C(++)? Fortran? Python? Cython!

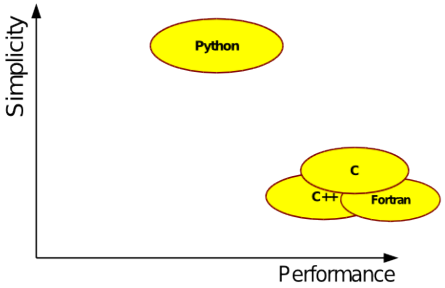
Make code run up to 1000 x faster in only 5 minutes

Arne Meyer

Gatsby Unit, Tea talk

12th February 2015

Background



- C(++) is very fast but often inconvenient for research (especially plotting)
- Interpreted languages (here: Python) are excellent for research but in some cases very slow
- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python

Background

Donald Knuth (1974)

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.**"

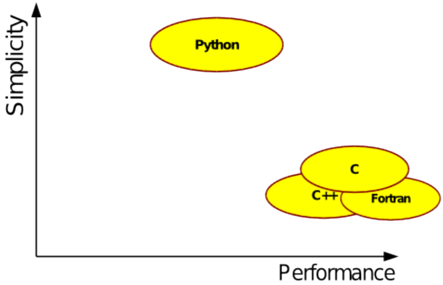
Background

Donald Knuth (1974)

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.**"

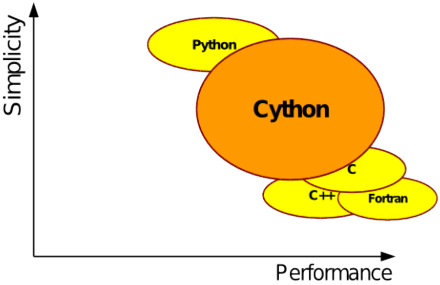
- Re-writing code is often very time-consuming and prone to errors
- How to optimize the critical 3% efficiently?

Background



- C(++) is very fast but often inconvenient for research (especially plotting)
- Interpreted languages (here: Python) are excellent for research but in some cases very slow
- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python

Background



- C(++) is very fast but often inconvenient for research (especially plotting)
- Interpreted languages (here: Python) are excellent for research but in some cases very slow
- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python
- Cython: combines the best of both worlds

Cython at a glance

- Open-source project: `www.cython.org`
- An optimizing compiler for the Python language
- Very active development
- Rapidly growing user base (many from science)

Use-cases:

1. Compiling Python code to machine-code
    - Supports a big subset of the Python language
    - Runs about 30% faster than plain Python code
2. Add types for speedups (hundreds of times)
    - Optimize, don't re-write!
3. Easily use native libraries (C/C++/Fortran) directly
    - There are better tools, e.g., SWIG

Example

Ridge regression using stochastic gradient descent

Goal: minimize

$$\frac{1}{2}\sum_i \left(y_i - \mathbf{x}_i^T\mathbf{w}\right)^2 + \frac{1}{2}\alpha\|\mathbf{w}\|^2$$

Pseudo code:

input: $\{\mathbf{x}_i, y_i\}$, α, N_{iter}

$\mathbf{w} \leftarrow \mathbf{0}$

for $t = 1, 2, ..., N_{\text{iter}}$ **do**

  $\mathbf{x}_i, y_i \leftarrow$ draw random sample

  $\gamma \leftarrow \frac{1}{\alpha t}$

  $\mathbf{w} \leftarrow \mathbf{w} - \gamma\alpha\mathbf{w}$

  $\mathbf{w} \leftarrow \mathbf{w} - \gamma\mathbf{x}_i^T\left(y_i - \mathbf{x}_i^T\mathbf{w}\right)$

end

Naive Python implementation

```
def ridge_sgd_naive(X, y, w, alpha, perm):

    D = X.shape[1]
    for t, i in enumerate(perm):

        gamma = 1. / (1 + alpha*t)

        # regularization step
        for j in range(D):
            w[j] *= (1. - gamma * alpha)

        # loss step
        z = 0
        for j in range(D):
            z += w[j] * X[i, j]

        for j in range(D):
            w[j] += gamma * X[i, j] * (z - y[i])
```

