Circumventing The Linker: Using SciPy's BLAS and LAPACK Within Cython

Ian Henriksen[‡*]

https://www.youtube.com/watch?v=R4yB-8tB0J0

Abstract—BLAS, LAPACK, and other libraries like them have formed the underpinnings of much of the scientific stack in Python. Until now, the standard practice in many packages for using BLAS and LAPACK has been to link each Python extension directly against the libraries needed. Each module that calls these low-level libraries directly has had to link against them independently. The task of finding and linking properly against the correct libraries has, in the past, been a substantial obstacle in the development and distribution of Python extension modules.

Cython has existing machinery that allows C-level declarations to be shared between Cython-compiled extension modules without linking against the original libraries. The Cython BLAS and LAPACK API in SciPy uses this functionality to make it so that the same BLAS and LAPACK libraries that were used to compile SciPy can be used in Python extension modules via Cython. This paper will demonstrate how to create and use these APIs for both Fortran and C libraries in a platform-independent manner.

Index Terms—Cython, BLAS, LAPACK, SciPy

Introduction

Many of the primary underpinnings of the scientific Python stack rely on interfacing with lower-level languages, rather than working with code that is exclusively written in Python. SciPy [SciPy], for example, is a collection of algorithms and libraries implemented in a variety of languages that are wrapped to provide convenient and usable APIs within Python. Because programmers often need to call low-level libraries, F2PY [F2PY], Cython [Cython], and a variety of similar tools have been introduced to simplify that process.

In spite of the large number of tools for automatically wrapping low-level libraries, interfacing with low-level languages can still present a significant challenge. If performance bottlenecks depend on any third party algorithms, developers are faced with the daunting task of rewriting their algorithms to interface with completely different packages and adding large dependencies on existing low-level libraries. Adding these dependencies to an existing project can complicate the build process and expose the project to a much wider variety of bugs. When developers distribute code meant to work reliably with a variety of compilers in a variety of environments, low-level dependencies become a never-ending source of trouble. The problems caused by these dependencies are further complicated by the fact that, currently, each Python module must shoulder the burden of distributing or finding the libraries it uses.

For example, consider the case of a simple tridiagonal matrix solve. This sort of solve can be done easily within Python.

```
import numpy as np
def pytridiag(a, b, c, x):
    """ Solve the system A y = x for y
    where A is the square matrix with subdiagonal
    'a', diagonal 'b', and superdiagonal 'c'. """
    A = np.zeros((b.shape[0], b.shape[0]))
    np.fill_diagonal(A[1:], a)
    np.fill_diagonal(A, b)
    np.fill_diagonal(A[:,1:], c)
    return np.linalg.solve(A, x)
```

This function works fine for small problems, but, if it needs to be called frequently, a more specialized algorithm could provide major improvements in both speed and accuracy. An ideal candidate for this sort of optimization is LAPACK's [LAPACK] routine `dgtsv`. That routine can be used within Cython to solve the same problem more quickly and with fewer numerical errors.

```
# cython: wraparound = False
# cython: boundscheck = False

cdef extern from "lapacke.h" nogil:
    void dgtsv "LAPACK_dgtsv"(int *n, int *nrhs,
                              double *dl, double *d,
                              double *du, double *b,
                              int *ldb, int *info)

cpdef tridiag(double[::1] a, double[::1] b,
              double[::1] c, double[::1] x):
    cdef int n=b.shape[0], nrhs=1, info
    # Solution is written over the values in x.
    dgtsv(&n, &nrhs, &a[0], &b[0], &c[0], &x[0],
          &n, &info)
```

Though this process for calling an external function from a library is not particularly difficult, the setup file for the Python module now must find a proper LAPACK installation. If there are several different versions of LAPACK present, a suitable one must be chosen. The proper headers and libraries must be found, and, if at all possible, binary incompatibilities between compilers must be avoided. If the desired routine is not a part

* Corresponding author: iandh@byu.edu
‡ Brigham Young University Math Department

of one of the existing C interfaces, then it must be called via the Fortran ABI and the name mangling schemes used by different Fortran compilers must be taken into account. All of the code needed to do this must also be maintained so that it continues to work with new versions of the different operating systems, compilers, and BLAS and LAPACK libraries.

An effective solution to this unusually painful problem is to have existing Python modules provide access to the low-level libraries that they use. NumPy has provided some of this sort of functionality for BLAS and LAPACK by making it so that the locations of the system's BLAS and LAPACK libraries can be found using NumPy's distutils module. Unfortunately, the existing functionality is only usable at build time, and does little to help users that do not compile NumPy and SciPy from source. It also does not include the various patches used by SciPy to account for bugs in different BLAS and LAPACK versions and incompatibilities between compilers.

