Cython: The Best of Both Worlds

Cython is a Python language extension that allows explicit type declarations and is compiled directly to C. As such, it addresses Python's large overhead for numerical loops and the difficulty of efficiently using existing C and Fortran code, which Cython can interact with natively.

Python's success as a platform for scientific computing to date is primarily due to two factors. First, Python tends to be readable and concise, leading to a rapid development cycle. Second, Python provides access to its internals from C via the Python/C API. This makes it possible to interface with existing C, C++, and Fortran code, as well as to write critical sections in C when speed is essential.

Although Python is fast enough for many tasks, low-level computational code written in Python tends to be slow, largely due to Python's extremely dynamic nature. In particular, low-level computational loops are simply infeasible. Although NumPy eliminates the need for many such loops,[1] there will always be computations that can be expressed well only through looping constructs. Cython aims to be a good companion to NumPy in such cases.

Given the magnitude of existing, well-tested code in Fortran and C, rewriting it in Python would waste valuable resources. A big part of Python's role in science is its ability to couple existing components instead of reinventing the wheel. For example, the Python-specific SciPy library contains more than 200,000 lines of C++, 60,000 lines of C, and 75,000 lines of Fortran, compared to about 70,000 lines of Python code. Wrapping existing code has traditionally been the domain of Python experts because the Python/C API has a high learning curve. Although you can use such wrappers without ever knowing their internals, this approach draws a sharp line between users (using Python) and developers (using C with the Python/C API).

Cython solves both of these problems by compiling Python code (with some extensions) directly to C, which is then compiled and linked against Python and ready to use from the interpreter. Because it uses C types, Cython makes it possible to embed numerical loops, running at C speed, directly in Python code. Cython also significantly lowers the learning curve for calling C, C++, and Fortran code from Python. Using Cython, any programmer with knowledge of both Python and C, C++, or Fortran can easily use them together.

Here, we present an overview of the Cython language and compiler in several examples. We then offer guidelines as to when Cython can provide significantly higher performance than pure Python and NumPy code, and when NumPy is a good choice in its own right. We further show how the Cython compiler speeds up Python code, and how you can use it to interact directly with C code. Finally, we describe Fwrap, a close Cython

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Stefan Behnel
Senacor Technologies AG
Robert Bradshaw and Craig Citro
Google
Lisandro Dalcin
National Council for Scientific and Technological Research, Argentina
Dag Sverre Seljebotn
University of Oslo
Kurt Smith
University of Wisconsin–Madison

Related Work: Beyond Pure Python

The Cython-Pyrex fork has been one of the friendlier forks in open source, and we're thankful for Greg Ewing's cooperation. The two projects have somewhat different goals. Pyrex (www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex) aims to be a "smooth blend of Python and C," while Cython focuses more on preserving Python semantics where it can. Cython also contains features for numerical computation that aren't found in Pyrex (in particular, fast NumPy array access). Although there's a subset of syntax that works both in Pyrex and Cython, the languages are diverging; typically, you must choose one or the other. For example, the syntax for calling C++ code is different in Pyrex and Cython because this feature was added long after the fork.

Other projects—such as Weave and Instant—also make it possible to include compiled code in Python.[1] Another common approach is to implement the core algorithm in C, C++, or Fortran and then create wrappers for Python. You can create such wrappers using Cython or more specialized tools, such as the simplified wrapper and interface generator (SWIG), ctypes, Boost.Python, or Fortran to Python (F2PY). Each tool has its own flavor. SWIG can automatically wrap C or C++ code, while Cython and ctypes require redeclaration of the functions to wrap. SWIG and Cython require a compilation stage; ctypes doesn't. On the other hand, if you get a declaration wrong using ctypes, it can result in unpredictable program crashes. With Boost.Python, you implement a Python module in C++, which depending on whom you ask is either a great feature or a great disadvantage. Finally, as we describe in the main article, numexpr (http://code.google.com/p/numexpr) and Theano (http://deeplearning.net/software/theano) are specialized tools for quickly evaluating numerical expressions.

Generally speaking, Cython can be viewed as a Swiss army knife: it lacks the targeted functionality of more specialized tools, but its generality and versatility let you apply it in almost any situation that requires going beyond pure Python code.

