within “cstintrp_from_call”, their interactions occur internally, and therefore do not need to be wrapped.

Because the Fortran file must be able to be called from C, a header file must be created. A header file has the file extension “.h” and contains C function declarations for sharing between different file sources (Tutorials Point, n.d.).

After creating a header file, it then becomes possible to call the function from C, and interface it with Cython. Cython files have the extension “.pyx” when written in a Unix environment, and”.pyd” when written in a Windows environment.

Writing Cython files is very similar to writing a Python module to be used from within other files. The goal of the Cython file is to create a Python function that calls the Fortran subroutine from within the C header file. This is only possible because of Cython’s ability to interface with C code.

After creating a Cython file, it is then required to create a setup file that compiles and links the Cython, C, and Fortran code into a shared library, which can then be imported from within Python, and used.

3. Materials and Methods

3.1. Required Software

- Preferred text editor (visual studio code, notepad ++, etc.)
- Fortran Compiler (Intel Fortran Compiler used, gcc is acceptable)
- C compiler (GCC was used)
- Python 3.6.9 or earlier

3.2. Methods

3.2.1. Analyzing Performance of Cython, Ctypes, and f2py

A Fortran90 subroutine is developed to perform some simple array operations. A 2 x 2 matrix of floating-point values from between 1 and 100 will be initialized. The subroutine will then calculate the determinant of the matrix. The time it takes for this program to be run will be calculated and recorded.

The subroutine will then be wrapped using Ctypes, Cython, and F2PY. The time it takes for these programs to run will be calculated. The fastest time will indicate the best program in terms of

speed. To ensure accurate results, the test will be run 3 times for each wrapper and compared to the speed of the original Fortran subroutine.

3.2.2. Analyzing Performance of Wrapped Coastal Interpolation Tool

The time it takes for the interpolation tool to run in pure Fortran mode will be measured and recorded. The same inputs will then be provided to the Python wrapped tool, and the time it takes to run will be measured and recorded. The test will be repeated 3 times with the same inputs to ensure accuracy.

4. Results and Analysis

Currently, the interpolation tool is not user friendly at all. To combat this, it was decided that developing a Python interface that limited end-users' interaction with the back-end environment would provide this. However, this interface should not detract from the tool in any way. Whatever the tool can do in the pure Fortran version should be completely possible using the interface.

Also, while there are concessions being made for possible performance reductions, the code can not be significantly slower than the original code.

If the completed interface improves end-user experiences, with limited processing slowdowns, and room for future improvements, then the interface is viable and successful.

4.1. Comparison of Ctypes, Cython, and F2PY

As a result of wanting a Python interface for the Fortran code, it limited the options available to complete this task. Of these options, three stood out – Ctypes, Cython, and F2PY.

Ctypes is a python library that provides C data types that are compatible with Python and wraps C and Fortran code for use in Python. It is a simple library to use, as python does all the wrapping for the user [5].

However, there is limited opportunities for optimizations when the compiler has to continuously enter C or Fortran libraries. This can lead to a bottleneck due to being forced to use only Python. Additionally, there is limited support for user defined data types.

F2PY is a Python library specifically for creating a Fortran – Python interface. The library allows for more finite control of the interface. However, the library is only available for the Python 2 language, which is unacceptable.

The final wrapping technique that was considered was Cython. Cython is technically an entirely separate programming language from C/Fortran and Python. However, because of this it allows

for the use of both Python and C techniques. Code can be written to be identical to C, or it can be written to be Python code.

Before writing any code and testing the differences between Ctypes, Cython, and F2PY, it was important to determine if they had the desired feature set. Tables 1 – 3 (below) detail the pros and cons of each wrapping method.

Cython	
Pros	**Cons**
Can use both Python and C memory management	Is a different language from Python and Fortran - Will be a learning curve
Can optimize code for speed if performance is an issue	
Flexibility to write code in Fortran to call in Cython if need be later	

Table 1: Pros and Cons of Cython

Ctypes	
Pros	**Cons**
Built-in Python library - Completely written in Python	Limited optimization opportunities when calling into multiple libraries
No Fortran or C code to compile	Limited derived type support
Very fast for quickly wrapping code	

Table 2: Pros and Cons of Ctypes

F2PY	
Pros	**Cons**
Allows for more finite control of wrapper - However still not as much control as Cython	Written in Python 2, and hasn't been updated in long time

Table 3: Pros and Cons of F2PY

Because F2PY was written for Python 2 (which has not been used since Python 3 released in 2008), and its lack of support means it is not viable.

