A Survey of Tools for Writing Faster Programs in Python

Robert Bassett

Naval Postgraduate School
OR Dept Seminar

March 10 2022

Motivation

Why should writing fast code should be prioritized?

Motivation

Why should writing fast code should be prioritized?

First point: Faster code permits scaling to larger, DoD-sized problems. Especially relevant in optimization.

Example: The algorithmic foundation for solving mixed integer linear programs quickly is...

Motivation

Why should writing fast code should be prioritized?

First point: Faster code permits scaling to larger, DoD-sized problems. Especially relevant in optimization.

Example: The algorithmic foundation for solving mixed integer linear programs quickly is...

The (dual) simplex method. Used for repeated solves of the relaxations in branch and bound.

Motivation

Why should writing fast code should be prioritized?

First point: Faster code permits scaling to larger, DoD-sized problems. Especially relevant in optimization.

Example: The algorithmic foundation for solving mixed integer linear programs quickly is...

The (dual) simplex method. Used for repeated solves of the relaxations in branch and bound.

Pop Quiz: Which solvers are fastest for LPs via simplex? Any in the **top 2** counts as a correct answer.

Answer[1]

No, it's not CPLEX or Gurobi.

Since 2018 the top 3 have been:

1. MindOpt (by Alibaba, a Chinese company).
2. COPT (by Cardinal Operations, a Chinese company).
3. Gurobi (the best among U.S. entries).

Newsflash: American dominance in optimization software is **over**.
It is not going to be regained by writing software that doesn't scale.

[1] Source: Mittleman Benchmarks, from Hans Mittleman of ASU Math Dept. Accessed March 4 2022.

Benchmark Details

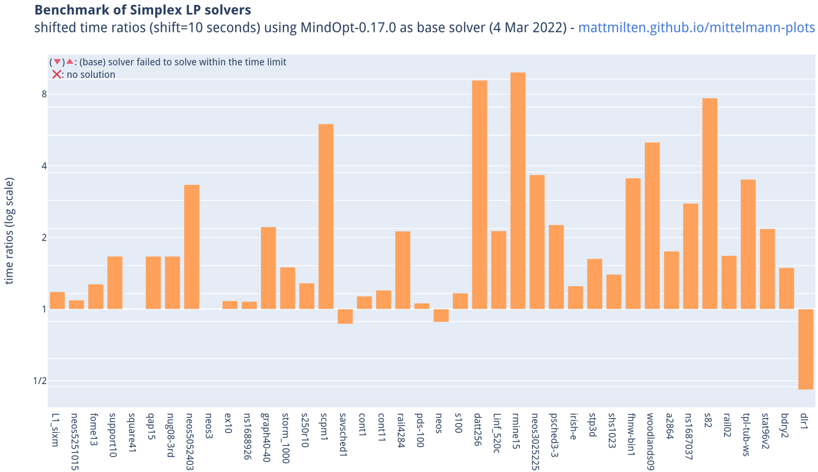


Figure: $\frac{\text{Gurobi-9.5.0 time}}{\text{MindOpt-0.17.0 time}}$ for problems in test suite.

More Motivation

Why should writing fast code should be prioritized?

[2] to project sponsors or publicly, when appropriate

More Motivation

Why should writing fast code should be prioritized?

Second point: Releasing[2] research code increases our impact. It lowers the barrier to other researchers *actually using* our work.

i.e. you can post it on Github.

[2] to project sponsors or publicly, when appropriate

More Motivation

Why should writing fast code should be prioritized?

Second point: Releasing[2] research code increases our impact. It lowers the barrier to other researchers *actually using* our work.

i.e. you can post it on Github.

Example: How many of us have used an `R` package without reading the white paper on which it was based?

You don't have to be a software engineer to identify low hanging fruit that makes your code run faster. This is important polish for a final product.

[2] to project sponsors or publicly, when appropriate

Python is Great

Python is a wonderful scripting language.

- Python is easy to learn.
- Python is fast to write.
- Python handles many low-level details (i.e. memory management) automatically.
- Python has numerous modules available to extend the language's functionality.
- Python has an enormous user base.
- Python has a business friendly BSD license[3].

[3] this is in contrast to R's GPL license

But Python is Slow

But these advantages come at a cost. Python is **not** fast.[4]

In many applications this is OK. It's often better to prioritize

- rapid prototyping
- readable code
- a developer's time

over writing code which is fast.

"Premature optimization is the root of all evil."