Reference

1. I.M. Wilbers, H.P. Langtangen, and A. Oedegaard, "Using Cython to Speed up Numerical Python Programs," *Proc. 7th Nat'l Conf. Computational Mechanics* (MekIT'09), Norwegian Univ. Science and Technology, 2009, pp. 495–512.

relative that automatically creates fast wrappers around Fortran code to make it callable from C, Cython, and Python.

Cython at a Glance

As the "Related Work: Beyond Pure Python" sidebar describes, Cython is based on Greg Ewing's Pyrex (www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex). Cython extends the Python language with explicit type declarations of native C types. You can annotate attributes and function calls to be resolved at compile-time (as opposed to runtime). With the extra information from the annotations, Cython can generate code that sidesteps most of the usual runtime costs.

The generated code can take advantage of all the optimizations the C/C++ compiler is aware of without having to reimplement them as part of Cython. By integrating C and the Python runtime by automatically converting between Python types and C types, Cython lets programmers switch between the two without having to do anything by hand. The same applies when calling into external libraries written in C, C++, or Fortran. Accessing them is a native operation in Cython code, so it's trivial to call back and forth between Python code, Cython code, and native library code.

Of course, if we're manually annotating every variable, attribute, and return type with type information, we might as well be writing C/C++ directly. This is where Cython's approach to extending the Python language really shines. Anything that Cython can't determine statically is compiled with the usual Python semantics, meaning that you can selectively speed up only those parts of your program that expose significant execution times. The key thing to keep in mind here is the Pareto Principle, also known as the 80/20 rule—80 percent of the runtime is spent in 20 percent of the source code. So, a bit of annotation in the right spot can go a long way.

Cython thus has an extremely productive workflow: users can simply develop with Python, and if they find that they're spending significant time paying Python overheads, they can compile parts or all of their project with Cython, possibly providing some annotations to speed up the critical parts of the code. For code that spends almost all of its execution time in libraries doing things like fast Fourier transforms (FFTs), matrix multiplication, or linear system solving, Cython fills the same rapid development role as Python. However, as you extend the code with new functionality and algorithms, you can do this directly in Cython; by simply providing a little extra type information, you can get all of C's speed without all of the headaches.

```
def integrate_gamma(double a, double b, int n=10000):
  if (min(a, b) <= 0 or max(a, b) >= GSL_SF_GAMMA_XMAX):
    raise ValueError('Limits out of range (0, %f)' % GSL_SF_GAMMA_XMAX)

  cdef int i
  cdef double dx = (b - a) / n, result = 0
  for i in range(n):
      result += gsl_sf_gamma(a + i * dx) * dx
  return result
```

Figure 1. A Cython function to approximate the definite integral. The function is callable from Python and, with the exception of C types, is pure Python code.

```
import numpy as np
y = scipy.special.gamma(np.linspace(a, b, n, endpoint=False))
y *= ((b - a) / n)
result = np.sum(y)
```

Figure 2. The NumPy expression corresponding to the Cython expression in Figure 1. Although no faster than NumPy, the Cython expression is much more memory efficient.

A Simple Cython Example

As an introductory example, consider naive numerical integration of the Gamma function. A fast C implementation of the Gamma function is available (in the GNU Scientific Library, for example) and can easily be made available to Cython through some C declarations in Cython code:

```
cdef extern from "gsl/gsl_sf.h":
      double gsl_sf_gamma(double x)
      double GSL_SF_GAMMA_XMAX
```

where `double` refers to the double-precision floating-point type in C. As Figure 1 shows, you can then write a Cython function, callable from Python, to approximate the definite integral. This is pure Python code, except that C types (`int`, `double`) are statically declared for some variables using Cython-specific syntax. The `cdef` keyword is a Cython extension to the language, as is prepending the type in the argument list. In effect, Cython provides a mixture of C and Python programming.

The code example in Figure 1 is 30 times faster than the corresponding Python loop, and much more memory efficient (although not any faster) than the corresponding NumPy expression (see Figure 2).

As we describe later, Cython especially shines in more complicated examples for which loops are the most natural—or only viable—solution.

Writing Fast, High-Level Code

Python is a high-level programming language that constrains itself to a comparatively small set of simple yet powerful language constructs. To map them to efficient C code, the Cython compiler applies tightly tailored and optimized implementations for different use patterns. It therefore becomes possible to write simple code that executes efficiently.

Given how much time most programs spend in loops, an important target for optimizations is Python's `for` loop, which is really a `for-each` loop that can run over any iterable object. For example, the following code iterates over the lines of a file:

```
f = open('a_file.txt')
for line in f:
     handle(line)
f.close()
```

The Python language avoids special cases where possible, so there's no special syntax for a plain integer `for` loop. However, there's a common idiom for it, such as an integer loop from 0 to 999:

```
for i in range(1000):
     do_something(i)
```

The Cython compiler recognizes this pattern and transforms it into an efficient `for` loop in C if the value range and the loop variable's type allow it. Similarly, when iterating over a sequence, it's

```
cdef char* c_string = get_pointer_to_chars(10)
cdef char charval

# check if chars at offsets 3..9 are any of
'abcABC'
for char_val in c_string[3:10]:
   print( char_val in b'abcABC' )
```

Figure 3. An example of Cython's `for` loop optimizations. In addition to optimizing for the most important built-in Python container and string types, Cython can iterate directly over low-level types, such as C arrays of a known size or sliced pointers.

sometimes necessary to know the current index within the loop body. Python has a special function for this, called `enumerate()`, which wraps the iterable in a counter:

```
f = open('a_file.txt')
for line_no, line in enumerate(f):
     # prepend line number to line
     print("%d: %s" % (line_no, line))
```

Cython knows this pattern, too, and reduces the wrapping of the iterable to a simple counter variable so that the loop can run over the iterable itself, with no additional overhead. Cython's `for` loop has optimizations for the most important built-in Python container and string types and it can even iterate directly over low-level types, such as C arrays of a known size or sliced pointers (see Figure 3).

Another example where high-level language idioms lead to specialized low-level code is cascaded `if` statements. Many languages provide a special `switch` statement for testing integer(-like) values against a set of different cases. A common Python idiom uses the standard `if` statement:

```
if int_value == 1:
     func_A()
elif int_value in (2,3,7):
     func_B()
else:
     func_C()
```

This reads well, without needing a special syntax. However, C compilers often fold `switch` statements into more efficient code than sequential or nested `if-else` statements. If Cython knows that the type of the `int_value` variable is compatible with a C integer (such as an `enum` value), it can extract an equivalent switch statement directly from the previous code.

Several of these patterns have been implemented in the Cython compiler, and new optimizations are easy to add. It therefore becomes reasonable for code writers to

- stick to the Python language's simple and readable idioms;
- rely on the compiler to transform them into fast, well-specialized C language constructs; and
- take a closer look only at any code sections that prove to be performance critical in benchmarks.

Apart from its powerful control flow constructs, a high-level language feature that makes Python so productive is its support for object-oriented programming. True to the rest of the language, Python classes are highly dynamic: methods and attributes can be added, inspected, and modified at runtime, and new types can be dynamically created on the fly. Of course, this flexibility comes with a performance cost. Cython lets you statically compile classes into C-level `struct` layouts (with virtual function tables) such that they integrate seamlessly into the Python class hierarchy without any of the Python overhead. Although scientific data often fits nicely into arrays, that's not always the case, and Cython's support for compiled classes lets you efficiently create and manipulate more complicated data structures, including trees, graphs, maps, and other heterogeneous, hierarchal objects.

Typical Use Cases

Developers have successfully used Cython in many situations—from the half-million lines of Cython code in Sage (www.sagemath.org), to providing Python-friendly wrappers to C libraries, to creating small, personal projects. The following example use cases demonstrate Cython's value.

Sparse Matrices

SciPy and other libraries provide the basic high-level operations for working with sparse matrices. However, constructing sparse matrices often follows complicated rules for which elements are nonzero. Such code can rarely be expressed in terms of NumPy expressions—the most naive method would need temporary arrays of the same size as the corresponding dense matrix, thus defeating the purpose! As Figure 4 shows, Cython is ideal for this, however, because we can easily and quickly populate sparse matrices element by element.

Data Transformation and Reduction

Consider computing a simple expression for a large number of different input values, such as

```
v = np.sqrt(x**2 + y**2 + z**2)
```

```
import numpy as np
cimport numpy as np
...
cdef np.ndarray[np.intc_t] rows = np.zeros(nnz, dtype=np.intc)
cdef np.ndarray[np.intc_t] cols = np.zeros(nnz, dtype=np.intc)
cdef np.ndarray[double] values = np.zeros(nnz, dtype=np.double)
cdef int idx = 0

for idx in range(0, nnz):
    # Compute next non-zero matrix element
    ...
    rows[idx] = row; cols[idx] = col; values[idx] = value

# Finally, we construct a regular SciPy sparse matrix:
return scipy.sparse.coo_matrix((values, (rows, cols)), shape=(N,N))
```

Figure 4. Constructing sparse matrices. Unlike SciPy and other libraries, which can complicate this process, Cython lets users easily and quickly populate sparse matrices element by element.

where the variables are arrays for three vectors `x`, `y`, and `z`. For such expressions, you usually don't need to use Cython because the NumPy operations tend to be fast enough.

