Cython: The Best of Both Worlds

Article *in* Computing in Science and Engineering · May 2011

DOI: 10.1109/MCSE.2010.118 · Source: IEEE Xplore

Lisandro Daniel Dalcin
King Abdullah University of Science and Tech…
80 PUBLICATIONS **737** CITATIONS
SEE PROFILE

Dag Sverre Seljebotn
University of Oslo
6 PUBLICATIONS **203** CITATIONS
SEE PROFILE

Cython: The best of both worlds

Stefan Behnel, Senacor Technologies AG Germany
Robert Bradshaw, Google USA
Craig Citro, Google USA
Lisandro Dalcin, National University of the Littoral Argentina
Dag Sverre Seljebotn, University of Oslo Norway
Kurt Smith, University of Wisconsin-Madison USA

This article is published in *IEEE Computing in Science and Engineering.* Please refer to the published version if accessible, as it contains editor's improvements. (c) 2011 IEEE.[*]
Permalink: http://dx.doi.org/10.1109/MCSE.2010.118

Abstract

Cython is an extension to the Python language that allows explicit type declarations and is compiled directly to C. This addresses Python's large overhead for numerical loops and the difficulty of efficiently making use of existing C and Fortran code, which Cython code can interact with natively. The Cython language combines the speed of C with the power and simplicity of the Python language.

Introduction

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 write critical sections in C when speed is essential.

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

Given the magnitude of existing, well-tested code in Fortran and C, rewriting any of this code in Python would be a waste of our valuable resources. A big part of the role of Python in science is its ability to couple together existing components instead of reinventing the wheel. For instance, the Python-specific SciPy library contains over 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 of existing code for use from Python has traditionally been the domain of the Python experts, as the Python/C API has a high learning curve. While one can use such wrappers without ever knowing about their internals, they draw 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, ready to use from the interpreter. Through its use of 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++/Fortran can easily use them together.

[*]Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

In this paper, we present an overview of the Cython language and the Cython compiler in several examples. We give guidelines on where Cython can be expected to provide significantly higher performance than pure Python and NumPy code, and where NumPy is a good choice in its own right. We further show how the Cython compiler speeds up Python code, and how it can be used to interact directly with C code. We also cover Fwrap, a close relative of Cython. Fwrap is used for automatically creating fast wrappers around Fortran code to make it callable from C, Cython, and Python.

Cython is based on Pyrex [Pyrex] by Greg Ewing. It's been one of the more friendly "forks" in open source, and we are thankful for Greg's cooperation. The two projects have somewhat different goals. 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 some features for numerical computation that are not found in Pyrex (in particular fast NumPy array access). While there is a subset of syntax that will work both in Pyrex and Cython, the languages are diverging and one will in general have to choose one or the other. For instance, the syntax for calling C++ code is different in Pyrex and Cython, since this feature was added long after the fork.

There are other projects that make possible the inclusion of compiled code in Python (e.g. Weave and Instant). A comparison of several such tools can be found in [comparison]. Another often used approach is to implement the core algorithm in C, C++ or Fortran and then create wrappers for Python. Such wrappers can be created with Cython or with more specialized tools such as SWIG, ctypes, Boost.Python or F2PY. Each tool has its own flavor. SWIG is able to automatically wrap C or C++ code while Cython and ctypes require redeclaration of the functions to wrap. SWIG and Cython require a compilation stage which ctypes does not. On the other hand, if one gets a declaration wrong using ctypes it can result in unpredictable program crashes without prior warning. With Boost.Python one implements a Python module in C++ which – depending on who you ask – is either a great feature or a great disadvantage.

Finally, numexpr[1] and Theano[2] are specialized tools for quickly evaluating numerical expressions (see below).

To summarize, Cython could be described as a swiss army knife: It lacks the targeted functionality of more specialized tools, but its generality and versatility allow its application in almost any situation that requires going beyond pure Python code.

Cython at a glance

Cython is a programming language based on Python, that is translated into C/C++ code, and finally compiled into binary extension modules that can be loaded into a regular Python session. Cython extends the Python language with explicit type declarations of native C types. One can annotate attributes and function calls to be resolved at compile-time (as opposed to runtime). With the extra information from the annotations, Cython is able to 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 re-implement them as part of Cython. Cython integrates the C language and the Python runtime through automatic conversions between Python types and C types, allowing the programmer to 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 calling back and forth between Python code, Cython code and native library code is trivial.

