Using Cython to Speed up Numerical Python Programs

Ilmar M. Wilbers[1], *Hans Petter Langtangen*[1,3], *and Åsmund Ødegård*[2]

[1]Center for Biomedical Computing, Simula Research Laboratory, Oslo
[2]Simula Research Laboratory, Oslo
[3]Department of Informatics, University of Oslo, Oslo

Abstract

The present study addresses a series of techniques for speeding up numerical calculations in Python programs. The techniques are evaluated in a benchmark problem involving finite difference solution of a wave equation. Our aim is to find the optimal mix of user-friendly, high-level, and safe programming in Python with more detailed and more error-prone low-level programming in compiled languages. In particular, we present and evaluate Cython, which is a new software tool that combines high- and low-level programming in an attractive way with promising performance. Cython is compared to more well-known tools such as F2PY, Weave, and Instant. With the mentioned tools, Python has a significant potential as programming platform in computational mechanics.

Keywords: Python, compiled languages, numerical algorithms

INTRODUCTION

Development of scientific software often consumes large portions of research budgets in computational mechanics. Using human-efficient programming tools and techniques to save code development time is therefore of paramount importance. Python [13] has in recent years attracted significant attention as a potential language for writing numerical codes in a human-efficient way. There are several reasons for this interest. First, many of the features that have made MATLAB so popular are also present in Python. Second, Python is a free, open source, powerful, and very flexible modern programming language that supports all major programming styles (procedural, object-oriented, generic, and functional programming). Third, there exist a wide range of modules available for efficient code development related to the many non-numerical tasks met in scientific software, including I/O, XML, graphical user interfaces, Web interfaces, databases, and file/folder handling. Another important feature of Python is the strong support for building, testing, and distributing large simulation codes containing a mixture of Python, Fortran, C, and C++ code. Since the administrative, non-numerical tasks often fill up most of large scientific software packages, many code writers have a desire to develop new code in Python.

There is one major concern when using Python for scientific computations, namely the possible loss of computational speed. The present paper addresses this issue and explains how Python code written in MATLAB-style can be speeded up by using certain software tools and implementation techniques. In particular, we focus at a new and promising tool, Cython, which can greatly speed up pure Python code in a user-friendly way.

Numerical algorithms typically involve loops over array structures. It is well known that standard Python loops over long lists or arrays run slowly in Python. The speed can be acceptable in many settings, for example, when solving partial differential equations in one space dimension. However, one soon encounters computational mechanics applications where standard Python code needs hours to run while a corresponding Fortran or C code finishes within minutes.

One way of speeding up slow loops over large arrays is to replace the loops by a set of operations on complete arrays. This is known as vectorization. The speed-up can be dramatic, as we show later, but the correspondance between computer code and the mathematical exposition of the algorithm is not as clear as when writing loops over arrays.

Another strategy is to migrate the loops to compiled code, either in Fortran, C, or C++. There exist a range of tools for simplifying the combination of Python and compiled languages. F2PY [12] is an almost automatic tool for gluing Fortran 77/90 with Python. SWIG [1] and Boost.Python [2] are similar tools for combining Python with C or C++ code, but they require more manual intervention than F2PY. Tools like Instant [14] and Weave [6] allow "inline" C or C++ code in Python, i.e., strings with C or C++ code are inserted in the Python code, compiled on the fly, and called from Python. Cython [5] is a recently developed tool that extends the Python language with new syntax which enables automatic compilation of constructs to machine code and thereby achieving substantial speed-up of the program. One can also manually write all the necessary C code for migrating loops to C, without relying on any tool for automating the (quite complicated) communication of data between C and Python.

For a computational scientist who needs to develop computer code and who is interested in using Python, there is a key question: What type of technique should be used to speed up loops over arrays? Should the loops be vectorized? Should the loops be implemented in Fortran 77 and glued with Python by F2PY? Or should the loops be written in C or C++ and glued via SWIG, Boost.Python, Instant, or Weave? Or should one apply the new tool Cython and implement the loops in Python with some extra commands? The present paper provides information to help answer these questions. The information is extracted from a specific case study involving finite difference schemes over two-dimensional grids. Such type of algorithms arise in numerous contexts throughout computational mechanics.

The topic of optimizing Python code has received some attention in the literature. The website `scipy.org` [7] contains much useful information. Comparisons of some of the mentioned tools, applied to typical numerical operations in finite difference schemes, have been published in the scientific literature [3, 4, 9, 8]. None of these contributions address the potential of Cython, which is the aim of the present paper. The hope is that the present paper can act as a repository of smart tools and programming techniques for Python programmers who want to speed up their codes. For non-experts the paper may act as a motivation for picking up a new programming platform in computational mechanics and help them to navigate in the jungle of implementation tools.

The paper is organized as follows. First, we describe what Cython is. Then we describe a benchmark problem and how the various tools and programming techniques perform in that problem. Finally, in an appendix, we describe in detail how one can start with a MATLAB-style implementation of the benchmark problem in pure Python and apply tools and techniques to speed up the code.

WHAT IS CYTHON?

Cython [5] is a new extension of the Python language that aims at making the integration of C and Python simpler. In many ways, it is a mix of the both of them. Cython is based on Pyrex, which has been around for some years, but the former supports more cutting-edge functionality and optimizations. Development started in 2007, and in the summer of 2008, integration with NumPy [11] arrays was added to the project, allowing us to write fast numerical array-based code that is very close to Python itself, while running at the speed of C. Cython is developed actively, and therefore functionalities might still be added, and the documentation is not always correct. The Cython community, however, has a very active mailing list.

Cython does not simply translate Python code to C code. Instead, it uses the Python run-time environment, compiling everything directly to machine code. Because of this, the Python header files should be available on the system (e.g., by installing the package `python2.5-dev` on Ubuntu Linux).