- Donald Knuth, author of *The Art of Computer Programming* and creator of TeX.

[4]Disclaimer: I mean native Python, not when used as a interface to fast code written in other languages, e.g. NumPy

But Python is Slow

The Computer Language Benchmarks Game is a database for comparing language performance.

Here is one numerics-heavy example problem, computing the largest absolute singular value of a matrix.

Timings of two programs on the same task.

Fortran: 0.72 seconds

Python: 112.97 seconds

Python is $\approx$ 156 times slower than Fortran for this task.

But Python is Slow

Python is slow because it is an *interpreted* language, not a *compiled* language.

- Interpreted languages use an *interpreter* to translate source code to CPU instructions *at run time*.
- Compiled languages use a *compiler* to translate source code into CPU instructions *at compile time* (i.e. before run time).

CPU instructions are not human readable but are very fast.

Also, the **Python interpreter does not support parallel execution**.

But Python is Slow

Python is slow because it is an *interpreted* language, not a *compiled* language.

- Interpreted languages use an *interpreter* to translate source code to CPU instructions *at run time*.
- Compiled languages use a *compiler* to translate source code into CPU instructions *at compile time* (i.e. before run time).

CPU instructions are not human readable but are very fast.

Also, the **Python interpreter does not support parallel execution**.

Analogy: You must travel from Monterey to Santa Cruz.

Compiled languages are like programming a route into GPS.

Interpreted languages are like asking for directions at each leg of the journey.

Outline

Getting to compiled code is *the key* to making it fast.

We will look at three ways to do it.

1. **Numba**: A just-in-time (JIT) compiler for Python. JIT compilers assess the input variables and compile the code during/after its first run.

2. **Cython**: An extension of the Python language. Developer provides additional information to Python syntax. Cython translates this into compiled code.

3. **f2py**: An extension of NumPy for calling Fortran functions from Python. Sounds daunting but isn't. Fortran is *very* easy to learn and quickly write fast code.

A Running Example

Swap the location of minimal and maximal elements in an array. Here's a Python implementation.

```python
def swap_min_max(arr):
    max_val = arr[0]
    max_ind = 0
    min_val = arr[0]
    min_ind = 0
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[min_ind] = arr[max_ind]
    arr[max_ind] = min_val
```

Note: you may be tempted to compute the min and max separately using built-in Python functions. But doing so loops through the array twice, whereas this only loops through once.

Python Timings

```
In [1]: X = np.array(range(int(1e8))) #100 million numbers

In [2]: %timeit python_version.swap_min_max(X)
20.7 s +/- 3.29 s per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)
```

Numba Version

```
from numba import jit

@jit(nopython=True)
def swap_min_max(arr):
    max_val = arr[0]
    max_ind = 0
    min_val = arr[0]
    min_ind = 0
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[min_ind] = max_val
    arr[max_ind] = min_val
```

It really is that easy. In Numba, decorators are used to identify functions that should be JIT compiled.

Numba Timings

First run (longer because it includes compilation):

```
In [1]: X = np.array(range(int(1e8))) #100 million numbers
In [2]: start = time.time(); numba_version.swap_min_max(X);
                 print(time.time() - start)
0.3200979232788086 (seconds)
```

Additional runs:

```
In [3]: %timeit numba_version.swap_min_max(X)
182 ms +/- 3.95 ms per loop
      (mean +/- std. dev. of 7 runs, 1 loops each)
```

Numba runtime is **113.7x faster** than Python runtime.

Parallel Numba Example

By modifying our decorator, we can also parallelize loops when appropriate.

Our example doesn't permit easy parallelization, because `max_val` and `min_val` can't be updated independently within each loop iteration.

But we can parallelize a big sum.

Parallel Numba Example

```
from numba import jit, prange

@jit(nopython=True, parallel=True)
def parallel_sum(arr):
    total = 0
    for i in prange(0, len(arr)):
        total += arr[i]
    return total

@jit(nopython=True)
def numba_sum(arr):
    total = 0
    for i in range(0, len(arr)):
        total += arr[i]
        return total
```

Eye Candy: Utilize your CPUs

Parallel Numba Timings

```
In [1]: np.random.seed(0)

In [2]: X = np.random.normal(size=int(1e9)) #1 billion numbers

In [3]: %timeit sum(X)
1min +/- 618 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [4]: %timeit numba_sum(X)
1.04 s +/- 36.2 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [5]: %timeit parallel_sum(X)
272 ms +/- 2.14 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)
```

Parallelized Numba is **220x** faster than sum in Python.

