Tutorial: Easy Python and Fortran with f2py

Viktor K. Decyk
UCLA

Python

Python is an interpreted, dynamic language, now widespread in scientific programming
- It supports both modular and object-oriented programming
- Has a huge variety of module and libraries available
- Useful for rapid programming

Python's interpreter is slow
- Interpreter and run time typing system have high overhead
- Interpreter can only run one script at a time (no parallel processing)

Python is very useful as a "glue" language
- Many modules, written in low level languages, provide high performance
- Python provides modules such as graphics are not available in Fortran

Numpy is a module designed for high performance array processing
- Much of it is written in C
- Processing entire arrays has less overhead than processing for loops
- Numpy array processing functions similar to Fortran90 array intrinsics

Numpy:f2py

f2py is a numpy tool to create dynamic libraries for Python from Fortran or C sources
- Allows you to create your own high performance functions in Python
- Useful when numpy array processing insufficient

f2py can automatically translate some Fortran90 interfaces to Python functions
- Requires intent attributes
- Arrays are all assumed-shape arrays
- Does not have any derived types

If the Fortran uses derived types, wrappers have to be written

This tutorial will illustrate its use in a primitive PIC code.
- Use parts of a Fortran PIC code in a Python program

Fortran PIC: Initializing particles

```
      program f2pydemo1
      use init1
      implicit none
!
! Inputs:
! indx = exponent which determines grid points in x direction: nx = 2**indx.
      integer, parameter :: indx =   9
! npx = number of electrons distributed in x direction.
      integer, parameter :: npx = 18432
! vtx = thermal velocity of electrons in x direction
! vx0 = drift velocity of electrons in x direction.
      real, parameter :: vtx = 1.0, vx0 = 0.0
! idimp = number of particle coordinates = 2
      integer, parameter :: idimp = 2
!
! declare scalars for standard code
      integer :: np, nx, n
!
! declare arrays for standard code:
! part = particle array
      real, dimension(:,:), allocatable :: part
!
! initialize scalars for standard code
! nx = number of grid points in x direction
! np = total number of particles in simulation
      nx = 2**indx; np = npx;
!
! allocate data for standard code
      allocate(part(idimp,np))
!
! initialize uniform plasma and maxwellian velocity: updates part
      call DISTR1(part,vtx,vx0,npx,nx,0)
```

Fortran PIC: Initializing particles

We want to allow Python to call the initialization function DISTR1 in initlib1.f90.
This function is inside a module init1 and has the interface:

```
interface
   subroutine DISTR1(part,vtx,vdx,npx,nx,ipbc)
   implicit none
   integer, intent(in) :: npx, nx, ipbc
   real, intent(in) :: vtx, vdx
   real, dimension(:,:), intent(inout) :: part
end interface
```

This interface has all the 3 attributes needed for automatic generation of python interfaces
Assuming no dependencies, the dynamic library plasmalib can be generated by executing:

```
f2py --fcompiler=gnu95 -m plasmalib -c initlib1.f90
```

Python PIC: Initializing particles

```
import math
import numpy

from plasmalib import *

float_type = numpy.float32

def main():
# Inputs:
# indx = exponent which determines grid points in x direction: nx = 2**indx.
   indx =   9
# npx = number of electrons distributed in x direction.
   npx = 18432
# vtx = thermal velocity of electrons in x direction
# vx0 = drift velocity of electrons in x direction.
   vtx = 1.0; vx0 = 0.0
# idimp = number of particle coordinates = 2
   idimp = 2

# initialize scalars for standard code
# nx = number of grid points in x direction
# np = total number of particles in simulation
   nx = int(math.pow(2,indx)); np = npx;

# allocate data for standard code
# part = particle array
   part = numpy.empty((idimp,np),float_type,'F')

# initialize uniform plasma and maxwellian velocity: updates part
   init1.distr1(part,vtx,vx0,npx,nx,0)
```

Fortran PIC: Pushing free-streaming particles

- push1zf advances particle without forces

```
      program f2pydemo1
      use init1
      implicit none
!
! Inputs:
      ......
! nloop = number of time steps in the simulation
      integer, parameter :: nloop = 100
! dt = time interval between successive calculations.
      real, parameter :: dt = 0.1
! wke = particle kinetic energy
      real :: wke = 0.0
!
! * * * start main iteration loop * * *
!
      do n = 1, nloop
!
! push free-streaming particles: updates part, wke
         wke = 0.0
         call push1zf(part,dt,wke,idimp,np,nx)
!
      enddo
!
! * * * end main iteration loop * * *
```

Fortran PIC: Pushing free-streaming particles

We want to allow Python to call the push function PUSH1ZF in pushlib1.f.
This function has the interface:

```
interface
   subroutine PUSH1ZF(part,dt,ek,idimp,nop,nx)
   implicit none
   integer, intent(in) ::  nop, idimp, nx
   real, intent(in) :: dt
   real, intent(inout) :: ek
   real, dimension(idimp,nop), intent(inout) :: part
   end subroutine
end interface
```

This interface does NOT has all the 3 attributes needed for automatic generation of python interfaces

- The part array is an explicit-shape array
- The procedure is not in a module

Assuming one does not want to change the original code, one needs to write a wrapper function

- The original source code may not be available or written in another language.

Fortran PIC: Pushing free-streaming particles

- Create a wrapper function wpush1zf in module wpush1

```
!
      module wpush1
      implicit none
!
      contains
!
      subroutine wpush1zf(part,dt,ek,nx)
! push free-streaming particles
      implicit none
      integer, intent(in) :: nx
      real, intent(in) :: dt
      real, intent(inout) :: ek
      real, dimension(:,:), intent(inout) :: part
! local data
      integer :: idimp, nop
! extract dimensions
      idimp = size(part,1); nop = size(part,2)
! call low level procedure
      call PUSH1ZF(part,dt,ek,idimp,nop,nx)
      end subroutine
!
      end module
```

The function wpush1zf now has all the 3 attributes needed for automatic generation of python interfaces
Procedures not exported need to be compiled with -fPIC option to generate Position Independent Code
The dynamic library plasmalib can be generated by executing:

```
gfortran -fPIC -c pushlib1.f
f2py --fcompiler=gnu95 -m plasmalib -c initlib1.f90 pushmod1.f90 pushlib1.o
```

Python PIC: Pushing Free-streaming particles

```
import math
import numpy

from plasmalib import *

float_type = numpy.float32

def main():
# Inputs:
   ...
# nloop = number of time steps in the simulation
   nloop = 100
# dt = time interval between successive calculations.
   dt = 0.1
# wke = particle kinetic energy
   wke = numpy.zeros((1),float_type)

# * * * start main iteration loop * * *

   for n in range(0,nloop):

# push free-streaming particles: updates part, wke
      wke[:] = 0.0
      wpush1.wpush1zf(part,dt,wke,nx)

# * * * end main iteration loop * * *

if (__name__=="__main__"):
   main()
```

Note: A scalar (wke) which is returned by a Fortran function, is represented as a numpy array of length 1

Fortran PIC: Pushing relativistic free-streaming particles

- Consider an example of a pusher rpush1zf which contains a derived type part1d

```
      program f2pydemo2
      use init1
      use rpush1
      implicit none
!
! Inputs:
      ......
! ci = reciprocal of velocity of light.
      real, parameter :: ci = 0.1
!
! rpart = helper object for relativistic particles
      type (part1d) :: rpart
!
! construct helper object for electrons: update rpart
      call new_part1d(rpart,0.0,0.0,idimp,np)
!
! * * * start main iteration loop * * *
!
      do n = 1, nloop
!
! push free-streaming particles: updates part, wke
        wke = 0.0
        call rpush1zf(rpart,part,dt,ci,wke,nx)
!
      enddo
!
! * * * end main iteration loop * * *
```