Naive Python implementation

```python
def ridge_sgd_naive(X, y, w, alpha, perm):

    D = X.shape[1]
    for t, i in enumerate(perm):

        gamma = 1. / (1 + alpha*t)

        # regularization step
        for j in range(D):
            w[j] *= (1. - gamma * alpha)

        # loss step
        z = 0
        for j in range(D):
            z += w[j] * X[i, j]

        for j in range(D):
            w[j] += gamma * X[i, j] * (z - y[i])
```

- Python: approx. 135 s
- Cython: approx. 97 s

```python
import pyximport
pyximport.install()

from cython_file import cython_function
...
```

Vectorized (Numpy) implementation

```
import numpy as np

def ridge_sgd_vectorized(X, y, w, alpha, perm):

    for t, i in enumerate(perm):

        gamma = 1. / (1 + alpha*t)

        # regularization step
        w *= (1. - gamma * alpha)

        # loss step
        z = np.dot(w, X[i, :])
        w += gamma * X[i, :] * (z - y[i])
```

Vectorized (Numpy) implementation

```python
import numpy as np

def ridge_sgd_vectorized(X, y, w, alpha, perm):

    for t, i in enumerate(perm):

        gamma = 1. / (1 + alpha*t)

        # regularization step
        w *= (1. - gamma * alpha)

        # loss step
        z = np.dot(w, X[i, :])
        w += gamma * X[i, :] * (z - y[i])
```

- Python: approx. 1.65 s
- Cython: approx. 1.44 s

Cython: adding static types to naive implementation

```
def ridge_sgd_cython_types(np.ndarray[np.float64_t, ndim=2] X,
                           np.ndarray[np.float64_t, ndim=1] y,
                           np.ndarray[np.float64_t, ndim=1] w, double alpha,
                           np.ndarray[np.int64_t, ndim=1] perm):

    cdef int D = X.shape[1]
    cdef int i, j, t
    cdef double gamma, z

    for t, i in enumerate(perm):

        gamma = 1. / (1. + alpha*t)

        # regularization step
        for j in range(D):
            w[j] *= (1. - gamma * alpha)

        # loss step
        z = 0
        for j in range(D):
            z += w[j] * X[i, j]

        for j in range(D):
            w[j] += gamma * X[i, j] * (z - y[i])
```

Cython: adding static types to naive implementation

```
def ridge_sgd_cython_types(np.ndarray[np.float64_t, ndim=2] X,
                           np.ndarray[np.float64_t, ndim=1] y,
                           np.ndarray[np.float64_t, ndim=1] w, double alpha,
                           np.ndarray[np.int64_t, ndim=1] perm):

    cdef int D = X.shape[1]
    cdef int i, j, t
    cdef double gamma, z

    for t, i in enumerate(perm):

        gamma = 1. / (1. + alpha*t)

        # regularization step
        for j in range(D):
            w[j] *= (1. - gamma * alpha)

        # loss step
        z = 0
        for j in range(D):
            z += w[j] * X[i, j]

        for j in range(D):
            w[j] += gamma * X[i, j] * (z - y[i])
```

- Run time: approx. 0.33 s

Cython: static types and C pointers

```
def ridge_sgd_cython_pointers(np.ndarray[np.float64_t, ndim=2] X,
                              np.ndarray[np.float64_t, ndim=1] y,
                              np.ndarray[np.float64_t, ndim=1] w, double alpha,
                              np.ndarray[np.int64_t, ndim=1] perm):

    cdef int D = X.shape[1]
    cdef int i, j, t
    cdef double gamma, z

    cdef double *Xp = <double*> X.data
    cdef double *yp = <double*> y.data
    cdef double *wp = <double*> w.data
    cdef long *pp = <long*> perm.data

    for t, i in enumerate(perm):

        ...

        for j in range(D):
            z += wp[j] * Xp[i*D + j]

        for j in range(D):
            wp[j] += gamma * Xp[i*D + j] * (z - yp[i])
```

Cython: static types and C pointers

```
def ridge_sgd_cython_pointers(np.ndarray[np.float64_t, ndim=2] X,
                              np.ndarray[np.float64_t, ndim=1] y,
                              np.ndarray[np.float64_t, ndim=1] w, double alpha,
                              np.ndarray[np.int64_t, ndim=1] perm):

    cdef int D = X.shape[1]
    cdef int i, j, t
    cdef double gamma, z

    cdef double *Xp = <double*> X.data
    cdef double *yp = <double*> y.data
    cdef double *wp = <double*> w.data
    cdef long *pp = <long*> perm.data

    for t, i in enumerate(perm):

        ...

        for j in range(D):
            z += wp[j] * Xp[i*D + j]

        for j in range(D):
            wp[j] += gamma * Xp[i*D + j] * (z - yp[i])
```