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. Here is where Cython's approach of 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 in this context is the Pareto Principle, also known

[1] http://code.google.com/p/numexpr/
[2] http://deeplearning.net/software/theano/

as the 80/20 rule: 80% of the runtime is spent in 20% of the source code. This means that a little bit of annotation in the right spot can go a long way.

This leads to an extremely productive workflow in Cython: users can simply develop with Python, and if they find that a significant amount of time is being spent 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 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 – and just by providing a little extra type information, you can get all the speed of C without all the headaches.

A simple example

As an introductory example, consider naive numerical integration of the Gamma function. A fast C implementation of the Gamma function is available e.g. in the GNU Scientific Library, and can easily be made available to Cython through some C declarations in Cython code (`double` refers to the double precision floating point type in C):

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

One can then write a Cython function, callable from Python, to approximate the definite integral:

```
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
```

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 above code is 30 times faster than the corresponding Python loop, and much more memory efficient (although not any faster) than the corresponding NumPy expression:

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

Cython especially shines in more complicated examples where for loops are the most natural or only viable solution. Examples are given below.

Writing fast high-level code

Python is a very high-level programming language, and constrains itself to a comparatively small set of language constructs that are both simple and powerful. 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 very efficiently.

Given how much time most programs spend in loops, an important target for optimizations is the for loop in Python, 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 is no special syntax for a plain integer `for` loop. However, there is a common idiom for it, e.g. for 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 type of the loop variable allow it. Similarly, when iterating over a sequence, it is sometimes required to know the current index inside of 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:

```
cdef char* c_string = \
    get_pointer_to_chars(10)
cdef char char_val

# 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' )
```

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 normal `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 (e.g. an `enum` value), it can extract an equivalent switch statement directly from the above 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 simple and readable idioms of the Python language, to rely on the compiler to transform them into well specialized and fast C language constructs, and to only take a closer look at the code sections, if any, that still 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 very 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 allows one to statically compile classes into C-level `struct` layouts (with virtual function tables) in such a way that they integrate seamlessly into the Python class hierarchy without any of the Python overhead. Though much scientific data fits nicely into arrays, sometimes it does not, and Cython's support for compiled classes allows one to efficiently create and manipulate more complicated data structures like trees, graphs, maps, and other heterogeneous, hierarchal objects.

Some typical usecases

Cython has been successfully used in a wide variety of situations, from the half a million lines of Cython code in Sage (http://www.sagemath.org), to providing Python-friendly wrappers to C libraries, to small personal projects. Here are some example usecases demonstrating where Cython has proved valuable.

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 seldomly 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!

Cython is ideal for this, as we can easily and quickly populate sparse matrices element by element:

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

Data transformation and reduction

Consider computing a simple expression for a large number of different input values, e.g.:

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

where the variables are arrays for three vectors `x`, `y` and `z`. This is a case where, in most cases, one does not need to use Cython – it is easily expressed by pure NumPy operations that are already optimized and usually fast enough.

The exceptions are for either extremely small or large amounts of data. For small data sets that are evaluated many, many 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 (for a detailed explanation see [Alted]). In the example above, 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.

In Cython, it is possible to manually write a loop running at native speed:

```
cimport libc
...
cdef np.ndarray[double] x, y, z, v
x = ...; y = ...; z = ...
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)
```

which avoids these problems, as no temporary buffers are required. The speedup is on the order of a factor of ten for large arrays.

If one is doing a lot of such transformations, one should also evaluate numexpr and Theano which are dedicated to the task. Theano is able to reformulate the expression for optimal numerical stability, and is able to compute the expression on a highly parallel GPU.

Optimization and equation solving

In the case of numerical optimization or equation solving, the algorithm in question must be handed a function (a “callback”) which evaluates the function. The algorithm then relies on making new steps depending on previously computed function values, and the process is thus inherently sequential. Depending on the nature and size of the problem, different levels of optimization can be employed.