While Ctypes seems like a viable option, there are over 20 separate modules used, and multiple calls into them. This could possibly lead to severe performance deficits that were determined to be not worthy of the risk.

Also, there are dozens of user defined data types within the Fortran modules. To call the interpolation tool from Python, these user defined types must be ported as well. Since Ctypes does not provide support for user defined types, it is not a viable option.

Cython, on the other hand provides support for all user defined types, and the majority of Fortran and C's feature set. Because of this, it was determined that Cython would be the best option.

4.2. Performance of Python Interface Compared to Original

As of this moment, the Python interface is not actually completed and ready for deployment. Because of this, no data can be provided to compare the performance of both software solutions. However, it is expected that the Python interface will perform worse than the pure Fortran code. This is due to the intermediate steps required by calling the Fortran file from within a C and Cython file. However, because there are not many operations occurring outside of the Fortran subroutine, the decrease in performance is not expected to be large.

5. Conclusions and Recommendations

For the finest control of any Python interface for C and Fortran code, Cython is the best choice due to its ability to write code as similar to C or Python as the user desires. With its support for the majority of Fortran and C's feature list, it is the best option for creating Python interfaces for C and Fortran code.

While the use of Python to limit end-users' interaction with unnecessary back-end functions will help improve ease-of-use, steps can be taken to further improve this. By creating a Python interface, it becomes possible to further leverage Python's many libraries. Using a GUI development library such as tkinter, the creation of a GUI to further simplify code execution would continue to improve ease-of-use.

While the performance analysis of the interface compared to the original code is not yet complete, the expected results imply that using a Python interface should not drastically increase function run-time. However, as more and more code for the tool is written in Python, it is possible to introduce performance bottlenecks due to the interpretive nature of Python. For major code overhauls, it is important to develop that in Fortran, or another low-level, declarative coding language where it would be easier to perform memory management.

6. References

1. Python Software Foundation. (n.d.). *Welcome to Cython's documentation*. Cython 3.0.0a9 documentation. Retrieved from https://cython.readthedocs.io/en/latest/

2. Cmglee. (n.d.). *Comparison of 1D and 2D interpolation.svg*. Wikimedia Commons. Retrieved from https://commons.wikimedia.org/w/index.php?curid=53064904.

3. Gouy, I. (2018). *Pidigits*. pidigits | Computer Language Benchmarks. Retrieved from https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits.html

4. NumPy Foundation. (n.d.). *F2PY users guide and reference manual*. F2PY Users Guide and Reference Manual - NumPy v1.21 Manual. Retrieved from https://numpy.org/doc/stable/f2py/

5. Python Software Foundation. (n.d.). *Ctypes - a foreign function library for python¶*. ctypes - A foreign function library for Python - Python 3.10.0 documentation. Retrieved from https://docs.python.org/3/library/ctypes.html

6. Sun Microsystems. (2005). *C H A P T E R 11 - C-fortran interface*. C. Retrieved from https://docs.oracle.com/cd/E19422-01/819-3685/11_cfort.html

7. Tiobe Software BV. (2021). *Tiobe Index* . TIOBE. Retrieved from https://www.tiobe.com/tiobe-index/

8. Tutorials Point. (n.d.). *Header files*. C Tutorials. Retrieved from https://www.tutorialspoint.com/cprogramming/c_header_files.htm

9. Wicklin, R. (2020, May 18). *What is bilinear interpolation?* The DO Loop. Retrieved from https://blogs.sas.com/content/iml/2020/05/18/what-is-bilinear-interpolation.html

7. Appendix

7.1. cstintrp_py.f90

```
module cstintrp_py


  ! module declaration
    USE iso_c_binding
    USE cstintrp_mod !
    USE intrp_itfc!
    USE utils
    USE mympp
    USE std
    USE grids!
    USE llxy!
    USE intrp_oce

    IMPLICIT NONE

    PUBLIC

!! Argument variables:

    CONTAINS




    subroutine cstintrp_from_call(xsrc,ysrc,fsrc,xdst,ydst,fdst, nx_src0,ny_src0,
nx_dst0, ny_dst0) bind(C, name="cstintrp_from_call_c")

    !-------------------------------------------------------------------------
--
    ! main call for treating interpolation between src and dst grids`
    !-------------------------------------------------------------------------
--
    ! arguments
    integer(c_int) :: nx_src0, ny_src0, nx_dst0, ny_dst0
    real(c_double), dimension(nx_src0,ny_src0) :: xsrc,ysrc,fsrc
    real(c_double), dimension(nx_dst0,ny_dst0) :: xdst,ydst,fdst
    type(type_grid), target &
                        :: F_grds, F_grdr
    ! locals
    type(type_data), target  :: psrc, pdst
    type(std_grid),  pointer :: grdp
```

```
    type(type_grid), pointer :: subg, subs, subr
    type(type_location), target :: plocr, ploci
    type(type_weights), target  :: pintr, psprd
    integer nx, ny
    logical vector
    type(variable)      :: var_grds

    integer :: ngsrc = 1, ngdst = 1

    call init_general_param

    intyp = 'bilin'

    !-------------------------------------------------------------------------
-----------
    call read_vector_table

    ! Initialize parameters and validate options with source and destination grid
    write(*,*) 'XSRC SHAPE AND SIZE'

    nx = nx_src0
    ny = ny_src0

    print *, nx, ny
    grdp => F_grds % std_grd
    grdp % ni = nx
    grdp % nj = ny
    grdp % grtyp = 'O' ! by default assume the grid of being irregular in type
    F_grds % dxtree  = dxred
    F_grds % dytree  = dyred
    F_grds % nbdy    = snbdy
    F_grds % nrepeat = snrep
    F_grds % webdy   = sebdy
    F_grds % werepeat= serep
    F_grds % nx      = nx
    F_grds % ny      = ny
    F_grds % ng      = 1
    allocate( F_grds % subgrd(1) )
    subg => F_grds % subgrd(1)
    subg % std_grd = grdp
    subg % nx      = F_grds % nx
    subg % ny      = F_grds % ny
    subg % dxtree  = F_grds % dxtree
    subg % dytree  = F_grds % dytree
    subg % nbdy    = F_grds % nbdy
```

```
    subg % nrepeat = F_grds % nrepeat
    subg % webdy   = F_grds % webdy
    subg % werepeat= F_grds % werepeat
    allocate( subg % lld % latt(nx,ny), &
              subg % lld % lont(nx,ny) )
    subg % lld % lont(:,:) = xsrc(:,:)
    subg % lld % latt(:,:) = ysrc(:,:)


    nx = nx_dst0
    ny = ny_dst0
    grdp => F_grdr % std_grd
    grdp % ni = nx
    grdp % nj = ny
    grdp % grtyp = 'O' ! by default assume the grid of being irregular in type
    F_grdr % dxtree  = dxred
    F_grdr % dytree  = dyred
    F_grdr % nbdy    = dnbdy
    F_grdr % nrepeat = dnrep
    F_grdr % webdy   = debdy
    F_grdr % werepeat= derep
    F_grdr % nx      = grdp % ni
    F_grdr % ny      = grdp % nj
    F_grdr % ng      = 1
    allocate( F_grdr % subgrd(1) )
    subg => F_grdr % subgrd(1)
    subg % std_grd = grdp
    subg % nx      = F_grdr % nx
    subg % ny      = F_grdr % ny
    subg % dxtree  = F_grdr % dxtree
    subg % dytree  = F_grdr % dytree
    subg % nbdy    = F_grdr % nbdy
    subg % nrepeat = F_grdr % nrepeat
    subg % webdy   = F_grdr % webdy
    subg % werepeat= F_grdr % werepeat
    allocate( subg % lld % latt(nx,ny), &
              subg % lld % lont(nx,ny) )
    subg % lld % lont(:,:) = xdst(:,:)
    subg % lld % latt(:,:) = ydst(:,:)

    !-------------------------------------------------------------------------
-----------
    ! find the variables associated with that particular source grid
    var_grds % varstd % nomvar = 'NAME'
    var_grds % varstd % typvar ='P'
```

```
    var_grds % varstd % ip1 = 0
    var_grds % varstd % npas = 0
    var_grds % ln_mask = .false.
    var_grds % ln_vector = .false.



    call intrp_itfc_params( F_grds, F_grdr )



    ! Allocate memory and copy grid descriptors to destination file
    nxsrc = F_grds % nx
    nysrc = F_grds % ny
    nxdst = F_grdr % nx
    nydst = F_grdr % ny
    if ( F_grds % std_grd % grtyp == 'U' ) nysrc = nysrc / 2
    if ( F_grdr % std_grd % grtyp == 'U' ) nydst = nydst / 2

    allocate(Msrc(nxsrc,nysrc,1))
    allocate(Mdstw(nxdst,nydst,1,1))

    !--------------Added Lines of Code ------------------------------------

    ! call allocate_dst
    ! call allocate_src
    if (allocated(subgsrc)) deallocate(subgsrc, subgdst)
    allocate( subgsrc(ngsrc), subgdst(ngdst) )
    ! if (allocated(intr)) deallocate(intr)
    ! allocate( intr(ngdst,ngsrc) )
    ! if (allocated(locr)) deallocate(locr)
    ! allocate( locr(ngdst,ngsrc) )
    ! if (allocated(locr)) deallocate(locr)
    ! allocate( locr(ngdst,ngsrc) )
    ! if (allocated(loci)) deallocate(loci)
    ! allocate( loci(ngsrc,ngdst) )

    !-----------End of Added Lines of Code --------------------

      subs => F_grds % subgrd(1)
      subr => F_grdr % subgrd(1)
      subgsrc( 1 ) = subs % std_grd
      subgdst( 1 ) = subr % std_grd
      ! plocr => locr(1,1)
      ! ploci => loci(1,1)
```

```