Cython has provided similar functionality that allows C-level APIs to be exported between Cython modules without linking. In the past, these importing systems have been used primarily to share Cython-defined variables, functions and classes between Cython modules. If used carefully, however, the existing machinery in Cython can be used to expose functions and variables from existing libraries to other extension modules. This makes it so that other Python extension modules can use the functions it wraps without having to build, find, or link against the original library.

The Cython API for BLAS and LAPACK

Over the last year, a significant amount of work has been devoted to exposing the BLAS and LAPACK libraries within SciPy at the Cython level. The primary goals of providing such an interface are twofold: first, making the low-level routines in BLAS and LAPACK more readily available to users, and, second, reducing the dependency burden on third party packages.

Using the new Cython API, users can now dynamically load the BLAS and LAPACK libraries used to compile SciPy without having to actually link against the original BLAS and LAPACK libraries or include the corresponding headers. Modules that use the new API also no longer need to worry about which BLAS or LAPACK library is used. If the correct versions of BLAS and LAPACK were used to compile SciPy, the correct versions will be used by the extension module. Furthermore, since Cython uses Python capsule objects internally, C and C++ modules can easily access the needed function pointers.

BLAS and LAPACK proved to be particularly good candidates for a Cython API, resulting in several additional benefits:

- Python modules that use the Cython BLAS/LAPACK API no longer need to link statically to provide binary installers.
- The custom ABI wrappers and patches used in SciPy to provide a more stable and uniform interface across different BLAS/LAPACK libraries and Fortran compilers are no longer needed for third party extensions.
- The naming schemes used within BLAS and LAPACK make it easy to write type-dispatching versions of BLAS and LAPACK routines using Cython's fused types.

In providing these low-level wrappers, it was simplest to follow the calling conventions of BLAS and LAPACK as closely as possible, so all arguments are passed as pointers. Using the new Cython wrappers, the tridiagonal solve example shown above can be implemented in Cython in nearly the same way as before, except that all the needed library dependencies have already been resolved within SciPy.

```
# cython: wraparound = False
# cython: boundscheck = False

from scipy.linalg.cython_lapack cimport dgtsv

cpdef tridiag(double[::1] a, double[::1] b,
              double[::1] c, double[::1] x):
    cdef int n=b.shape[0], nrhs=1, info
    # Solution is written over the values in x.
    dgtsv(&n, &nrhs, &a[0], &b[0], &c[0], &x[0],
          &n, &info)
```

Since Cython uses Python's capsule objects internally for the cimport mechanism, it is also possible to extract function pointers directly from the module's `__pyx_capi__` dictionary and cast them to the needed type without writing the extra shim.

Exporting Cython APIs for Existing C Libraries

The process of exposing a Cython binding for a function or variable in an existing library is relatively simple. First, as an example, consider the following C file and the corresponding header.

```
// myfunc.c
double f(double x, double y){
    return x * x - x * y + 3 * y;
}

// myfunc.h
double f(double x, double y);
```

This library can be compiled by running `clang -c myfunc.c -o myfunc.o`.

This can be exposed at the Cython level and exported as a part of the resulting Python module by including the header in the pyx file, using the function from the C file to create a Cython shim with the proper signature, and then declaring the function in the corresponding pxd file without including the header file. A similar approach using function pointers is also possible. Here's a minimal example that demonstrates this process:

```
# cy_myfunc.pyx
# Use a file-level directive to link
# against the compiled object.
# distutils: extra_link_args = ['myfunc.o']
cdef extern from 'myfunc.h':
    double f(double x, double y) nogil
# Declare both the external function and
# the Cython function as nogil so they can be
# used without any Python operations
# (other than loading the module).
cdef double cy_f(double x, double y) nogil:
    return f(x, y)

# cy_myfunc.pxd
# Don't include the header here.
```

```
# Only give the signature for the
# Cython-exposed version of the function.
cdef double cy_f(double x, double y) nogil
```

```
# cy_myfunc_setup.py
from distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules=cythonize('cy_myfunc.pyx'))
```

From here, once the module is built, the Cython wrapper for the C-level function can be used in other modules without linking against the original library.