Almost all Python code is valid Cython code, but not all Cython code is Python code. This means that one can start with plain Python code, and add Cython constructs to gain speed, as we illustrate in detail in the appendix. The main difference between Python code and Cython code is that the latter is statically typed, which means that we have to explicitly declare the type of our variables. We can use pure Python objects, but if these are to interact with any of the Cython code, they need to be casted to the right type, which in many cases is done automatically, but in some cases requires manual intervention.

For a Cython example, let us look at the following simple code for numerical integration using the Trapezoidal rule. The Python version looks like:

```
def f(x):
    return 2*x*x + 3*x + 1

def trapez(a, b, n):
    h = (b-a)/float(n)
    sum = 0
    x = a
    for i in range(n-1):
        x += h
        sum += f(x)
    sum += 0.5*(f(a) + f(b))
    return sum*h
```

This code runs fine with Cython, but in order to get a speed-up, we need to statically declare types using the `cdef` keyword. The following Cython code runs about 30 times faster than the Python code:

```
cdef f(double x):
    return 2*x*x + 3*x + 1

def trapez(double a, double b, int n):
    cdef double h = (b-a)/n
    cdef double sum = 0, x = a
    cdef int i
    for i in range(n-1):
        x += h
        sum += f(x)
    sum += 0.5*(f(a) + f(b))
    return sum*h
```

Functions defined with `def` and `cdef` can call each other within the Cython code, but functions defined with `cdef` cannot be called directly from Python, so one either has to use a function defined with the keyword `def`, or another keyword `cpdef` indicating that the function is to be used both as a Python and a Cython function. In the appendix we show more advanced use of the Cython functionality.

THE BENCHMARK PROBLEM

We prefer to investigate the performance of different programming tools and techniques on a not too complicated mathematical problem, yet one that has a real application in mechanics and that contains algorithmic steps of widespread importance in more advanced computational mechanics applications. Our benchmark of choice is then a standard wave equation in a heterogeneous medium with local wave velocity k:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [k \nabla u] \,. \tag{1}$$

For the tests in the present paper, we restrict the attention to two space dimensions:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(k(x,y)\frac{\partial u}{\partial y}\right). \tag{2}$$

We set the boundary condition to $u = 0$ for the whole boundary of a rectangular domain $\Omega = (0,1) \times (0,1)$. Further, u has the initial value $I(x,y)$ at $t = 0$ while $\partial u/\partial t = 0$. The initial shape of the wave is chosen as a Gaussian bell function,

$$I(x,y) = A \exp\left(\left(\frac{x - x_c}{2\sigma_x}\right)^2 + \left(\frac{y - y_c}{2\sigma_y}\right)^2\right),$$

with $A = 2$, $x_c = y_c = 0.5$, and $\sigma_x = \sigma_y = 0.15$. The function k is defined as $k(x,y) = \max(x,y)$, causing the waves to move along the line between the corners $(0,0)$ and $(1,1)$ of the domain instead of uniformly over the grid.

We solve the wave equation using the following finite difference scheme:

$$\begin{aligned} u^l_{i,j} = {} & \left(\frac{\Delta t}{\Delta x}\right)^2 [k_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - k_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})]^{l-1} \\ & + \left(\frac{\Delta t}{\Delta y}\right)^2 [k_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - k_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1})]^{l-1}. \end{aligned} \tag{3}$$

Here, $u^l_{i,j}$ represents u at the grid point x_i and y_j at time level t_l, where

$$\begin{aligned} x_i &= i\Delta x, i = 0, \ldots, n \\ y_i &= j\Delta y, j = 0, \ldots, m \text{ and} \\ t_l &= l\Delta t, \end{aligned}$$

Also, $k_{i+\frac{1}{2},j}$ is short for $k(x_{i+\frac{1}{2}}, y_j)$.

The finite difference scheme is explicit and only conditionally stable. We choose the largest possible time step,

$$\Delta t = \frac{1}{\max_{x,y} k(x,y)} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-\frac{1}{2}} .$$

The relevance of this benchmark problem goes far beyond physical problems that can be modeled by the standard wave equation. In essence, the resulting computer code will perform an update of a variable-coefficient Laplace operator, i.e., an operation of the form

$$u_{\text{new}} = Q + \nabla \cdot [k \nabla u_{\text{old}}] \,,$$

where Q is some known quantity. Such updates appear in a wide range of physical problems and numerical methods. Therefore, the code segment to be examined in this paper will be of relevance to most finite difference-based software for fluid flow, heat transport, and electrostatics. For example, consider an implicit scheme for a heat transport problem with nonlinear heat conduction. Splitting convection and diffusion in two steps, the diffusion step will give rise to a linear system that must be solved by iterative methods. Using a Conjugate Gradient method for this purpose, combined with a Multigrid preconditioner, typically involves the discrete differential operator in the smoother step of the Multigrid algorithm. This smoother step will be an update from u_{old} to u_{new} of the type shown above, and k may be computed from u_{old} values. Similarly, finite difference schemes for the Navier-Stokes equations almost exclusively apply a splitting of the equations, with Laplace operator steps of the same kind as in our wave equation. The main limitation of our benchmark problem is the use of structured grids and finite difference operators.

The scheme then gives rise to a linear system we have an implicit scheme for, say a heat transport problem, an iterative numerical solution strategy, say a multigrid method, will lead to updates of the type we have for an explicit scheme for the wave equation.