The exceptions are for either very small or very large amounts of data. For small data sets that are evaluated numerous times, the Python overhead of the NumPy expression will dominate, and making a loop in Cython removes this overhead. For large amounts of data, NumPy has two problems: it requires large amounts of temporary memory, and it repeatedly moves temporary results over the memory bus. In most scientific settings, the memory bus can easily become the main bottleneck, *not* the CPU.[2] In the last example, NumPy will first square `x` in a temporary buffer, then square `y` in another temporary buffer, then add them together using a third temporary buffer, and so on. As Figure 5 shows, in Cython, it's possible to manually write a loop running at native speed that avoids these problems because it doesn't require a temporary buffer. The speedup, for large arrays, is on the order of a factor of 10.

If you do many transformations like this, you should also evaluate numexpr (http://code.google.com/p/numexpr) and Theano (http://deeplearning.net/software/theano), which are dedicated to such tasks. Theano, for example, can reformulate the expression for optimal numerical stability and compute it on a highly parallel graphics processing unit (GPU).[3]

```
cimport libc
...
cdef np.ndarray[double] x = ..., y = ...,
  z = ...
cdef np.ndarray[double] v = np.zeros_
  like(x) ...
for i in range(x.shape[0]):
   v[i] = libc.sqrt(x[i]**2 + y[i]**2 +
     z[i]**2)
```

Figure 5. Manually writing a loop that runs at native speed. Because this loop doesn't require a temporary buffer, it offers considerable speedup for large arrays.

Optimization and Equation Solving

For numerical optimization or equation solving, the algorithm in question must be handed a function (a callback) that evaluates the function. The algorithm then relies on making new steps based on previously computed function values; the process is thus inherently sequential.

Depending on the problem's nature and size, you can use different optimization levels. For medium to large problems, the standard scientific Python routines integrate well with Cython. You simply declare types within the callback function and hand the callback to the solver just as you would with a pure Python function. Given the frequency with which this function might be called, the act of typing the callback function's variables combined with the Cython-implemented Python functions' reduced call overhead can have a noticeable impact on performance. How much impact depends heavily on the problem in question; as a rough indicator, we've noted a 40 times speedup when using this method on an ordinary differential equation in 12 variables.

For computationally simple problems in only a few variables, evaluating the function can be such

a quick operation that the Python function call's overhead for each step becomes relevant. In these cases, you might want to explore calling existing C or Fortran code directly from Cython. Some libraries have ready-made Cython wrappers: for example, Sage has Cython wrappers around the ordinary differential equation solvers in the GNU Scientific Library. In some cases, you might opt for implementing the algorithm directly in Cython to avoid any callback whatsoever—using Newton's method on equations in a single variable, for example.

Nonrectangular Arrays and Data Repacking

Cython is especially well suited to those situations in which data doesn't fit naturally in rectangular arrays. One such example arises in cosmology. Satellite experiments, such as the Wilkinson microwave anisotropy probe, have produced high-resolution images of the cosmic microwave background, a primary source of information about the early universe. The resulting images are spherical because they contain values for all directions of the sky.

It's possible to quickly reorder the data the way we want it with a Cython loop. With all the existing code out there wanting data in slightly different orders and formats, `for` loops aren't about to disappear.

The spherical harmonic transform of these maps, a Fourier transform on the sphere, is especially important. It has complex coefficients a_{lm}, where the indices run over $0 \leq l \leq l_{\max}$, $-l, \leq m \leq l$. An average of the entire map is stored in $a_{0,0}$, followed by three elements to describe the dipole component, $a_{1,-1}$, $a_{1,0}$, $a_{1,1}$, and so on. Data such as this can be stored in a 1D array and elements looked up at position $l^2 + l + m$.