Downsides to Numba

The benefits of Numba are obvious: Much faster execution with minimal effort.

But there are some limitations.

- Numba only plays well with NumPy arrays and other elementary data types.
- JIT compiler means slow first-time execution.
- Wider Python ecosystem (SciPy, Pandas, etc) cannot be JIT compiled.

Introducing Cython

Cython is an extension of Python that provides enough additional information for the code to be **compiled**.

Compilation usually occurs when installing package, i.e. via **pip** or **conda**.

Cython code can also be compiled directly in a Jupyter notebook. Code should be **developed** but not **distributed** this way.

Good news: All valid Python code is valid Cython code. But providing additional information via Cython's unique syntax is what gives you speed improvements.

Cython Version (No Change From Python)

```
def swap_min_max_cython(arr, n):
    max_val = arr[0]
    max_ind = 0
    min_val = arr[0]
    min_ind = 0
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

14.5 s runtime compared to 20.7 s in original. **1.43x Faster**.

Cython Version: Typing all the Variables

```
cimport numpy as np

def swap_min_max_cython(np.ndarray[ndim=1, dtype=np.int64_t] arr, int n):
    cdef int max_val = arr[0]
    cdef int max_ind = 0
    cdef int min_val = arr[0]
    cdef int min_ind = 0
    cdef int i
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

150 ms runtime compared to 20.7 s in original. **138.0x Faster**.

Cython Version: Ints (32 bit) vs Longs (64 bit)

```
cimport numpy as np

def swap_min_max_cython(np.ndarray[ndim=1, dtype=np.int64_t] arr, int n):
    cdef long max_val = arr[0]
    cdef int max_ind = 0
    cdef long min_val = arr[0]
    cdef int min_ind = 0
    cdef int i
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

125 ms runtime compared to 20.7 s in original. **165.6x Faster**.

Cython Version: Drop NumPy Dependency

```
#dtype[::1] means a contiguous chunk of memory dtype
def swap_min_max_cython(long[::1] arr, int n):
    cdef long max_val = arr[0]
    cdef int max_ind = 0
    cdef long min_val = arr[0]
    cdef int min_ind = 0
    cdef int i
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

86.3 ms runtime compared to 20.7 s in original. **239.9x Faster**.

Cython Version: Tricks with Indices

```
cimport cython

@cython.boundscheck(False) #disable index checking
@cython.wraparound(False) #forbid negative indices
def swap_min_max_cython(long[::1] arr, int n):
    cdef long max_val = arr[0]
    cdef int max_ind = 0
    cdef long min_val = arr[0]
    cdef int min_ind = 0
    cdef Py_ssize_t i #special type for indexing Python arrays
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

77.3 ms runtime compared to 20.7 s in original. **267.8x Faster**.

Cython Timings

```
In [1]: X = np.array(range(int(1e8))) #100 million numbers

In [2]: %timeit python_version.swap_min_max(X)
20.7 s +/- 3.16 s per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [3]: %timeit swap_min_max_cython(X, len(X))
77.3 ms +/- 487 µs per loop
        (mean +/- std. dev. of 7 runs, 10 loops each)
```

Cython is **267.8x times faster** than Python at 20.7 seconds.

Cython Annotation

What if we miss adding some important Cython syntax?

Using Jupyter magic `%%cython -a` can help us find it.

Hypothetical: We forget to type `max_ind` in our running example.

Cython Annotation: Missed Type

Generated by Cython 0.29.21

Yellow lines hint at Python interaction.

Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
cimport cython
cimport numpy as np

@cython.boundscheck(False)
@cython.wraparound(False)
def swap_min_max_cython(long[::1] arr, int n):
    max_val = arr[0]
    cdef int max_ind = 0
    cdef long min_val = arr[0]
    cdef int min_ind = 0
    cdef Py_ssize_t i
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

Cython Annotation: Fixed

Generated by Cython 0.29.21

Yellow lines hint at Python interaction.

Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
cimport cython
cimport numpy as np

@cython.boundscheck(False)
@cython.wraparound(False)
def swap_min_max_cython(long[::1] arr, int n):
    cdef long max_val = arr[0]
    cdef int max_ind = 0
    cdef long min_val = arr[0]
    cdef int min_ind = 0
    cdef Py_ssize_t i
    for i in range(1, n):
        if arr[i] > max_val:
            max_val = arr[i]
            max_ind = i
        if arr[i] < min_val:
            min_val = arr[i]
            min_ind = i
    arr[max_ind] = min_val
    arr[min_ind] = max_val
```

Cython: Parallelism

Parallelizing code is also extremely easy in Cython.