! Calcul des xy de la grille source sur la destination

      ! subs => F_grds % subgrd(1)
      ! subr => F_grdr % subgrd(1)
      subgsrc(1) = subs % std_grd !Error here
      subgdst(1) = subr % std_grd

      ! do the localization of the destination points on the source grid

      if( subs % std_grd % grtyp /= 'Y' .and. &
          (intyp /= "aggr" .and. intyp(1:5) /= "scrip") ) then
        write(*,*) 'CSTINTRP: CALCUL SUR LA GRILLE DESTINATION DES XY DU REPERE
SOURCE'


        call ll2xy( subs, subr, plocr )

             ! FD debug
             write(*,*) 'forward localization'
             write(*,*) 'min/max I',minval( plocr % ig(:,:) ),maxval( plocr %
ig(:,:) )
             write(*,*) 'min/max J',minval( plocr % jg(:,:) ),maxval( plocr %
jg(:,:) )
             write(*,*) 'min/max A',minval( plocr % ag(:,:) ),maxval( plocr %
ag(:,:) )
             write(*,*) 'min/max B',minval( plocr % bg(:,:) ),maxval( plocr %
bg(:,:) )
             write(*,*) 'min/max a',minval( plocr % pg(:,:) ),maxval( plocr %
pg(:,:) )
      endif

      ! do reverse localization used in aggregation
      select case(intyp)
      case( 'aggr', 'mixt', 'mixtbc')
        write(*,*) 'CSTINTRP: CALCUL SUR LA GRILLE SOURCE DES XY DU REPERE
DESTINATION'
        call ll2xy( subr, subs, ploci )
             ! FD debug
             write(*,*) 'backward localization'
             write(*,*) 'min/max I',minval( ploci % ig(:,:) ),maxval( ploci %
ig(:,:) )
             write(*,*) 'min/max J',minval( ploci % jg(:,:) ),maxval( ploci %
jg(:,:) )
```

```
            write(*,*) 'min/max A',minval( ploci % ag(:,:) ),maxval( ploci %
ag(:,:) )
            write(*,*) 'min/max B',minval( ploci % bg(:,:) ),maxval( ploci %
bg(:,:) )
            write(*,*) 'min/max a',minval( ploci % pg(:,:) ),maxval( ploci %
pg(:,:) )
      end select

        write(*,*) 'CSTINTRP: FIRST FIELD ... FULL COMPUTATION'

        select case(intyp)
        case('bicub','mixtbc')
          write(*,*) 'CSTINTRP: BI-CUBIC INTERPOLATION'
          nwgtb=12
        case('bilin','mixt')
          write(*,*) 'CSTINTRP: BI-LINEAR INTERPOLATION'
          nwgtb=4
        case('nearst')
          write(*,*) 'CSTINTRP: NEARST INTERPOLATION'
          nwgtb=1
        end select

        call allocate_weights(.true., pintr)

        if ( subs % std_grd % grtyp == 'M') then ! unstructured grids
          SELECT CASE ( intyp )
          CASE('nearst')

            call intrp_itfc_tri(plocr, pintr, subs, Msrc(:,:,1), nearst=.true.)
          CASE('bilin')

            call intrp_itfc_tri(plocr, pintr, subs, Msrc(:,:,1) )
          CASE('aggr') ! aggr

            call intrp_itfc_aggr(plocr, ploci, pintr, subs, subr, &
Msrc(:,:,1) )
          CASE('scrip0')
            call intrp_itfc_scrip(plocr, ploci, pintr, subs, subr, &
Msrc(:,:,1), .false.)
          END SELECT



        else ! regular grids
          write(*,*) '---------------Regular grids---------------'
```

```
          SELECT CASE ( intyp )
          CASE('nearst')
            write(*,*) '---------------nearst---------------'
            call intrp_itfc_nearst(plocr, pintr, psprd, Msrc(:,:,1))
          CASE('bilin','bicub') ! bilin (and possibly nearest) or bicub
            write(*,*) '---------------bilin or bicub---------------'
            call intrp_itfc_bilc(plocr, pintr, psprd, Msrc(:,:,1))
          CASE('aggr') ! aggr
            write(*,*) '---------------aggr---------------'
            call intrp_itfc_aggr(plocr, ploci, pintr, subs, subr, &
Msrc(:,:,1))
          CASE('mixt','mixtbc') ! mixt or mixtbc
            write(*,*) '---------------mixt or mixtbc---------------'
            call intrp_itfc_mixtlc(plocr, ploci, pintr, psprd, subs, &
subr, Msrc(:,:,1))
          CASE('scrip0','scrip1')
            write(*,*) '---------------scrip0 or scrip1---------------'
            call intrp_itfc_scrip(plocr, ploci, pintr, subs, subr, &
Msrc(:,:,1), intyp=='scrip1')
          END SELECT
        endif

        call adjust_periodic(subs, subr, pintr )

        write(*,*) '---------------adjust_periodic complete---------------'

             ! FD debug
             write(*,*) 'weights', pintr % numwgt
             write(*,*) 'min/max W',minval( pintr %  wgt ),maxval( pintr %  wgt )
             write(*,*) 'min/max I',minval( pintr % iwgt ),maxval( pintr % iwgt )
             write(*,*) 'min/max J',minval( pintr % jwgt ),maxval( pintr % jwgt )
             write(*,*) 'min/max N',minval( pintr % nwgt ),maxval( pintr % nwgt )

! Initialize working mask
        Mdstw(:,:,1,1) = 0

        print *, pintr % mwgt

        where( pintr % mwgt(:,:) ) Mdstw(:,:,1,1) = 1 ! ... Mdstw is the one
written
        write(*,*)'---------------working mask initialized---------------'
!...  Application des poids
      allocate(psrc % u(nxsrc,nysrc)); psrc % u(:,:) = fsrc(:,:)
      allocate(pdst % u(nxdst,nydst))
```

```
      write(*,*) '---------------allocation complete---------------'

      call apply_weights( pdst % u, psrc % u, pintr)
             ! FD debug
             write(*,*) 'min/max F',minval(pdst % u ),maxval(pdst % u )
      fdst(:,:) = pdst % u (:,:)

    end subroutine cstintrp_from_call

end module cstintrp_py
```