For medium-sized to large problems, the standard scientific Python routines integrate well with with Cython. One simply declares types within the callback function, and hands the callback to the solver just like one would with a pure Python function. Given the frequency with which this function may be called, the act of typing the variables in the callback function, combined with the reduced call overhead of Cython implemented Python functions, can have a noticeable impact on performance. How much depends heavily on the problem in question; as a rough indicator, we have noted a 40 times speedup when using this method on a particular 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 overhead of the Python function call for each step becomes relevant. In these cases, one might want to explore calling existing C or Fortran code directly from Cython. Some libraries have ready-made Cython wrappers – for instance, Sage has Cython wrappers around the ordinary differential equation solvers in the GNU Scientific Library. In some cases, one might opt for implementing the algorithm directly in Cython, to avoid any callback whatsoever – using Newton's method on equations of a single variable comes to mind.

Non-rectangular arrays and data repacking

Sometimes data does not fit naturally in rectangular arrays, and Cython is especially well-suited to this situation. 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, as they contain values for all directions on the sky.

The spherical harmonic transform of these maps, a "fourier transform on the sphere", is especially important. It has complex coefficients $a_{\ell m}$ where the indices run over $0 \leq \ell \leq \ell_{\max}$, $-\ell \leq m \leq \ell$. 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 like this can be stored in a one-dimensional array and elements looked up at position $\ell^2 + \ell + m$.

It is 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 towards rectangular arrays. If the data was rectangular, one could estimate the variance per ℓ, averaging over m, by calling `np.var(data, axis=1)`. This doesn't work for non-rectangular data. While there are workarounds, such as the `reduceat` method and masked arrays, we have found it much more straightforward to write the obvious loops over ℓ and m using Cython. For comparison, with Python and NumPy one could loop over ℓ and call repeatedly call `np.var` for subslices of the data, which was 27 times slower in our case ($\ell_{\max} = 1500$). Using a naive double loop over both ℓ and m was more than a 1000 times slower in Python than in Cython. (Incidentally, the variance per ℓ, or power spectrum, is the primary quantity of interest to observational cosmologists.)

The spherical harmonic transform mentioned above is computed using the Fortran library HEALPix[3], which can readily be called 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 one can treat each set of coefficients as a vector, and perform linear algebra, estimate covariance matrices and so on, the usual way. Again, it is 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 order and formats, `for` loops are not about to disappear.

Fwrap

Whereas C and C++ integrate closely with Cython, Fortran wrappers in Cython are generated with Fwrap, a separate utility that is distributed separately from Cython. Fwrap [fwrap] is a tool that automates wrapping Fortran source in C, Cython and Python, allowing Fortran code to benefit from the dynamism and flexibility of Python. Fwrapped code can be seamlessly integrated into a C, Cython or Python project. The utility transparently supports most of the features introduced in Fortran 90/95/2003, and will handle nearly all Fortran 77 source as well. Fwrap does not currently support derived types or function callbacks, but support for these features is scheduled in an upcoming release.

Thanks to the C interoperability features supplied in the Fortran 2003 standard – and 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 handles the Fortran parsing and generation automatically. It also generates a build script for the project that will portably build a Python extension module from the wrapper files.

Fwrap is similar in intent to other Fortran-Python

[3] Hierarchical Equal Area isoLatitude Pixelization, Górski et al, http://healpix.jpl.nasa.gov/

tools such as F2PY, PyFort and 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 Cython.

Consider a real world example: wrapping a subroutine from netlib's `LAPACK` Fortran 90 source. We will use the Fortran 90 subroutine interface for `dgesdd`, 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 datatypes. We have augmented the argument declarations with `INTENT` attributes and removed extraneous work array arguments for illustration purposes:

```
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
```

When invoked on the above 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`:

```
>>> import numpy as np
>>> from numpy.random import rand
>>> m = 10
>>> rand_array = rand(m, m)
>>> a = np.asfortranarray(rand_array,
... dtype=np.double)
```

The `asfortranarray()` function is important – this ensures that the array `a` is laid out in column-major ordering, also known as "fortran ordering." This ensures that no copying is required when passing arrays to Fortran subroutines.

Any subroutine argument that is an `INTENT(OUT)` array needs to be passed to the subroutine. The subroutine will modify the array in place; no copies are made for arrays of numeric types. This is not required for scalar `INTENT(OUT)` arguments, such as the `INFO` argument. This is how one would create three empty arrays of appropriate dimensions:

```
>>> 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')
```

The `order='F'` keyword argument serves the same purpose as the `asfortranarray()` function.