In the scheme, we need values of k between the grid points, as in $k_{i+\frac{1}{2},j}$. This is a straightforward computation if k is known as an explicit mathematical function, which it is in the present problem. However, in the more general case, $k(x, y)$ comes from measurements and will normally be known only over a grid. Assuming that k is only available at the same grid as we use for u, we need to approximate quantities like $k_{i+\frac{1}{2},j}$ by k values at the grid points. The most immediate approximation is the arithmetic mean,

$$k_{i+\frac{1}{2},j} \approx \frac{1}{2}\left(k_{i,j} + k_{i+1,j}\right), \tag{4}$$

with similar definitions for $k_{i-\frac{1}{2},j}$, $k_{i,j+\frac{1}{2}}$, and $k_{i,j-\frac{1}{2}}$. For problems involving highly discontinuous media (e.g., waves in geological media), it is more common to apply the harmonic mean:

$$k_{i+\frac{1}{2},j} \approx \left(\frac{1}{2}\left(\frac{1}{k_{i,j}} + \frac{1}{k_{i+1,j}}\right)\right)^{-1}. \tag{5}$$

It turns out that whether one applies an arithmetic or harmonic mean has substantial effect on the relative performance of different programming tools and techniques, in the present problem. The reason is that the harmonic mean implies significantly more work, which may then dominate over the array look-ups in the scheme (for the fastest implementation the harmonic mean computation actually consumes about 80% of the time of an update in the scheme). We may view the arithmetic and harmonic mean alternatives as two (physically motivated) models for two classes of the complexity of a diffusion operator in more complicated partial differential equations. The arithmetic mean represents a model where the k coefficient is quickly evaluated, while the harmonic mean represents a model with a significant set of arithmetic operations, which typically arises in nonlinear diffusion operators, perhaps with complicated constitutive laws. One may in the latter case be forced to call mathematical functions (e.g., the exponential function) as part of the $k_{i+\frac{1}{2}}$ computation, and the result is even more work than for the harmonic mean. It therefore makes sense to test both means as models for "simple" and "complicated" variable-coefficient Laplace operators.

We can now summarize our algorithm to be implemented:

```
Define $u_{i,j}$, $up_{i,j}$, and $um_{i,j}$ to represent $u^{l}_{i,j}$, $u^{l+1}_{i,j}$ and $u^{l-1}_{i,j}$
SET THE INITIAL CONDITIONS:
$u_{i,j} = I(x,y)$, for $i = 0,\ldots,n$ and $j = 0,\ldots,m$
CALCULATE THE FIRST TIME STEP
$t = 0$
START THE TIME LOOP:
while time $t \leq t_{stop}$
    $t \leftarrow t + \Delta t$
    SOLVE FOR CURRENT TIME STEP:
    for $i = 1,\ldots,n-1$:
        for $j = 1,\ldots,m-1$:
            calculate $up_{i,j}$ from (3)
    SET BOUNDARY CONDITIONS
    INITIALIZE FOR NEXT STEP:
    $um = u$
    $u = up$
```

Set of Tests

For all programs except for the pure Fortran program, we only consider the time spent by the main time loop. The time spent setting the initial condition and calculating the first time step, which needs to be treated separately, is thus ignored. Compared to the execution time of all the other time steps, this time is negligible (for most implementations).

The allocation and initialization of data structures as well as the time loop is in pure Python for all implementations, except the pure Fortran 77 code. Only the space loop with the variable-coefficient Laplace operator is subject to various optimization techniques. We have also performed timings where the time loop has been migrated to compiled code, but this gave negligible gain in speed, reminding us that the need for optimization is usually limited to small, very computing intensive parts of a numerical code.

There are two ways of evaluating the variable coefficient $k_{i,j}$ in the scheme: we can either implement k as a Python function[1], or we can store k values at the spatial grid points in an array. We have adopted the latter approach, since it is clearly much more efficient than calling a function.

The timings have been done on a Linux laptop with the setup as shown in Table 1. The laptop is a Macbook Pro with a 2.4 GHz processor and 2GB of memory running the latest Ubuntu release (8.10, Intrepid Ibex).

BENCHMARK RESULTS

In section we briefly report our main findings about the efficiency of different speed-up tools and implementation constructs. The forthcoming text assumes that the reader is familiar with what is programmed, but a basic introduction to the various programming elements is provided in the appendix.

[1] Even when the loops over the grid points are migrated to compiled code in C or Fortran, that code can call a Python function for evaluating $k(x,y)$. This is very convenient for easy specification of different $k(x,y)$ functions, but results in very slow code, because the overhead of calling Python from C or Fortran is significant. Doing this for every grid point slows down the compiled code dramatically.

Table 1: Machine Configuration

Software	Version
Python	2.5.2
Cython	0.10.3
NumPy	1.1.1
Numeric	24.2-9
numarray	1.5.2-4
Instant	0.9.5
F2PY	1.1.1 (NumPy)
Weave	0.6.0-11 (SciPy)

Table 2: Timing results; arithmetic mean

Implementation	CPU time
Pure Fortran	1.0
Weave with C arrays	1.2
Instant	1.2
F2PY	1.2
Cython	1.3
Weave with Blitz++ arrays	1.8
Vectorized NumPy arrays	13.2
Python with lists and Psyco	170
Python with NumPy and `u.item(i,j)`	520
Python with lists	760
Python with NumPy and `u[i,j]`	1520

We started out with the most natural approach, namely using NumPy arrays and explicit loops. As indexing plain Python lists is generally faster than indexing NumPy arrays, we also tried pure lists. For such a program utilizing only standard Python (with no NumPy arrays), the just-in-time compiler Psyco can speed up the code, here we obtained almost an order of magnitude. Further speed-up can be achieved by vectorizing array operations, i.e., operating on whole slices of arrays instead of accessing individual elements. This resulted in a speed-up of two orders of magnitude. For even faster execution, it was necessary to export the most computational intensive part of the code to a compiled language, as is done with Weave, Instant, and F2PY. We also created a pure Fortran 77 program and used this as a timing reference. Finally, we utilized Cython, starting with the pure Python code with indexing of NumPy arrays, and adding a series of Cython constructs to see how the performance gain evolved.

All the timings reported in Table 2 are done for a 250×250 grid with 707 time steps using the arithmetic mean for k. The timings are based on the mean value from ten runs, and divided by the CPU-time of the fastest implementation, which is the pure Fortran program. Table 3 shows the same results using the harmonic mean. Note that the pure Fortran program again is set to one time unit, although taking almost 10 times longer to run than the Fortran program using arithmetic mean. The other implementations showed less difference in speed between the

Table 3: Timing results; harmonic mean

Implementation	CPU time
Pure Fortran	1.0
Weave with C arrays	1.0
Instant	1.0
F2PY	1.0
Cython	1.0
Weave with Blitz++ arrays	1.1
Vectorized NumPy arrays	2.9
Python with lists and Psyco	17.1
Python with lists	80
Python with NumPy and `u.item(i,j)`	150
Python with NumPy and `u[i,j]`	190

arithmetic and harmonic mean. One reason why Fortran is so much faster in the arithmetic mean case may be that the compiler manages to apply aggressive optimization techniques and utilize parallelism and registers on the chip in an effective way. With the harmonic mean, there are more complicated arithmetics involved, and the same optimizations may not be applicable in this case. For the implementations with a significant part of the computations in Python, there is so much overhead anyway that the difference between the types of mean computations drowns in other, more costly operations.

Psyco and Numerical Python Packages

There are three implementations of numerical Python arrays: Numeric is the original implementation from the mid 1990s, numarray appeared some years later and offered several useful new features, while NumPy is a merge of the latter two, aiming at being the only serious implementation of numerical arrays for Python. In this section we go more into detail on the relative performance of these three implementations.

Table 4 reports the differences between the three array implementations and various syntax for indexing arrays. We also show to what extent Psyco can speed up Python code. The time is scaled to the pure Python version with lists (instead of arrays) and arithmetic mean for k. Note that it is easy to transform a Python code with arrays to a code with (nested) lists: just perform the operation `a = a.tolist()` for every array `a`, and then substitute the indexing syntax for arrays, `a[i,j]`, by the syntax for nested lists, `a[i][j]`. Psyco is good for plain Python loops, as it reduces the execution time with 70-80%. For loops using the numerical modules we only get a reduction of 20-30%. This is expected, as Psyco only manages to optimize the Python code calling the numerical modules, not the code within the numerical modules themselves. The one exception here is NumPy when using the function `item` for accessing array elements. This is because `item` is a function. What Psyco does, is to scan the code, and keep compiled versions of blocks of code that are used often in memory, instead of interpreting every line of code on the fly. The downside is that it uses a lot of memory. It is, however, possible to control the amount of memory that Psyco is allowed to use. The conclusion is that Psyco is very efficient for speeding up pure Python loops over lists.

Instead of rewriting loops in an extension module in C, we might get a speed-up of a significant

Table 4: Timing results for various numerical Python packages and Psyco, for arithmetic and harmonic mean of k

Implementation	Arithm.	Arithm. Psyco	Harm.	Harm. Psyco
Python with lists	1.00	0.22	1.00	0.33
NumPy and `u[i,j]`	2.00	1.55	2.17	1.83
NumPy and `u[i][j]`	2.32	1.92	2.50	2.19
NumPy and `u.item(i,j)`	0.68	0.30	1.78	0.30
Numeric and `u[i,j]`	1.35	1.08	1.29	1.03
Numeric and `u[i][j]`	1.13	0.83	1.08	0.79
numarray and `u[i,j]`	2.31	1.86	2.09	1.76
numarray and `u[i][j]`	1.63	1.10	1.49	1.12

factor by simply adding the line `psyco.bind(foo)` for a function `foo`, so the extra amount of work needed is an absolute minimum. It is important to note that both the space and time loops should reside within functions in order to gain the maximum speed-up, because only functions can be bound with Psyco.

Psyco has no support for 64-bit platforms, and development has ceased. Instead, the developer has decided to contribute to a project called Pypy. The just-in-time (JIT) compiler for this project apparently has a similar speed-up as did Psyco, but we were not successful in compiling a version of Pypy with the JIT compiler.

Next we comment upon some differences we have observed between the three implementations of numerical Python arrays. When replacing Python lists with NumPy arrays in plain `for`-loops, the execution time is increased by a factor of 2. This is for NumPy arrays using the syntax `u[i,j]`. Using Numeric with the syntax `u[i][j]` the running time is only increased about 10% compared to the list implementation. The reason that we do not get a bigger speed-up is due to the fact that lists are intrinsic Python objects that are highly optimized, whereas using one of the numerical modules implies using external code. Unless we use vectorization or perform array operations instead of simple element-by-element operations, using these modules will not give us any speed-up.

We have observed that numarray is the slowest of the three numerical Python modules in all cases. Using NumPy, the syntax `u[i,j]` is faster than `u[i][j]`. With Numeric and numarray, this is the other way around. Numeric is still the fastest, but using NumPy and the syntax `u.item(i,j)` runs at almost the same speed, sometimes the latter is even faster. For vectorized expressions, NumPy seems to be fastest, see Tables 2 and 3 as well as [4, 9].

Some difference is seen when using Numeric and numarray on 64-bit platforms compared to 32-bit platforms compared to the timings from NumPy. This is because the former two modules have no support for 64-bit arrays, while the latter uses these by default when available.

Cache size is an important factor. It is probably the main reason for the vectorized numerical Python modules not being faster, as a lot of temporary arrays are created. Even though they do get cleaned up, copying all the values between arrays takes time. In fact, running the same benchmarks on a 64-bit 32GB machine, the vectorized NumPy code is twice as fast relative to the Fortran program for the arithmetic mean.

CONCLUDING REMARKS

Python is a great programming platform for scientific applications. Maybe the most important features of Python are that the code looks clean, few statements can do a lot, and the statements are close to how one would write the algorithm in pseudo code in a paper or report. In fact, Python is sometimes referred to as “executable pseudo code”. Python constitues in particular a very powerful and human-effective tool for implementing all the non-numerical tasks one has to deal with in scientific software.