7.2. cstintrp_py.h

```
extern void get_default(options *option)

extern void cstintrp_from_call_c(double *xsrc, double *ysrc, double *fsrc, double
*xdst, double *ydst, double *fdst, int *nx_src, int *ny_src, int *nx_dst, int
*ny_dst);
```

7.3. cstintrp_py.pyx

```
cimport numpy as cnp
import numpy as np


cdef extern from "cstintrp_py.h":
    cdef void get_default(options *option)
    cdef void cstintrp_from_call_c(double *xsrc, double *ysrc, double *fsrc,
double *xdst, double *ydst, double *fdst, int *nx_src, int *ny_src, int *nx_dst,
int *ny_dst)

def cstintrp_from_call_py(cnp.ndarray[cnp.double_t, ndim=2, mode = 'c'] xsrc,
                          cnp.ndarray[cnp.double_t, ndim=2, mode = 'c']  ysrc,
                          cnp.ndarray[cnp.double_t, ndim=2, mode = 'c']  fsrc,
                          cnp.ndarray[cnp.double_t, ndim=2, mode = 'c'] xdst,
                          cnp.ndarray[cnp.double_t, ndim=2, mode = 'c'] ydst):
    # cdef double fdst[len(xdst)][len(xdst[0])]
    # fdst = np.empty_like(xdst)
    # cdef np.ndarray[np.double_t, ndim=2, mode = 'c'] xsrc, ysrc, fsrc, xdst,
ydst


    xsrc_shape = xsrc.shape
    xdst_shape = xdst.shape

    nx_src = xsrc_shape[0]
    ny_src = xsrc_shape[1]
```

```
    nx_dst = xdst_shape[0]
    ny_dst = xdst_shape[1]

    cdef cnp.ndarray[cnp.double_t, ndim=2, mode = 'c'] fdst = np.zeros_like(xdst)



    cstintrp_from_call_c(<double*> xsrc.data, <double*> ysrc.data, <double*>
fsrc.data, <double*> xdst.data, <double*> ydst.data, <double*> fdst.data, <int*>
&nx_src, <int*> &ny_src, <int*> &nx_dst, <int*> &ny_dst )
```