A serialized sum:

```
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def sum_cython(double[::1] arr, int n):
    cdef double total = 0.0
    cdef Py_ssize_t i
    for i in range(n):
        total += arr[i]
    return total
```

Cython: Parallelism

Parallelizing code is also extremely easy in Cython.

A parallelized version:[5]

```
cimport cython
from cython.parallel import prange, parallel

@cython.boundscheck(False)
@cython.wraparound(False)
def parallel_sum_cython(double[::1] arr, int n):
    cdef double total = 0.0
    cdef Py_ssize_t i
    for i in prange(n, nogil=True):
        total += arr[i]
    return total
```

[5] You must add openmp to Jupyter magic syntax to parallelize in a notebook, i.e. `%%cython --compile-args=-fopenmp --link-args=-fopenmp --force`

Cython Parallel Performance

```
In [1]: np.random.seed(0)

In [2]: X = np.random.normal(size=int(1e9)) #1 billion numbers

In [3]: %timeit sum(X)
1min +/- 618 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [4]: %timeit sum_cython(X, len(X))
998 ms +/- 4.63 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [5]: %timeit parallel_sum_cython(X, len(X))
264 ms +/- 3.37 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)
```

Parallelized Cython is **227x faster** than Python.

Pros and Cons of Cython

Pros of Cython

- Fast. No drawbacks of JIT compiler.
- Portable. Pip and Conda installations will compile Cython.
- Versatile. Easily connect Python to C/C++ libraries.

Cons of Cython

- Often assumes developer understands conventions of C/C++.
- Cython only plays well with NumPy arrays and other elementary data types.
- Wider Python ecosystem (SciPy, Pandas, etc) can be included in Cython code, but you won't see speed gains.[6]

[6] Exception: SciPy provides alternative lapack & blas functions directly for Cython.

Introducing f2py

- `f2py` is an extension of NumPy that allows convenient calling of Fortran code from Python.
- Fortran isn't as widely used as C/C++ today. Perceived as primarily used in legacy code.
- Fortran is **amazing** for simple programs which are heavy on numerics. It is a domain-specific language for numerical computation, like R is a domain-specific language for statistics.
- Fortran can be an great resource for eliminating bottlenecks in Python code.
- It's easy to learn, and easy to write fast code.
- `f2py` makes linking Fortran functions to Python extremely easy.

Return to Running Example

```fortran
! file: fortran_version.f90
subroutine swap_min_max(arr, n)
    implicit none !don't use default variable definitions
    integer n, min_ind, max_ind, i
    integer*8 max_val, min_val
    !f2py integer intent(hide) depend(arr):: n = shape(arr,0)
    integer*8 arr(n) !integer*8 gives 64 bit integer i.e. long
    min_ind = 1 !fortran is 1-indexed instead of 0-indexed like Python
    max_ind = 1
    max_val = arr(1) !fortran uses () instead of [] to index arrays
    min_val = arr(1)
    do i=1,n !indents don't matter in Fortran, but help readability
        if (arr(i) > max_val) then
            max_val = arr(i)
            max_ind = i
        end if
        if (arr(i) < min_val) then
            min_val = arr(i)
            min_ind = i
        end if
    end do
    arr(min_ind) = max_val
    arr(max_ind) = min_val
end
```

Compiling and Running

```
[me@computer]$ f2py -c -m fortran_version fortran_version.f90

[me@computer]$ ipython
In [1]: import numpy as np; import python_version

In [2]: import fortran_version

In [3]: X = np.array(range(int(1e8)))

In [4]: %timeit python_version.swap_min_max(X)
20.7 s +/- 3.16 s per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [5]: %timeit fortran_version.swap_min_max(X)
95.5 ms +/- 391 µs per loop
        (mean +/- std. dev. of 7 runs, 10 loops each)
```

Fortran with `f2py` is **216.8x** faster than Python.