The extension module is named `fw_dgesdd.so` (the file extension is platform-dependent). We import `dgesdd` from it and call it from Python:

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

The return value is a tuple that contains all arguments that were declared intent `out`, `inout` or with no intent spec. The `a` argument (intent `inout`) is in both the argument list and the return tuple, but no copy has been made.

We can verify that the result is correct:

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

Here 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. Again, please see the documentation for details and examples.

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

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

Limitations

When compared to writing code in pure Python, Cython's primary disadvantages are compilation time and the need to have a separate build phase. Most projects using Cython are therefore written in a mix of Python and Cython, as Cython sources don't need to be recompiled when Python sources change. Cython can still be used to compile some of the Python modules for performance reasons. There is also an experimental "pure" mode where decorators are used to 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 a high runtime performance of the final product. There is also the question of code distribution. Many projects, rather than requiring Cython as a dependency, ship the generated .c files which 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, due to the use of a Global Interpreter Lock (GIL). Cython code can declare sections as only containing 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, for instance through MPI, is very well supported.

Compared to C++, a major weakness is the lack of built-in template support, which aids in writing code that works efficiently with many different data types. In Cython, one must either repeat code for each data type, or use an external templating system, in the same way that is often done for Fortran codes. Many template engines exists for Python, and most of them should work well for generating Cython code.

Using a language which can be either dynamic or static takes some experience. Cython is clearly useful when talking to external libraries, but when is it worth it to replace normal Python code with Cython code? The obvious factor to consider is the purpose of the code – is it a single experiment, for which the Cython compilation time might overshadow the pure Python run time? Or is it a core library function, where every ounce of speed matters?

It is possible to paint some broad strokes when it comes to the type of computation considered. Is the bulk of time spent 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 is indeed impossible to do without a loop of some kind. Pure Python loops can be very slow; but the impact of this still varies depending on the use case.

Further reading

If you think Cython might help you, then the next stop is the Cython Tutorial [tutorial]. [numerics] presents optimization strategies and benchmarks for computations.

As always, the online documentation at

http://docs.cython.org provides the most up-to-date information. If you are ever stuck, or just wondering if Cython will be able to solve your particular problem, Cython has an active and friendly mailing list at http://groups.google.com/group/cython-users.

References

[Alted] F. Alted. Why modern CPUs are starving and what can be done about it. CiSE 12, 68, 2010.

[comparison] I. M. Wilbers, H. P. Langtangen, A. Oedegaard, Using Cython to Speed up Numerical Python Programs, Proceedings of MekIT‘09, 2009. URL: http://simula.no/research/sc/publications/Simula.SC.578

[fwrap] K. W. Smith, D. S. Seljebotn, Fwrap: Fortran wrappers in C, Cython & Python. URL: http://conference.scipy.org/abstract?id=19 Project homepage: http://fwrap.sourceforge.net/

[numerics] D. S. Seljebotn, Fast numerical computations with Cython, Proceedings of the 8th Python in Science Conference, 2009. URL: http://conference.scipy.org/proceedings/SciPy2009/paper_2

[NumPy] S. van der Walt, S. C. Colbert, G. Varoquaux, The NumPy array: a structure for efficient numerical computation, CiSE, present issue

[Pyrex] G. Ewing, Pyrex: A language for writing Python extension modules. URL: http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[Sage] William A. Stein et al. Sage Mathematics Software, The Sage Development Team, 2010, http://www.sagemath.org.

[Theano] J. Bergstra. Optimized Symbolic Expressions and GPU Metaprogramming with Theano, Proceedings of the 9th Python in Science Conference (SciPy2010), Austin, Texas, June 2010.

[numexpr] D. Cooke, F. Alted, T. Hochberg, G. Thalhammer, *numexpr* http://code.google.com/p/numexpr/

[tutorial] S. Behnel, R. W. Bradshaw, D. S. Seljebotn, Cython Tutorial, Proceedings of the 8th Python in Science Conference, 2009. URL: http://conference.scipy.org/proceedings/SciPy2009/paper_1