The potential problem with Python used for pure number crunching compared to compiled languages is the loss of speed. The present paper has presented a number of techniques to overcome this problem and evaluated the efficiency of each technique for solving the standard wave equation. At its core, the numerical test consists in applying a finite difference stencil to all points in a two-dimensional grid. Our findings is that plain Python runs slowly, but we are only required to optimize the loop over the grid – the rest of the code (data allocation, initialization, and the time loop) can be kept in Python. This is good news as it indicates that only very computing intensive nested loops need to be optimized.

Migrating the loops over the grid points to compiled code, either to Fortran 77 via F2PY or to C/C++ via Instant or Weave, results in a speed close to that of having the whole application in pure Fortran 77. The Fortran, C, or C++ code for the loops can be placed inside the Python program and compiled and linked on the fly.

Cython is a new, user-friendly alternative to mixing Python with Fortran, C, or C++ code. Since Cython can be viewed as an extension of the Python language, the optimization consists in “decorating” a first, rough, pure Python implementation of, e.g., nested loops by special constructs (see the appendix). These constructs automatically move large parts of the calculation to assembly code. The performance is almost as good as that of using Fortran, C, or C++ from Python, but Cython seems considerably easier to work with.

The present paper has outlined the potential of Cython and compared it to older alternatives in a quite simple model problem. Whether Cython is easy to program with and gives high performance for other numerical operations than applying finite difference stencil to structured grids, requires further investigation.

APPENDIX: DETAILS ABOUT THE DIFFERENT IMPLEMENTATIONS

The purpose of this appendix is two-fold: i) to give the reader a glimpse of what it means to program in Python and speed up the code using vectorization, F2PY, Weave, Instant, or Cython, and ii) to make the reader quickly started with the mentioned tools and list some important programming tricks that are not presented in a coherent fashion, related to a scientific computing problem, elsewhere in the literature.

In the following, we start with a naive implementation in pure Python, very close the type of implementation one would aim at in Matlab, IDL, or similar environments. We then introduce various tools and implementation tricks to speed up the code, one at a time. For each trick that came to our mind, we have judged whether the trick represents a reasonably clean implementation or if it is a specialized hack. In the latter case, we omit to describe the trick in this text. This means that one can sometimes get faster code than what we present by turning to constructions that the authors would not like to have in a clear scientific computing code. For this reason, we have also abandoned writing all C code by hand for the communication between Python and compiled code, and we have not considered operating directly on pointers to NumPy array objects in C, which could have some performance gain (see [8] for what this means technically).

The forthcoming text is written for readers with a basic knowledge of Python and NumPy programming.

NumPy

A standard implementation of the basic algorithms, utilizing NumPy arrays, goes as follows:

```
import numpy

def k_function(x, y):
    return max(x, y)

m = 250; n = 250    # grid size
dx = 1.0/m
dy = 1.0/n
# arrays that enter the numerical scheme:
k  = zeros((m+1, n+1))
up = zeros((m+1, n+1))
u  = zeros((m+1, n+1))
um = zeros((m+1, n+1))

# initialize u from initial condition, plus um and k

# make sure dt is float, not numpy.float:
dt = float(1/sqrt(1/dx**2 + 1/dy**2)/k.max())

while t <= t_stop:
    t += dt
    up = calculate_u(dt, dx, dy, u, um, up, k)
    um[:] = u
    u[:] = up
```

The `calculate_u` function implements the nested space loop in the finite difference scheme. Using an arithmetic mean for the variable coefficient, this loop reads

```
def calculate_u(dt, dx, dy, u, um, up, k):
    hx = (dt/dx)**2
    hy = (dt/dy)**2
    for i in xrange(1, u.shape[0]-1):
        for j in xrange(1, u.shape[1]-1):
            k_c = k[i,j]
            k_ip = 0.5*(k_c + k[i+1,j])
            k_im = 0.5*(k_c + k[i-1,j])
            k_jp = 0.5*(k_c + k[i,j+1])
            k_jm = 0.5*(k_c + k[i,j-1]
            up[i,j] = 2*u[i,j] - um[i,j] +
              hx*(k_ip*(u[i+1,j] - u[i,j]) - k_im*(u[i,j] - u[i-1,j])) +
              hy*(k_jp*(u[i,j+1] - u[i,j]) - k_jm*(u[i,j] - u[i,j-1]))
    return up
```

A point is worth mentioning here. It is tempting to compute the (maximum) time step as

```
dt = 1/sqrt(1/dx**2 + 1/dy**2)/k.max()
```

However, the `sqrt` function used here is the one imported from NumPy (since we do a `from numpy import *`), but it works with scalars too. The problem is that this `sqrt` function returns a `numpy.float64` object rather than a plain Python `float` object. The former turns out to slow down scalar computations.

Vectorization

One way to get rid of loops in a finite difference scheme is to rewrite the algorithm to operate on displaced slices of arrays:

```
def calculate_u(dt, dx, dy, u, um, up, k):
    hx = (dt/dx)**2
    hy = (dt/dy)**2
    k_c = k[1:m,1:n]
    k_ip = 0.5*(k_c + k[2:m+1,1:n])
    k_im = 0.5*(k_c + k[0:m-1,1:n])
    k_jp = 0.5*(k_c + k[1:m,2:n+1])
    k_jm = 0.5*(k_c + k[1:m,0:n-1])
    up[1:m,1:n] = 2*u[1:m,1:n] - um[1:m,1:n] +
      hx*(k_ip*(u[2:m+1,1:n] - u[1:m,1:n]) -
                  k_im*(u[1:m,1:n] - u[0:m-1,1:n])) +
      hy*(k_jp*(u[1:m,2:n+1] - u[1:m,1:n]) -
                  k_jm*(u[1:m,1:n] - u[1:m,0:n-1]))
    return up
```

This approach might give fast enough code (e.g., when using harmonic mean in the present case), but there is a severe limitation of the possible speed of vectorization. Although all loops are now in C, there are a dozen of binary operations between vectors in the code segment above, and each binary operation needs a temporary array to store the results. This gives rise to a number of temporary arrays, which are quite efficiently created and destroyed by Python, but the overhead prevents vectorization from reaching the speed of compiled loops where there are no temporary arrays present.