It's possible, but not trivial, to operate on such data using NumPy whole-array operations. The problem is that NumPy functions, such as finding the variance, are primarily geared toward rectangular arrays. If the data was rectangular, we could estimate the variance per l, averaging over m, by calling `np.var(data, axis=1)`. This doesn't work for nonrectangular data. Although there are workarounds, such as the `reduceat` method and masked arrays, we've found it much more straightforward to write the obvious loops over l and m using Cython. In our case ($l_{max} = 1{,}500$), this is more than 1,000 times faster than the same loop written in Python. Using NumPy, we could loop over l and repeatedly call `np.var` for data subslices, but the Cython loops are still faster by a factor of 27. (Incidentally, the variance per l, or power spectrum, is the primary quantity of interest to observational cosmologists.)

The spherical harmonic transform we mentioned earlier is computed using the Fortran Hierarchical Equal Area isoLatitude Pixelization Library (HEALPix, http://healpix.jpl.nasa.gov), which can be called readily from Cython with the help of Fwrap. However, HEALPix spits out the result as a 2D array, with roughly half of the elements unoccupied. The waste of storage aside, 2D arrays are often inconvenient—with 1D arrays, we can treat each set of coefficients as a vector and perform linear algebra, estimate covariance matrices, and so on in the usual way. Again, it's possible to quickly reorder the data the way we want it with a Cython loop. With all the existing code out there wanting data in slightly different orders and formats, `for` loops aren't about to disappear.

Fwrap

Whereas C and C++ integrate closely with Cython, Fortran wrappers in Cython are generated with Fwrap (http://fwrap.sourceforge.net), a separate—and separately distributed—utility. The Fwrap tool automates wrapping Fortran source in C, Cython, and Python, which lets Fortran code benefit from Python's dynamism and flexibility. Developers can seamlessly integrate Fwrapped code into C, Cython, or Python projects. The utility transparently supports most of Fortran's 90/95/2003 features and handles nearly all Fortran 77 sources. Although Fwrap doesn't currently support derived types or function callbacks, such support is planned for an upcoming release.

Thanks to the Fortran 2003 standard's C interoperability features—which are supported in recent versions of all widely used Fortran 90/95 compilers—Fwrap generates wrappers that are portable across platforms and compilers. Fwrap is intended to be as friendly as possible, and it automatically handles Fortran parsing and generation. It also generates a project build script that will portably build a Python extension module from the wrapper files.

Fwrap is similar in intent to other Fortran-Python tools, including F2PY, PyFort, and

```
SUBROUTINE DGESDD(JOBZ, M, N, A, LDA, S, U, LDU, VT, LDVT, INFO)
!    . . Scalar Arguments . .
     CHARACTER, INTENT(IN) :: JOBZ
     INTEGER, INTENT(OUT) :: INFO
     INTEGER, INTENT(IN) :: LDA, LDU, LDVT, M, N
!    . . Array Arguments . .
     DOUBLE PRECISION, INTENT(INOUT) :: A(LDA, *)
     DOUBLE PRECISION, INTENT(OUT) :: S(*), U(LDU, *), VT(LDVT, *)

!    DGESDD subroutine body END SUBROUTINE DGESDD
```

Figure 6. Augmenting the Fortran 90 subroutine interface for `dgesdd`. To do this, we augment argument declarations with `INTENT` attributes and remove extraneous work array arguments.

Forthon. F2PY is distributed with NumPy and is a capable tool for wrapping Fortran 77 codes. Fwrap's approach differs in that it leverages Cython to create Python bindings. Manual tuning of the wrapper can be easily accomplished by simply modifying the generated Cython code, rather than using a restricted domain-specific language. Another benefit is reduced overhead when calling Fortran code from Python.

Consider a real-world example: wrapping a subroutine from netlib's Lapack Fortran 90 source. We'll use the Fortran 90 subroutine interface for `dgesdd`, which is used to compute the singular value decomposition arrays `U`, `S`, and `VT` of a real array `A`, such that `A = U * DIAG(S) * VT`. This routine is typical of Fortran 90 source code; it has scalar and array arguments with different intents and different data types. To simplify it here, we've augmented the argument declarations with `INTENT` attributes and removed extraneous work array arguments (see Figure 6).

When invoked on this Fortran code, Fwrap parses the code and makes it available to C, Cython, and Python. If desired, we can generate a deployable package for use on computers that don't have Fwrap or Cython installed. To use the wrapped code from Python, we must first set up the subroutine arguments—in particular, the `a` array argument. To do this, we set the array dimensions and then create the array, filling it with random values. To simplify matters, we set all array dimensions equal to `m` (see Figure 7).