Exporting a Cython API for an existing Fortran library

When working with a Fortran library, the name mangling scheme used by the compiler must be taken into account. The simplest way to work around this would be to use Fortran 2003's ISO C binding module. Since, for the sake of platform/compiler independence, such a recent version of Fortran cannot be used in SciPy, an existing header with a small macro was used to imitate the name mangling scheme used by the various Fortran compilers. In addition, for this approach to work properly, all the Fortran functions in BLAS and LAPACK were first wrapped as subroutines (functions without return values) at the Fortran level.

```
!     myffunc.f
!     The function to be exported.
      double precision function f(x, y)
        double precision x, y
        f = x * x - x * y + 3 * y
      end function f

!     myffuncwrap.f
!     A subroutine wrapper for the function.
      subroutine fwrp(out, x, y)
        external f
        double precision f
        double precision out, x, y
        out = f(x, y)
      end
```

```
// fortran_defs.h
// Define a macro to handle different
// Fortran naming conventions.
// Copied verbatim from SciPy.
#if defined(NO_APPEND_FORTRAN)
#if defined(UPPERCASE_FORTRAN)
#define F_FUNC(f,F) F
#else
#define F_FUNC(f,F) f
#endif
#else
#if defined(UPPERCASE_FORTRAN)
#define F_FUNC(f,F) F##_
#else
#define F_FUNC(f,F) f##_
#endif
#endif
```

```
// myffuncwrap.h
#include "fortran_defs.h"
void F_FUNC(fwrp, FWRP)(double *out, double *x,
                        double *y);
```

```
# cyffunc.pyx
cdef extern from 'myffuncwrap.h':
    void fort_f "F_FUNC(fwrp, FWRP)"(double *out,
                                     double *x,
                                     double *y) nogil
cdef double f(double *x, double *y) nogil:
    cdef double out
    fort_f(&out, x, y)
    return out

# cyffunc.pxd
cdef double f(double *x, double *y) nogil
```

Numpy's distutils package can be used to build the Fortran libraries and compile the final extension module. The interoperability between NumPy's distutils package and Cython is limited, but the C file resulting from the Cython compilation can still be used to create the final extension module.

```
# cyffunc_setup.py
from numpy.distutils.core import setup
from numpy.distutils.misc_util import Configuration
from Cython.Build import cythonize
def configuration():
    config = Configuration()
    config.add_library('myffunc',
                       sources=['myffunc.f',
                                'myffuncwrap.f'])
    config.add_extension('cyffunc',
                         sources=['cyffunc.c'],
                         libraries=['myffunc'])
    return config
# Run Cython to get the needed C files.
# Doing this separately from the setup process
# causes any Cython file-specific distutils
# directives to be ignored.
cythonize('cyffunc.pyx')
setup(configuration=configuration)
```

There are many routines in BLAS and LAPACK, and creating these wrappers currently still requires a large amount of boilerplate code. When creating these wrappers, it was easiest to write Python scripts that used F2PY's existing functionality for parsing Fortran files to generate a set of function signatures that could, in turn, be used to generate the needed code.

Since SciPy supports several versions of LAPACK, it was also necessary to determine which routines should be included as a part of the new Cython API. In order to support all currently used versions of LAPACK, we limited the functions in the Cython API to include only those that had a uniform interface from version 3.1 through version 3.5.

Conclusion

The new Cython API for BLAS and LAPACK in SciPy helps to alleviate the substantial packaging burden imposed on Python packages that use BLAS and LAPACK. It provides a model for including access to lower-level libraries used within a Python package. It makes BLAS and LAPACK much easier to use for new and expert users alike and makes it much easier for smaller modules to write platform and compiler independent code. It also provides a model that can be extended to other packages to help fight dependency creep and reduce the burden of package maintenance. Though it is certainly not trivial, it is still fairly easy to add new Cython bindings to an existing library. Doing so makes the lower-level libraries vastly easier to use.

Going forward, there is a great need for similar APIs for a wider variety of libraries. Possible future directions for the work within SciPy include using Cython's fused types to expose a more type-generic interface to BLAS and LAPACK,

writing better automated tools for generating wrappers that expose C, C++, and Fortran functions automatically, and making similar interfaces available in ctypes and CFFI.

REFERENCES

[SciPy] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[Cython] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. Cython: The Best of Both Worlds, Computing in Science and Engineering, 13, 31-39 (2011), DOI:10.1109/MCSE.2010.118

[F2PY] Pearu Peterson. F2PY: a tool for connecting Fortran and Python programs, International Journal of Computational Science and Engineering, 4 (4), 296-305 (2009), DOI:10.1504/IJCSE.2009.029165

[LAPACK] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. LAPACK Users' Guide Third Edition, Society for Industrial and Applied Mathematics, 1999.