- Run time: approx. 0.24 s
- Replacing loops by BLAS functions: approx. 0.18 s

Numba JIT implementation

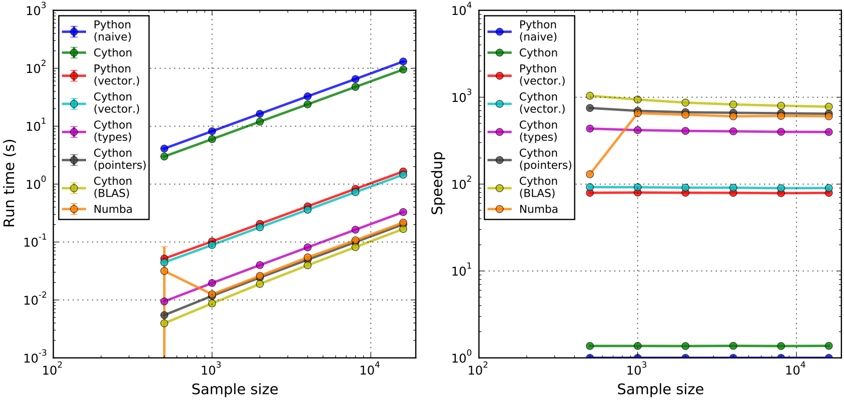
```
from numba.decorators import autojit

ridge_sgd_numba = autojit(ridge_sgd_vectorized)
```

- http://numba.pydata.org/
- Just-in-time (JIT) compiler
- Run time: approx. 0.22 s

Summary

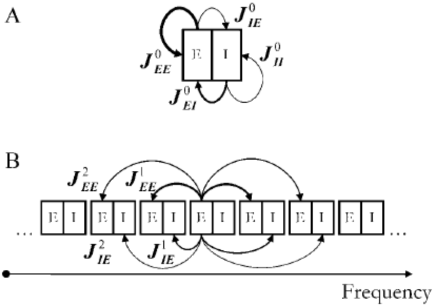
Stochastic gradient descent



- Cython about 400 – 1000 times faster than naive Python
- Cython about 5 – 10 times faster than (vectorized) Numpy
- Comparable to Numba

Example 2

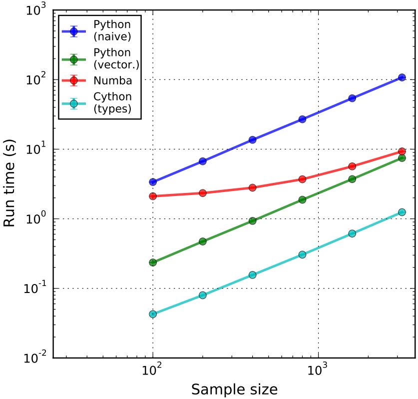
Recurrent neural network

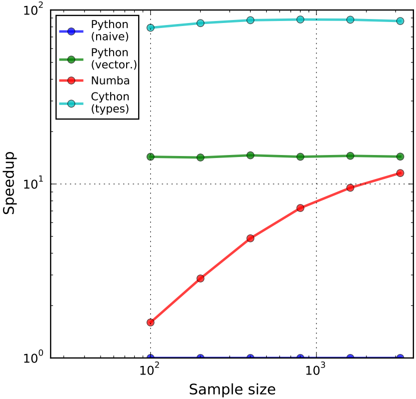


- Loebel & Tsodyks (2007)
- 15 coupled EI networks (cortical columns)
- Each column: $N_E = 100$, $N_I = 100$
- External stimulus input

Example 2

Recurrent neural network





- Cython about 85 times faster than naive Python
- Cython about 7 times faster than (vectorized) Numpy
- JIT compiler (Numba) much slower than Cython version

What was that all about?

- Goal: writing fast code in interpreted language
- Avoid unneccessary re-writing of (working) code
- Cython: simply add static types to existing (Python) code
- Only a few extra lines (about 5 minutes ...)
- Speedup: 50-1000 times (naive Python), 1-250 times (vectorized Numpy)
- In some cases, JIT compilers (e.g., Numba) may help, too