The `asfortranarray()` function is important; it ensures that the array `a` is laid out in column-major ordering, or Fortran ordering. This ensures that no copying is required when passing arrays to Fortran subroutines. Any subroutine argument that is an `INTENT(OUT)` array must be passed to the subroutine. The subroutine will modify the array in place; no copies are made

```
>>> import numpy as np
>>> from numpy.random import rand

>>> m = 10

>>> rand_array = rand(m, m)
>>> a = np.asfortranarray(rand_array,
  dtype=np.double)
```

Figure 7. Using wrapped Python code. We first set up subroutine arguments, setting the array dimensions and creating an array filled with random values. To simplify matters, we set all array dimensions equal to `m`.

```
>>> s = np.empty(m, dtype=np.double,
  order='F')
>>> u = np.empty((m, m), dtype=np.double,
  order='F')
>>> vt = np.empty((m, m), dtype=np.double,
  order='F')
```

Figure 8. Three empty arrays. The `order='F'` keyword argument serves the same purpose as the `asfortranarray()` function.

for arrays of numeric types (this isn't required for scalar `INTENT(OUT)` arguments, such as the `INFO` argument). As Figure 8 shows, we create three empty arrays of appropriate dimensions. The `order='F'` keyword argument serves the same purpose as the `asfortranarray()` function.

Next, we import `dgesdd` from it and call it from Python (see Figure 9). The return value is a tuple that contains all arguments that were declared intent `out`, `inout`, or no intent spec. The `a` argument (intent `inout`) is in both the argument list and the return tuple, but no copy has been made.

```
>>> from fw_dgesdd import dgesdd

>>> jobz = 'A' # specify that we want all the output vectors
>>> (a, s, u, vt, info) = dgesdd(jobz, m, n, a, m, s, u, m, vt, m)
```

Figure 9. Importing `dgesdd`. The return value is a tuple that contains all arguments that were declared intent `out`, `inout`, or no intent spec.

```
>>> s_diag = np.diag(s)
>>> a_computed = np.dot(u, np.dot(s_diag, vt))
>>> np.allclose(a, a_computed)
True
```

Figure 10. Verifying the result. `a_computed` is equivalent to the matrix product `u * s_diag * vt`, and we verify that `a` and `a_computed` are equal to within machine precision.

We now verify that the result is correct. As Figure 10 shows, to do this, we create `a_computed`, which is equivalent to the matrix product `u * s_diag * vt`, and we verify that `a` and `a_computed` are equal to within machine precision. When calling the routine from within Cython code, the invocation is identical, and the arguments can be typed to reduce function call overhead.

Fwrap handles any kind of Fortran array declaration, whether assumed-size (as in our example), assumed-shape, or explicit shape. Options exist for hiding redundant arguments (such as the array dimensions `LDA`, `LDU`, and `LDVT`) and are covered in Fwrap's documentation.

Our example here covers the basics of what Fwrap can do. For more information, examples, downloads, and help using Fwrap, see http://fwrap.sourceforge.net. You can reach other users and the Fwrap developers on the Fwrap users mailing list, http://groups.google.com/group/fwrap-users.

Limitations

Compared to writing code in pure Python, Cython's primary disadvantages are compilation time and the need for a separate build phase. Most projects using Cython are therefore written in a mix of Python and Cython because Cython sources don't need to be recompiled when Python sources change. Cython can still be used to compile some Python modules for performance reasons. There's also an experimental "pure" mode in which decorators indicate static type declarations, which are valid Python and ignored by the interpreter at runtime, but are used by Cython when compiled. This combines the advantage of a fast edit-run cycle with the final product's high runtime performance. There's also the question of code distribution. Rather than requiring Cython as a dependency, many projects ship the generated C files that compile against Python 2.3 to 3.2 without any modifications as part of the distutils setup phase.

Compared to compiled languages such as Fortran and C, Cython's primary limitation is the limited support for shared memory parallelism. Python is inherently limited in its multithreading capabilities because it uses a global interpreter lock (GIL). Cython code can declare sections as containing only C code (using a `nogil` directive), which are then able to run in parallel. However, this can quickly become tedious. Currently, there's also no support for OpenMP programming in Cython. On the other hand, message passing parallelism using multiple processes—such as through MPI—is well supported.

Compared to C++, a major Cython weakness is its lack of built-in template support, which can help write code that works efficiently with many different data types. In Cython, you must either repeat code for each data type or use an external templating system (as with Fortran codes). Many template engines exist for Python, and most of them should work well for generating Cython code.

Using a language that can be either dynamic or static requires some experience. Cython is clearly useful when talking to external libraries, but when is it best to use Cython code in place of normal Python code? The obvious factor to consider is the code's purpose—is it a single experiment, for which the Cython compilation time might overshadow the pure Python runtime? Or is it a core library function, where every ounce of speed matters?

It's possible to paint some broad strokes when it comes to the type of computation considered. Are you spending the bulk of your time doing low-level number crunching in your code, or is the heavy lifting done through calls to external libraries? How easy is it to express the computation in terms of NumPy operations? For sequential algorithms such as equation solving and statistical simulations, it's indeed impossible to do without a loop of some kind. Pure Python loops can be very slow, but the impact varies depending on the use case.

If you think Cython might help with your project, your next stop is the Cython tutorial.[4] Optimization strategies and computation benchmarks are also available.[5] As always, the online documentation at http://docs.cython.org provides the most up-to-date information. Finally, if you're ever stuck, or just wondering if Cython can solve your particular problem, Cython has an active and friendly mailing list at http://groups.google.com/group/cython-users.

Acknowledgments

The US National Science Foundation partially supported the work of Robert Bradshaw (grant DMS 61-5655) and Craig Citro (grant DMS 0713225).

References

1. S. van der Walt, S.C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Eng.*, vol. 13, no. 2, 2011, pp. 22–30.
2. F. Alted, "Why Modern CPUs are Starving and What Can Be Done About It," *Computing in Science & Eng.*, vol. 12, no. 2, 2010, pp. 68–71.
3. J. Bergstra et al., "Optimized Symbolic Expressions and GPU Metaprogramming with Theano," *Proc. 9th Python in Science Conf.*, SciPy Community, 2010; www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/461.
4. S. Behnel, R.W. Bradshaw, and D.S. Seljebotn, "Cython Tutorial," *Proc. 8th Python in Science Conf.*, SciPy Community, 2009, http://conference.scipy.org/proceedings/SciPy2009/paper.1.
5. D.S. Seljebotn, "Fast Numerical Computations with Cython," *Proc. 8th Python in Science Conf.*, SciPy Community, 2009; http://conference.scipy.org/proceedings/SciPy2009/paper.2.

Stefan Behnel *is a senior software developer at Senacor Technologies AG in Germany and a freelance developer and consultant. His research interests include programming languages and software tooling for server architectures and high-performance computing. He regularly contributes to open source projects and is a core developer of two major projects in the Python community, including the Cython compiler. Behnel has a doctoral degree in computer science from the Darmstadt University of Technology. Contact him at consulting@behnel.de.*

Robert Bradshaw *is a software engineer at Google. His research interests include mathematical research (especially questions related to the Birch and Swinnerton-Dyer conjecture), programming language design, and open source software. Bradshaw has a PhD in mathematics from the University of Washington. Contact him at robertwb@gmail.com.*

Craig Citro *is a software engineer at Google. His research interests include computational mathematics (especially number theory), programming languages, and functional programming. Citro has a PhD in mathematics from the University of California, Los Angeles. Contact him at craigcitro@gmail.com.*

Lisandro D. Dalcin *is an assistant researcher at the National Council for Scientific and Technological Research of Argentina. His research interests include scientific computing in distributed memory architectures, medium- to large-scale finite element simulation, and computational fluid mechanics. Dalcin has a PhD in engineering from the National University of Littoral, Santa Fe, Argentina. Contact him at dalcini@gmail.com.*

Dag Sverre Seljebotn *is a doctoral student in the Institute of Theoretical Astrophysics at the University of Oslo, Norway, where he studies computational cosmology, in particular cosmic microwave background analysis. Seljebotn has a MSc in computational science from the University of Oslo, Norway. Contact him at dagseljebotn@gmail.com.*

Kurt Smith *is a doctoral student in the Department of Physics at the University of Wisconsin Madison, where he studies small-scale plasma turbulence. His research interests include numerical simulation, computational fluid dynamics, and interlanguage programming. Smith has a BS in physics and applied mathematics from the University of Dallas. He is a member of the American Physical Society and the ACM. Contact him at kwsmith1@wisc.edu.*