Parallel Computation with Fortran/f2py

Again, let's parallelize a large sum. Serial version:

```
! file parallel_sum.f90
subroutine fortran_sum(arr, n, total)
    implicit none
    integer n, i
    !f2py integer intent(hide) depend(arr):: n = shape(arr, 0)
    real*8, intent(in):: arr(n) !fortran intents are parsed by f2py
    real*8, intent(out):: total
    total = 0d0
    do i=1,n
        total = total + arr(i)
    end do
end
```

Parallel Computation with Fortran/f2py

Again, let's parallelize a large sum. Parallel version:

```
! file parallel_sum.f90
subroutine fortran_sum(arr, n, total)
    implicit none !don't use default variable definitions
    integer n, i
    !f2py integer intent(hide) depend(arr):: n = shape(arr, 0)
    real*8, intent(in):: arr(n) !fortran intents are parsed by f2py
    real*8, intent(out):: total
    total = 0d0
    !$omp parallel do reduction(+:total)
    do i=1,n
        total = total + arr(i)
    end do
    !$omp end parallel do
end
```

Fortran gives access to **openmp**, a powerful tool for parallelization.

f2py: Parallel Performance

```
[me@computer]$ f2py -c -m parallel_sum parallel_sum.f90

[me@computer]$ ipython
In [1]: import numpy as np; import parallel_sum

In [2]: np.random.seed(0); X = np.random.normal(size=int(1e9))

In [3]: %timeit sum(X)
1min +/- 618 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [4]: %timeit parallel_sum.fortran_sum(X)
1.03 s +/- 35.8 ms per loop
        (mean +/- std. dev. of 7 runs, 1 loop each)

In [5]: %timeit parallel_sum.fortran_sum_parallel(X)
273 ms +/- 9.08 ms per loop
        (mean +/- std. dev. of 7 runs, 10 loops each)
```

Parallel Fortran/f2py is **219.8x** faster than Python.

Pros and Cons of f2py

Pros of f2py

- Easily call fast Fortran from Python. Eliminate bottlenecks.
- Compiling Fortran into a Python module is a single execution of f2py.
- Fortran is easy to learn. Can also be ported to other languages (R, Julia) easily.
- Fortran is a complete language, with its own fast libraries (BLAS, LAPACK, ScaLAPACK, etc.)

Cons of f2py

- Requires learning some details of yet another language.
- Fortran/f2py only play well with NumPy arrays and other elementary data types.
- Wider Python ecosystem (SciPy, Pandas, etc) cannot be utilitzed within Fortran.

Other Tools

Natural question: What about tools for linking Python with C/C++, similar to `f2py`'s use of Fortran?

These tools exist. Most popular is `pybind11`, with `swig` an older alternative. Python packages like `cppimport` automate the complilation step.

My opinion:

- These tools are **not** designed for Python developers looking to eliminate bottlenecks.
- These tools are best used for C/C++ developers looking to provide a Python interface.
- They are complicated, and require a lot of troubleshooting to get them working.
- C/C++ have conventions which are (a) not geared towards code performance (b) difficult if you primarily work in Python.

Conclusion

We've introduced three tools for writing faster Python programs.

1. Numba
   - **Low effort**. Only requires function decorators.
   - Must be **JIT compiled**, so first runs are not fast.
   - Low amount of developer control, because entire process is automated.
2. Cython
   - **Medium effort**. Takes familiar Python syntax and modifies it to produce very fast code.
   - Portable and universal. Used in libraries like SciPy, Scikit-learn, and Statsmodels.
   - Medium amount of developer control. Flexible within **constraints of Cython and C/C++ interaction**.
3. f2py
   - **Higher effort**. Requires some familiarity with Fortran.
   - Can be ported to other languages besides Python more easily.
   - High amount of developer control, because you have all **Fortran syntax and libraries** at your disposal.

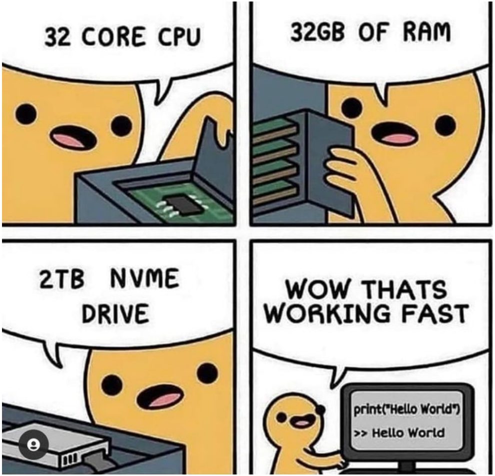
For More Information

1. I'm always available to chat.

2. These slides and the source code for all examples are at my website: faculty.nps.edu/rbassett

3. Documentation (these are links).
   - **Numba**
   - **Cython**
   - **f2py**

Happy Coding!



Cartoon source: @code_memez on twitter