

np_inline 0.2

J. David Lee

May 7, 2011

Overview

The *np_inline* module was written as a simplified replacement for *scipy.weave.inline* for numeric computations on numpy arrays.

Simple example usage

Hello world

```
from np_inline import inline_debug as inline

code = r'printf("Program #%i: Hello world.\n", i);'

inline('hello_world_example',
      args=(1, ),
      py_types=((int, 'i'), ),
      code=code)
```

Note that inline cannot be used without passing at least one argument in *args*. This is a feature.

Matrix multiplication

```
import numpy as np
from np_inline import inline_debug as inline

arr = np.array((
    (0, 1, 2, 3),
    (4, 5, 6, 7),
    (8, 9, 0, 1)), dtype=np.float32)

m = 0.5

code = r'''
int i, j;
```

```

for(i = 0; i < arr_shape(0); ++i) {
    for(j = 0; j < arr_shape(1); ++j) {
        arr(i, j) = m * arr(i, j);
    }
}
"""

inline('array_mult_example',
      args=(m, arr),
      py_types=((float, 'm'), ),
      np_types=((np.float32, 2, 'arr'), ),
      code=code)

print(arr)

```

Returning a value

```

import numpy as np
from np_inline import inline_debug as inline

arr = np.arange(0, 10, 0.5, dtype=np.float32)

code = r'''
int i;
return_val = 0;
for(i=0; i < arr_shape(0); ++i) {
    return_val += arr(i);
}
'''
inline('return_val_test',
      args=(arr, ),
      np_types=((np.float32, 1, 'arr'), ),
      code=code,
      return_type=float)

```

Usage notes

Passing arguments

The primitive python types that can be passed as arguments are *int* and *double*, while almost any numeric type of numpy *ndarray* can be pass as an argument. The arguments must be sorted so the primitive python types precede the numpy types.

Primitive python types

Python primitive types are converted to C types as follows:

```
int    --> long int
float  --> double
```

For each primitive python object passed in, the type and name must be provided in the *py_types* argument. For example, to pass an integer argument, *i*, and a float argument, *f*, the *py_types* argument would be

```
((int, 'i'), (float, 'f'))
```

Numpy arrays

Numpy defines C data types corresponding to each numeric type:

```
uint8    --> npy_uint8
uint16   --> npy_uint16
uint32   --> npy_uint32
uint64   --> npy_uint64
int8     --> npy_int8
int16    --> npy_int16
int32    --> npy_int32
int64    --> npy_int64
float32  --> npy_float32
float64  --> npy_float64
float128 --> npy_float128
```

For each numpy array passed in, the numpy type, dimension, and name must be provided in the *np_types* argument. For example, to pass in a 1D int32 array, *arr*, and a 3D float64 array, *arr3d*, the *np_types* argument would be

```
((np.int32, 1, 'arr'), (np.float64, 3, 'arr3d'))
```

For an *N*-dimensional numpy array, *arr*, several variables and a macro are available for use in the C code:

```
py_arr      <-- The PyArrayObject pointer.
arr(i1, i2, ..., iN) <-- Macro indexing into the array.
arr_ndim    <-- Macro giving the number of dimensions of arr.
arr_shape(i) <-- Macro giving the length of arr in dimension i.
               Like arr.shape[i] in python.
```

Return values

The function can return three types of values, int, float, or None, depending on the value of the *return_type* argument. In order to set the return value in the C code, set the variable *return_val* to the return value in your C code.

Passing numpy arrays to C functions

In order to utilize the macros above to access array variables in functions given in the support code, the `PyArrayObject` pointer would need to be passed with the same name. That is, if the array is named *arr*, then the function would need to take an argument of the form

```
PyArrayObject *py_arr
```

External source files

Both the function's C-code and the support-code can be stored in external files, and accessed using the *code_path* and *support_code_path* arguments.

External libraries

In order to link external libraries, the appropriate keyword arguments need to be supplied to the *distutils.core.Extension* constructor. These arguments can be passed into the *inline* or *inline_debug* functions as a dictionary via the *extension_kwargs* argument.

Debugging

There are two inline functions in *np_inline*. One is called *inline*, and the other is called *inline_debug*. The *inline_debug* function performs several checks of the incoming data before calling the *inline* function. It also forces the code to be compiled the first time it is called, instead of checking if the module exists on disk. A simple way to switch between the two is to use the following imports:

```
from np_inline import inline_debug as inline # Uses debugging.  
from np_inline import inline                # No debugging.
```

Implementation notes

Generated source files are stored in `~/np_inline` using the unique name provided by the user. The C extension file is built and installed in the same directory using *distutils*.

Compilation is triggered only if the user-supplied unique name changes, or the module file doesn't exist.