Weave

Weave comes as part of SciPy [7], which is a comprehensive package for numerical computations, containing Python interfaces to LAPACK/ATLAS, a wide range of special mathematical functions, and many packages from Netlib [10] for numerical integration, optimization, and solution of ordinary differential equation – to mention some of SciPy's rich functionality. One may view SciPy as an extension of NumPy.

Weave allows parts of the Python code to be written in C or C++, inside Python strings. The strings are compiled and linked with the Python program at run time (if not a previous compilation is sufficient). In our implementation, we used the function `inline` from Weave.

A central question is: how large parts of the original pure Python program above must be migrated to compiled code? Ideally, we would like to have as large portions of a numerical code as possible in Python and minimize the parts to be migrated. In the present application, we limit migration of code to the nested loops over the computational grid, i.e., the application of the discrete Laplace operator. The allocation of data structures, initialization of these, and the time loop are kept in Python.

Here is the implementation of the loops over the 2D grid, i.e., the `calculate_u` function, utilizing Weave:

```
def calculate_u(dt, dx, dy, u, um, up, k):
    # Weave with plain C arrays
    m, n = u.shape
    c_code=r’’’
int i, j;
double hx, hy, k_c, k_ip, k_im, k_jp, k_jm;
hx = pow(dt/dx, 2);
hy = pow(dt/dy, 2);
for (i=1; i<m-1; i++)
    for (j=1; j<n-1; j++){
        k_c = k[i*m+j];
        k_ip = 0.5*(k_c + k[(i+1)*m+j]);
        k_im = 0.5*(k_c + k[(i-1)*m+j]);
        k_jp = 0.5*(k_c + k[i*m+j+1]);
        k_jm = 0.5*(k_c + k[i*m+j-1]);
        up[i*m+j] = 2*u[i*m+j] - um[i*m+j] +
          hx*(k_ip*(u[(i+1)*m+j] - u[i*m+j]) - k_im*(u[i*m+j] - u[(i-1)*m+j])) +
          hy*(k_jp*(u[i*m+j+1] - u[i*m+j])   - k_jm*(u[i*m+j] - u[i*m+j-1]));
    }
’’’
    err = weave.inline(c_code, [’m’, ’n’, ’dt’, ’dx’, ’dy’,
                                ’u’, ’um’, ’up’, ’k’], compiler=’gcc’,)
    return up
```

Note that the C/C++ code is placed inside strings and passed on to `weave.inline` for compilation and linking (if necessary) and execution. Also note that we use plain one-dimensional (flat) C arrays, which makes indexing like `u[i,j]` less readable as `u[i*m+j]`. A syntax `u(i,j)` is possible by letting Weave utilize Blitz++ arrays (in C++). This is easily done with the keyword argument `type_converters` for `weave.inline`. However, the C++ code with Blitz++ runs slower than the code with plain C arrays: the scaled speeds being 1.7 and 1.2, respectively (for arithmetic mean – the differences are much smaller when the harmonic mean is used). Obviously, iterating over one-dimensional arrays when working with a 3D grid results in a syntax that is harder to debug, but the extra speed gained probably outweights the less attractive syntax.

Instant

Instant [14] is similar to Weave in that one can write C code inside a Python program and get it compiled and linked with the Python application. Technically, Instant applies SWIG [1] to the C code and automates the otherwise more complicated, manual process of using SWIG directly on a C function. Instant uses a cache system based on the checksums of files, making sure to not recompile code unless needed. The difference compared with Weave is that we actually need to write the entire C function, i.e., also the function definition and arguments, in addition to the loops over the grid points. This means that we must handle type declarations, including pointers, and we also need to import the function afterwards:

```
c_code=r’’’
void calculate_u(double dt, double dx, double dy,
          int nu, int* pu, double* u,
          int num, int* pum, double* um,
          int nup, int* pup, double* up,
          int nk, int* pk, double* k){
    int i=0, j=0, m = pu[0], n = pu[1];
    double hx, hy, k_c, k_ip, k_im, k_jp, k_jm;
    hx = pow(dt/dx, 2);
    hy = pow(dt/dy, 2);
    for (i=1; i<m-1; i++)
        for (j=1; j<n-1; j++){
            k_c = k[i*m+j];
            k_ip = 0.5*(k_c + k[(i+1)*m+j]);
            k_im = 0.5*(k_c + k[(i-1)*m+j]);
            k_jp = 0.5*(k_c + k[i*m+j+1]);
            k_jm = 0.5*(k_c + k[i*m+j-1]);
            up[i*m+j] = 2*u[i*m+j] - um[i*m+j] +
              hx*(k_ip*(u[(i+1)*m+j] - u[i*m+j]) -
                  k_im*(u[i*m+j] - u[(i-1)*m+j])) +
              hy*(k_jp*(u[i*m+(j+1)] - u[i*m+j]) -
                  k_jm*(u[i*m+j] - u[i*m+(j-1)]));
        }
}
’’’
build_module(code=c_code, system_headers=["numpy/arrayobject.h"],
             include_dirs=[get_include()],
             init_code="import_array();",
             modulename="instant_",
             arrays=[[’nu’, ’pu’, ’u’],
                     [’num’, ’pum’, ’um’],
                     [’nup’, ’pup’, ’up’],
                     [’nk’, ’pk’, ’k’]])

import instant_

def calculate_u(dt, dx, dy, u, um, up, k):
    instant_.calculate_u(dt, dx, dy, u, um, up, k)
    return up
```

As Instant is nothing more than a layer on top of SWIG simplifying things, the execution times for the Instant implementation are really those implied by using SWIG on C code. There are quite a few keyword arguments available to the `build_module` function, allowing us to use Instant for much of the functionality available through SWIG.

