Semester 2 2024

Astroinformatics II

Tutorial 7: Integrating C++ and Python

In this turorial, we will see how to create a new Python function that makes use of C code for computations and that can be later used in a theory or view.

There are many solutions available to extend Python code with faster C code: Cython and Numba transform Python code into C executable and require minimal addition to the existing Python code. Ctypes provides C-compatible data types, and allows calling functions from external libraries, e.g. calling pre-compiled C functions. It is a very effective means to communicate with existing C code.

For this reason, we are using here Ctypes.

In general, Code already written in C will require no modifications to be used by Python. The only work we need to do to integrate C code in Python is on Python's side.

The steps for interfacing Python with C using Ctypes are:

1. write C code functions
2. compile the C code as a shared library
3. write some Python lines of code to use the C functions from the library
4. run your code.

Important: The shared library created can only be used by the same type of machine is was compiled on, i.e., a Windows machine creates libraries that cannot be used on Linux or Mac, and vice versa.

1 Compiling C code into a shared library

As an example of a C function, we write a simple function that takes an array of double as input and returns the square of it. It is evident that such a simple function would not justify the use of C, but it is a good place to start.

In a new text file, write the code below and save it as `basic_function.c`:

```c

#include <stdlib.h>

void c_square(int n, double *array_in, double *array_out)
{ //return the square of array_in of length n in array_out
    int i;

    for (i = 0; i < n; i++)
    {
        array_out[i] = array_in[i] * array_in[i];
    }
}
```

Task:

To compile the above C function into a library that can later be used by Python, open a terminal and change the working directory to the folder where `basic_function.c` is located. Then compile the library using
`gcc -o basic_function.so -shared -fPIC -O2 basic_function.c`
This created a new file `basic_function.so` containing our C function.

2 Use a C library in Python

We will now use the `basic_function.so` library via Ctypes.
For this, we create a Python file, named `basic_function_helper.py`, with the following content:

```
"""
Define the C-variables and functions from the C-files that are needed in Python
"""

from ctypes import c_double, c_int, CDLL
import sys

lib_path = 'basic_function.so'

try:
    basic_function_lib = CDLL(lib_path)

except:
    print('not found: ' % (sys.platform))


python_c_square = basic_function_lib.c_square

python_c_square.restype = None
```

Some explanations of the above code:
`from ctypes import c_double, c_int, CDLL`
imports the Python `ctypes` object we will be needing.

`basic_function_lib = CDLL(lib_path)`
defines the Python object `square_lib` where all the functions and variables from our C file `basic_function.c` are stored; in particular, the function `c_square`.

`python_c_square = basic_function_lib.c_square`
defines the Python equivalent of the C function `c_square`.

`python_c_square.restype = None`
defines what type of variables the C function returns. In our case, it is `void`, which translates in Python to `None`.

We now continue with writing code:
Our C function `c_square` accepts three arguments: an `int` and two `double *`. Hence, our Python function `python_c_square` accepts three arguments too but they must by of the appropriate types.
Therefore, to use `python_c_square`, we have to convert Python `int` into `c_int` type and Python `list` into `* c_double`.
The best way to do so is to write a Python function, in the file `basic_function_helper.py`:

```
def do_square_using_c(list_in):

    """Call C function to calculate squares"""

    n = len(list_in)
    c_arr_in = (c_double * n)(*list_in)
    c_arr_out = (c_double * n)()


    python_c_square(c_int(n), c_arr_in, c_arr_out)

    return c_arr_out[:]
```

Some explanations of the above code:
`c_arr_in = (c_double * n)(*list_in) c_arr_out = (c_double * n)()`
defines two ctypes arrays of `double` of size `n` that can be used by the C function. The first one is initialised with the values of `list_in`. It is equivalent to:

```
for i in range(n):
    c_arr_in[i] = c_double(list_in[i])
```

The line:
`python_c_square(c_int(n), c_arr_in, c_arr_out)`
calls the C function that does the computation of the square of `c_arr_in` and put the result in `c_arr_out`. Note the conversion `c_int(n)` that transforms the Python `int` into a `ctypes int`.

Finally, this line:
`return c_arr_out[:]`
returns a copy of the results as a Python list.

With the above, our C function `c_square` is now wrapped into a Python function `do_square_using_c`. To use it in a Python program, simply import the function by including in the module header.

As an example on how to use the C function, the following code calculates the square of numbers from 0 to 999:

```
from basic_function_helper import do_square_using_c
...
my_list = np.arange(1000)
squared_list = do_square_using_c(*my_list)
```

Task:

a) Compile the complete code example and run it.

b) Modify the code so now instead of computing the square, you have two arrays which you multiply.