Fortran PIC: Pushing relativistic free-streaming particles

module rpush1 in file rpushlib1.f90 contains the definition of part1d

```
      type part1d
         real :: qm, qbm         ! qm, qbm = charge,  charge/mass ratio  of particle
         integer :: idimp, nop   ! idimp = size of phase space, nop = number of particles in array
      end type
```

as well as the function rpush1zf whose interface is:

```
interface
   subroutine rpush1zf(this,part,dt,ci,ek,nx)
   type (part1d), intent(in) :: this
   integer, intent(in) :: nx
   real, intent(in) :: ci, dt
   real, intent(inout) :: ek
   real, dimension(:,:), intent(inout) :: part
end interface
```

This interface does NOT have all the 3 attributes needed for automatic generation of python interfaces

- An argument to the function is a derived type, which f2py does not support

Assuming one does not want to change the original code, one needs to write a wrapper function

There are two possible approaches in not exposing derived types to Python:

(1): Unpack elements of the derived type and replace its elements in the wrapper

(2): Hide the derived type inside the Fortran but allow Python functions to set its value

Fortran PIC: Pushing relativistic free-streaming particles, using approach 2

- Create a module wrpush1 in rpushmod1.f90 which contains a private variable wpart of the derived type
- Create a function set_part1d which sets the values of that private variable
- Create a wrapper w2rpush1zf which uses the private variable rather than an argument of the derived type

```
      module wrpush1
       use rpush1
       implicit none
!
       type (part1d), save, private :: wpart
!
       contains
!
       subroutine set_part1d(qm,qbm,idimp,nop)
! this subroutine set a private descriptor for 1d particles
! wpart = part1d descriptor of particle data
       implicit none
       real, intent(in) :: qm, qbm
       integer, intent(in) :: idimp, nop
! set descriptor
       wpart%qm = qm; wpart%qbm = qbm
       wpart%idimp = idimp; wpart%nop = nop
       end subroutine
!
       subroutine w2rpush1zf(part,dt,ci,ek,nx)
! push free-streaming relativistic particles, using wpart
       integer, intent(in) :: nx
       real, intent(in) :: dt, ci
       real, intent(inout) :: ek
       real, dimension(:,:), intent(inout) :: part
! call low level procedure
       call rpush1zf(wpart,part,dt,ci,ek,nx)
       end subroutine
!
       end module
```

Fortran PIC: Pushing free-streaming particles

The functions set_part1d and w2rpush1zf now have all the 3 attributes needed for automatic generation of python interfaces
The dynamic library rplasmalib can be generated by executing:

```
gfortran -fPIC -c rpushlib1.f90
f2py --fcompiler=gnu95 -m rplasmalib -c initlib1.f90 rpushmod1.f90 rpushlib1.o
```

Note:

- The original file rpushlib1.f90 which defines the derived type is not compiled by f2py.
- The wrapper file rpushmod1.f90 does not expose the derived type

Wrappers are also a good place to add some error checking or polymorphism

Python PIC: Pushing relativistic free-streaming particles

- Consider an example of a pusher w2rpush1zf which makes use of derived type part1d

```
import math
import numpy

from rplasmalib import *

float_type = numpy.float32

def main():
# Inputs:
      ......
# ci = reciprocal of velocity of light.
   ci = 0.1

# set hidden Fortran helper object for relativistic particles
   wrpush1.set_part1d(0.0,0.0,idimp,np)

# * * * start main iteration loop * * *

   for n in range(0,nloop):

# push free-streaming relativstic particles: updates part, wke
      wke[:] = 0.0
# using hidden Fortran helper object
      wrpush1.w2rpush1zf(part,dt,ci,wke,nx)

# * * * end main iteration loop * * *

if (__name__=="__main__"):
   main()
```

Conclusions

UPIC Framework codes can all generate dynamic libraries for Python and have Python scripts

- without graphics, the Fortran and Python codes run at the same speed
- Fortran dynamic libraries compiled with OpenMP

A 1d UPIC code with a GUI Window manager runs interactively

Python is written in C and it is possible for Fortran to call Python

Python can "glue" together multiple codes for multiscale supercodes or workflows

f90wrap is alternative tool which supports derived types, which I have not tried