F2PY

F2PY is a tool that makes it very simple to combine Python with Fortran 77 and 90 code. A complete rewrite of F2PY is in progress, aiming at having full support for wrapping Fortran 95 code and parts of Fortran 2003 features, in particular derived types. Documentation is good, although not completely up-to-date at the moment. F2PY now comes as part of NumPy. As with Weave and Instant, we can place the Fortran code inside strings in the Python program and get it compiled and linked on the fly. Alternatively, we may run F2PY (from the command-line) on a set of Fortran files. That is, F2PY can be used both to make large Fortran libraries callable from Python and to migrate slow loops in Python to Fortran code.

In the present case we may write some “inline” Fortran 77 code as follows:

```
f_code = """
      subroutine calculate_u(dt, dx, dy, u, um, up, k, n, m)
      integer m, n
      real*8 u(0:m, 0:n), um(0:m, 0:n)
      real*8 up(0:m, 0:n), k(0:m, 0:n)
      real*8 dt, dx, dy, hx, hy
      real*8 k_c, k_ip, k_im, k_jp, k_jm
      hx = (dt/dx)*(dt/dx)
      hy = (dt/dy)*(dt/dy)
Cf2py intent(in) u, up, k
Cf2py intent(out) up
      integer i, j
      do j = 1, n-1
         do i = 1, m-1
            k_c = k(i,j)
            k_ip = 0.5*(k_c + k(i+1,j))
            k_im = 0.5*(k_c + k(i-1,j))
            k_jp = 0.5*(k_c + k(i,j+1))
            k_jm = 0.5*(k_c + k(i,j-1))
            up(i,j) = 2*u(i,j) - um(i,j) +
     &        hx*(k_ip*(u(i+1,j) - u(i,j)) -
     &            k_im*(u(i,j) - u(i-1,j))) +
     &        hy*(k_jp*(u(i,j+1) - u(i,j)) -
     &            k_jm*(u(i,j) - u(i,j-1)))
         end do
      end do
      return
      end
"""
f2py.compile(f_code, modulename=’f2py_’, verbose=0)

import f2py_

def calculate_u(dt, dx, dy, u, um, up, k):
    up = f2py_.calculate_u(dt, dx, dy, u, um, up, k)
    return up
```

F2PY tries to make “Pythonic” interfaces to Fortran code, meaning that output arguments from a subroutine are returned to the user code. For example, if `o1` and `o2` are output arrays to be calculated in

```
       subroutine calc(o1, o2, i1, i2, m, n)
       integer m, n
       real*8 o1(m,n), o2(m,n), i1(m,n), i2(m,n)
```

F2PY will enable you to call `calc` as follows from Python:

```
o1, o2 = calc(i1, i2)
```

That is, output arguments are returned and array dimensions can be skipped since arrays in Python (here `i1` and `i2`) carry that information. However, Fortran has no syntax for distinguishing between input and output arguements. F2PY solves this problem through some special comment lines starting with `Cf2py`. By default, F2PY treats all arguments as input and none as output. In the subroutine `calculate_u` we must therefore specify `up` as `intent(out)` (as in Fortran 90) if we want it to be returned to the calling Python code. We have also for clarity specified the arrays that are input arguments. F2PY will also remove the `m` and `n` arguments from the argument list since these integers can automatically be extracted from the supplied arrays in the call. There are several other `intent` specifications that allow control over the memory usage when sending large arrays between Fortran and Python.

It is always important to traverse arrays in the way they are stored in (one-dimensional) memory. Two-dimensional C arrays are stored row by row, while Fortran applies a column by column storage. Therefore, we must be careful with the order of the loops in Fortran and remember to traverse our arrays column-wise. One may otherwise easily lose a factor of two in speed. Python, on the other hand, creates arrays with C storage by default. When NumPy arrays are to be sent to Fortran one must create them as, or convert them to, Fortran storage:

```
u = zeros((m, n), order='Fortran')
```

Otherwise, F2PY will make a copy of the array and transpose it for you, which may imply some significant overhead, especially when a Fortran routine is called many times in a time simulation as in our present example.

Cython

For all execution time results referred to in this section, we have used the arithmetic mean and a grid of size 250×250. Applying Cython is easy: we may cut the `calculate_u` function from the pure NumPy implementation above, paste it in a file with the suffix `.pyx`, and compile it. The speed-up with this configuration was minimal (about 10%). However, this demonstrates that the original Python `calculate_u` is also a valid Cython function.

To gain more speed, we need to utilize the power of the Cython language. A natural next step is to add types to the Cython function in the `.pyx` file, i.e., declaring all variables to be used as `float`s with the statement `cdef float dt` when declared within the function or simply as `float dt` when given as a keyword argument. This also includes the integer variables used for indexing arrays. The resulting speed-up of using static types is, unfortunately, not noticable, because accessing the arrays elements takes the majority of the execution time. However, specifying the types of variables is a necessary step (though not always sufficient) for fast code with Cython.

The second step is to take advantage of the integration between Cython and NumPy. Even though arrays are declared as `numpy.ndarray`, the `[]`-operator for accessing array elements is still implemented in Python. One needs to tell Cython about the type of the array elements. This is done with the syntax

```
numpy.ndarray[numpy.float_t, ndim=2] u
```

As a result, the indexing is now performed in C and the execution time is down to 0.3% of the pure NumPy implementation, or only about 5 times slower than the Fortran implementation.

