Interfacing Python with C and Fortran

Dr. Axel Kohlmeyer

Associate Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

Recap: dynamic linking via dlopen()

- POSIX compliant C libraries allow loading of shared objects are runtime via dlopen()/dlsym()
  - Calls to dlopen() open a handle to shared object; lookup of this file is subject to same rules as dynamic library searches
  - Calls to dlsym() look up symbol by its name in shared object pointed to by handle; returns pointer; for functions need to cast/assign to function pointer
  - Calls to dlclose() unload shared object (if last user) and revoke assignments to code made by dlsym()

Example: main program test-1.c

```
#include <dlfcn.h>

int main(int argc, char **argv)
{
    void *handle;      /* handle for dynamic object */
    void (*hi)();      /* function pointer for symbol */

    handle = dlopen("./hello.so", RTLD_LAZY);
    if (handle) {
       hi = (void (*)()) dlsym(handle,"hello");
       (*hi)();
       dlclose(handle);
    }
    return 0;
}
/* compile with: gcc -o test-1 -Wall -O test-1.c -ldl
 add -rdynamic if shared object needs symbols in main */
```

Example: shared object hello.c

```
#include<stdio.h>

void hello(void)
{
    puts("Hello, World!");
}
/*
 compile: gcc -shared -o hello.so -fPIC -Wall -O hello.c
 */
```

- With this setup, `hello.c` can be changed and `hello.so` recompiled without having to recompile and re-link `test-1`.
- Thus access to `test-1.c` is not needed.

Extending Python with `ctypes`

- The `ctypes` module in python provides an interface to dlopen()/dlsym() and thus allows to call compiled C code from python.
- Support for dll files on Windows is also included
- Since symbols in compiled objects have no information about calling sequence and return values, this has to be set on the python side
- Incorrect use can lead to segmentation faults or corrupted data; often prototypes are needed

Example: calling hello.c from python

```
#!/usr/bin/env python

from ctypes import *

# import shared object on POSIX compatible OS
dso = CDLL(“./hello.so”)

# call symbol in shared object as function w/o args
dso.hello()
```

- This python script does pretty much the same thing as the `test-1` compiled program
- Since there are no arguments and no return values, no code needs to know about the other

Arguments & Return Value

- By default ctypes will assume arguments and return values are standard size integer

```
#include<stdio.h>

int sum_of_int(int a, int b) {
    int c = a + b;
    printf("sum of %d and %d is %d\n",a,b,c);
    return c;
}
```

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
isum = dso.sum_of_int(1,2)
print "Integer sum is: ", isum
```

Prototypes with ctypes

- If argument and/or return value are of different type, ctypes needs to be informed about it; works similar to prototypes in C

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
dso.sum_of_int.argtypes = [ c_int, c_int ]
dso.sum_of_int.restype = c_int
isum = dso.sum_of_int(1,2)
print ("Integer sum w/ prototypes is: ", isum)

dso.sum_of_double.argtypes = [ c_double, c_double ]
dso.sum_of_double.restype = c_double
dsum = dso.sum_of_double(0.5,2.5)
print ("Double sum w/ prototypes is: ", dsum)
```

Passing Strings

- Strings in python are read-only, thus when a C-function will modify a string we have to use `create_string_buffer()`

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./hello.so”)

# hello() in hello.so takes a “char *” argument
dso.hello.argtypes = [ c_char_p ]
dso.hello(b”World”)

# create buffer for mutable string data
buf = create_string_buffer(b”World”)
dso.hello(buf)
```

Passing Arrays

- When passing allocatable objects like arrays, it is usually best to do the allocating in python. ctypes offers constructors for all basic types

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
num = 10
dlist = (c_double * num)()  # (primitive * length)()
for i in range(num):
   dlist[i] = 0.333*(i*0.5)
# note the use of POINTER()
dso.sum_of_doubles.argtypes=[POINTER(c_double),c_int]
dso.sum_of_doubles.restype = c_double
dsum = dso.sum_of_doubles(dlist,num)
print ("Double sum is: ", dsum)
```

Passing Structs

- Even complex storage elements like `struct` can be managed by ctypes. Derive a class from Structure that mimics the corresponding C-type

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./data.so”)

class parm(Structure):
  _fields_ = [ ("type", c_int), ("label", c_char_p),
               ("epsilon",c_double),("sigma",c_double) ]
# use constructor to initialize struct
p = parm(type=1,label=b"LJ-12-6",epsilon=0.1,sigma=3.4)
# p is passed by value, to pass by reference use byref(p)
dso.pass_by_value(p)
dso.pass_by_reference(byref(p))
```

Passing Structs (2)

- Below is the corresponding C code:

```
#include<stdio.h>

struct parm { int  type; char *label;
              double epsilon, sigma;
};

void pass_by_value(struct parm p) {
    printf("type=%d  label=%s epsilon=%g  sigma=%g\n",
           p.type, p.label, p.epsilon, p.sigma);
}

void pass_by_reference(struct parm *p) {
    printf("type=%d  label=%s epsilon=%g  sigma=%g\n",
           p->type, p->label, p->epsilon, p->sigma);
}
```

Interfacing Fortran with f2py

- Interfacing Fortran with Python is both easier and more complicated than interfacing C
  - The Fortran ABI can be much more complex and is more compiler specific than the C ABI
  - The NumPy project has a tool “f2py” that automates the process and hides most complications
- If you have a Fortran file with some functions or subroutine do: f2py -c code.f90 -m mymodule
  - Creates python loadable module “mymodule”
  - Flag '-c' calls compiler; flag '-m' sets module name

Interfacing Fortran with f2py (2)

- Then in python do: from mymodule import * and call the Fortran functions in python
- The f2py tool will parse the Fortran code and generate the necessary C-code for a module
- The f2py generated code will automatically insert code to convert data as needed; e.g. lists are converted to arrays
- The f2py tool works best with well formed Fortran code; otherwise data maps can help

Fortran examples for f2py

- Example code that converts cleanly with f2py:

```
subroutine hello
    print*, "Hello, World!"
end subroutine hello
function sum_of_int(a,b) result(c)
    integer, intent(in) :: a, b
    integer :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_int
function sum_of_double(a,b) result(c)
    double precision, intent(in) :: a, b
    double precision :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_double
```

Passing arrays with f2py

- Arrays are traditional style arrays with f2py:

```
function sum_of_doubles(a,n) result(s)
    double precision, intent(in) :: a(*)
    integer, intent(in) :: n
    double precision :: s
    integer :: i
    s = 0
    do i=1,n
        s = s + a(i)
    end do
end function sum_of_doubles
-----------------------------------------------
num = 10
dlist = [sqrt(float(i)) for i in range(1,num)]
dsum = sum_of_doubles(dlist,num)
```

Passing strings with f2py

- Strings are handled in a very similar fashion to traditional style arrays with f2py:

```
subroutine hello(name)
    character(len=*), intent(in) :: name
    print* , "Hello, ", name, "!"
end subroutine hello
-----------------------------------------------
#!/usr/bin/env python
from  hello import *

print ("Calling DSO")
hello('World')
print ("Done")
```

Interfacing Python with C and Fortran

Dr. Axel Kohlmeyer

Associate Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu