

Effective Programming Practices for Economists

Scientific Computing

Writing fast code with numba

Janoś Gabler and Hans-Martin von Gaudecker

Why numba can be fast

- Python code is just-in-time compiled to machine code
- It builds on top of numpy
 - Efficient storage of data in arrays
 - All dtypes known at compile-time
- Uses same technology as Julia under the hood

Implications

- Loops can be super fast
- Avoids some of the drawbacks of vectorized code
- However: Not guaranteed to be faster than numpy

Numba is picky

- Need to write out all loops
- Works best if all your data is in scalars or arrays
 - No dicts, lists or NamedTuples
- Inside compiled functions you cannot call uncompiled functions
 - Cannot call most libraries
 - Sometimes need to re-implement algorithms in numba compatible ways

Therefore, numba is intended to speed up your bottlenecks, not to compile your entire program

Naively jiting the example

```
from numba import njit
```

```
@njit
```

```
def array_cobb_douglas(factors, weights, a):
```

```
    out = np.empty(len(factors))
```

```
    for i in range(len(factors)):
```

```
        out[i] = _cobb_douglas(factors[i], weights, a)
```

```
    return out
```

```
@njit
```

```
def _cobb_douglas(factors, weights, a):
```

```
    return a * np.prod(factors**weights)
```

```
# (inputs as before)
```

```
%timeit array_cobb_douglas(factors, weights, a)
```

```
1.22 ms ± 39.6 µs per loop
```

- The function is numba compatible but will not be optimal because not all loops are written out
- It still becomes much faster than before (~20× speedup)

Full loops

```
@njit
def numba_array_cobb_douglas(factors, weights, a):
    out = np.empty(len(factors))
    for i in range(len(factors)):
        out_i = a
        for j in range(len(weights)):
            out_i *= factors[i, j]**weights[j]
        out[i] = out_i
    return out

%timeit numba_array_cobb_douglas(factors, weights, a)

602 µs ± 3.62 µs per loop
```

- Writing out loops yields gets us another $\sim 2\times$ speedup ($\sim 40\times$ over original)
- Here, still slower than vectorized numpy but this is not always the case!