To further speed things up, we need to turn off bounds checking. As a result, if you do try to index an array outside its bounds, the program will crash with a segmentation fault instead of an error. Hence, this should only be done when one has verified that the function works correctly. The run time is now halfed, and takes only 2.5 times that of the Fortran implementation.

Like with Python arrays, Cython allows negative indices, e.g., the syntax `u[-1]` accesses the last element of the array `u`. Turning off this feature may speed up the code (trying `u[-1]` will then most likely lead to a segmentation fault). One can either declare all indices as unsigned integers explicitly, and make sure to cast any indices that need to be computed, or one can add another option to the declaration of the arrays:

```
numpy.ndarray[numpy.float_t, ndim=2, negative_indices=False] u,
```

This reduces the CPU time further to 1.6

Cython can handle both arrays that are contiguous in memory and those that are not, as is the case if we perform slicing on an array. In case we only have arrays that are contiguous in memory, we can give the argument `mode=’c’` with this array, causing Cython to turn on further optimalizations. If we then send a sliced array as arguments, Cython will raise an exception.

Now, we have arrived at the optimal Cython version for our problem. The CPU time is now 1.3 (versus Fortran’s 1.0).

The complete Cython code is shown below, and is located in a file `cython_.pyx`:

```
import numpy as np
cimport numpy as np
cimport cython
DTYPE = np.float
ctypedef np.float_t DTYPE_t
@cython.boundscheck(False)
def calculate_u(float dt, float dx, float dy,
            np.ndarray[DTYPE_t, ndim=2, negative_indices=False, mode=’c’] u,
            np.ndarray[DTYPE_t, ndim=2, negative_indices=False, mode=’c’] um,
            np.ndarray[DTYPE_t, ndim=2, negative_indices=False, mode=’c’] up,
            np.ndarray[DTYPE_t, ndim=2, negative_indices=False, mode=’c’] k):
    cdef int m = u.shape[0]-1
    cdef int n = u.shape[1]-1
    cdef int i, j, start = 1
    cdef float k_c, k_ip, k_im, k_jp, k_jm
    cdef float hx = (dx/dt)**2
    cdef float hy = (dy/dt)**2
    for i in xrange(start, m):
        for j in xrange(start, n):
            k_c = k[i,j]
            k_ip = 0.5*(k_c + k[i+1,j])
            k_im = 0.5*(k_c + k[i-1,j])
            k_jp = 0.5*(k_c + k[i,j+1])
            k_jm = 0.5*(k_c + k[i,j-1])
            up[i,j] = 2*u[i,j] - um[i,j] +
              hx*(k_ip*(u[i+1,j] - u[i,j]) -
                  k_im*(u[i,j] - u[i-1,j])) +
              hy*(k_jp*(u[i,j+1] - u[i,j]) -
                  k_jm*(u[i,j] - u[i,j-1]))
    return up
```

Calling this code is done as follows:

```
import pyximport
pyximport.install()
import cython_

def calculate_u(dt, dx, dy, u, um, up, k):
    up = cython_.calculate_u(dt, dx, dy, u, um, up, k)
    return up
```

Trying to use a different form of indexing of NumPy arrays, i. e. `u[i][j]` or `u.item(i,j)`, instead of `u[i,j]`, was found to slow down the program to a speed similar to the initial Python program. This means that the integration with NumPy only works for the indexing `u[i,j]`.

Trying to vectorize the code also resulted in very poor performance, for the same reasons. Vectorization uses slicing, and slices are Python objects not implemented in Cython.

When changes are made to the Cython code, one would have to recompile the code before running the Python program again, but using the module `pyximport` eliminates the need for this step. Instead, code is only recompiled if necessary at import time.

Fortran

From these examples, it becomes clear that Python is a good alternative for coding even CPU-intensive numerical calculations. Combined with proper error messages and the fact that we can develop Python programs that, using extension modules for the most time consuming code, run at a speed only 10% slower than a pure Fortran program, shows us that although Python by itself is not fast enough for numerical calculations, simple steps can be taken that more than weigh up for this drawback.

References

[1] D. Beazley et al. SWIG: Simplified wrapper and interface generator *http://www.swig.org*.

[2] Boost C++ libraries *http://boost.org/libs*.

[3] X. Cai and H. P. Langtangen Parallelizing PDE solvers using the Python programming language In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 295–325. Springer, 2006.

[4] X. Cai, H. P. Langtangen and H. Moe On the performance of the Python programming language for serial and parallel scientific computations *Scientific Programming*, **vol.13**(1), 31–56, 2005.

[5] G. Ewing, R. Bradshaw, S. Behnel, D. S. Seljebotn et al. Cython: C-extensions for Python *http://cython.org*.

[6] E. Jones Weave: Tools for inlining C/C++ in Python *http://scipy.org/Weave*.

[7] E. Jones, T. Oliphant, P. Peterson et al. SciPy: Open source scientific tools for Python *http://scipy.org*.

[8] H. P. Langtangen *Python Scripting for Computational Science* Texts in Computational Science and Engineering, vol 3. Springer, third edition, 2009.

[9] H. P. Langtangen and X. Cai On the efficiency of Python for high-performance computing: A case study involving stencil updates for partial differential equations In H. G. Bock, E. Kostina, H. X. Phu and R. Rannacher, editors, *Modeling, Simulation and Optimization of Complex Processes*, pages 337–358. Springer, 2008.

[10] Netlib software collection. *http://netlib.org*.

[11] T. Oliphant et al. NumPy software package *http://numpy.org*.

[12] P. Peterson F2PY: Fortran to Python interface generator *http://scipy.org/F2py*.

[13] G. van Rossum et al. The Python programming language *http://python.org*.

[14] M. Westlie, K. A. Mardal and M. S. Alnæs Instant: Inlining of C/C++ in Python *http://fenics.org/instant*.