Kostenloses eBook

LERNEN

cython

Free unaffiliated eBook created from
Stack Overflow contributors.

#cython

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cython

It is an unofficial and free cython ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cython.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Cython

Bemerkungen

Was ist Cython?

Die Programmiersprache Cython bereichert Python durch C-artige statische Typisierung, die Möglichkeit, C-Funktionen direkt aufzurufen, und viele andere Funktionen. Dadurch kann die Leistung auf C-Ebene erreicht werden, während eine Python-artige Syntax verwendet wird.

Wie funktioniert es?

Cython-Code wird mit dem Cython-Quelle-zu-Quelle-Compiler kompiliert, um C- oder C ++ - Code zu erstellen, der wiederum mit einem C-Compiler kompiliert werden kann. Dies ermöglicht das Erstellen von Erweiterungen, die von Python oder ausführbaren Dateien importiert werden können.

Der Hauptleistungsgewinn, den Cython im Gegensatz zu reinem Python erzielen kann, ergibt sich aus der Umgehung der CPython-API. Wenn Sie zum Beispiel zwei Ganzzahlen hinzufügen, führt Python eine Typüberprüfung für jede Variable durch, sucht eine Add-Funktion, die die gefundenen Typen erfüllt, und ruft diese Funktion auf. Im durch Cython generierten C-Code sind die Typen bereits bekannt und es wird nur ein Funktionsaufruf an durchgeführt. Daher ist Cython besonders für mathematische Probleme geeignet, bei denen die Typen klar sind.

Wie kann ich meinen Code beschleunigen?

Ein häufiger Anwendungsfall beim Versuch, ein Programm mit Cython zu beschleunigen, besteht darin, den Code zu profilieren und die rechenintensiven Teile in kompilierte Cython-Module zu verschieben. Dadurch kann die Python-Syntax für den Großteil des Codes beibehalten und die Beschleunigung dort angewendet werden, wo sie am dringendsten benötigt wird.

Examples

Cython installieren

Für die Verwendung von Cython sind zwei Dinge erforderlich. Das Cython-Paket selbst, das den `cython` Source-to-Source-Compiler und die Cython-Schnittstellen zu mehreren C- und Python-Bibliotheken enthält (z. B. numpy). Zum Kompilieren des vom `cython` Compiler generierten C-Codes wird ein C-Compiler benötigt.

Schritt 1: Cython installieren

Systemagnostiker

Cython kann mit mehreren systemunabhängigen Paketverwaltungssystemen installiert werden. Diese schließen ein:

1. PyPI via pip oder easy_install:

```
$ pip install cython
$ easy_install cython
```

2. Anakonda mit Conda:

```
$ conda install cython
```

3. Überdachte Canopy mit dem enpkg-Paketmanager:

```
$ enpkg cython
```

Der Quellcode kann auch von github heruntergeladen und manuell installiert werden:

```
$ python setup.py install
```

Ubuntu, Debian

Für Ubuntu die Pakete `cython` und `cython3` stehen zur Verfügung. Beachten Sie, dass diese Versionen eine ältere Version als die oben genannten Installationsoptionen bieten.

```
$ apt-get install cython cython3
```

Windows

Für Windows wird eine .whl-Datei , die mit pip installiert werden kann, von einem Drittanbieter bereitgestellt. Details zum Installieren einer .whl-Datei unter Windows finden Sie hier .

Schritt 2: Installation eines C-Compilers

Zum Kompilieren der von Cython generierten C-Dateien ist ein Compiler für C und C ++ erforderlich. Der gcc-Compiler wird empfohlen und kann wie folgt installiert werden.

Ubuntu, Debian

Das `build-essential` Paket enthält alles, was benötigt wird. Es kann aus den Repositorys installiert werden mit:

```
$ sudo apt-get install build-essential
```

MAC

Die XCode-Entwicklerwerkzeuge enthalten einen gcc-ähnlichen Compiler.

Windows

MinGW (Minimalist GNU for Windows) enthält eine Windows-Version von gcc. Der Compiler von Visual Studio kann ebenfalls verwendet werden.

Hallo Welt

Eine Cython Pyx-Datei muss in C-Code übersetzt (*cythonisiert*) und kompiliert werden, bevor sie von Python verwendet werden kann. Ein gängiger Ansatz ist das Erstellen eines Erweiterungsmoduls, das dann in ein Python-Programm importiert wird.

Code

Für dieses Beispiel erstellen wir drei Dateien:

- `hello.pyx` enthält den Cython-Code.
- `test.py` ist ein Python-Skript, das die Hallo-Erweiterung verwendet.
- `setup.py` wird zum Kompilieren des Cython-Codes verwendet.

hallo.pyx

```
from libc.math cimport pow

cdef double square_and_add (double x):
    """Compute x^2 + x as double.

    This is a cdef function that can be called from within
    a Cython program, but not from Python.
    """
    return pow(x, 2.0) + x

cpdef print_result (double x):
    """This is a cpdef function that can be called from Python."""
    print("({} ^ 2) + {} = {}".format(x, x, square_and_add(x)))
```

test.py

```
# Import the extension module hello.
import hello

# Call the print_result method
```

```
hello.print_result(23.0)
```

setup.py

```
from distutils.core import Extension, setup
from Cython.Build import cythonize

# define an extension that will be cythonized and compiled
ext = Extension(name="hello", sources=["hello.pyx"])
setup(ext_modules=cythonize(ext))
```

Kompilieren

Dazu können Sie den Code mit `cython hello.pyx` nach C übersetzen und dann mit `gcc` kompilieren. Ein einfacher Weg ist es, Distutils damit umgehen zu lassen:

```
$ ls
hello.pyx  setup.py  test.py
$ python setup.py build_ext --inplace
$ ls
build  hello.c  hello.cpython-34m.so  hello.pyx  setup.py  test.py
```

Die Shared Object-Datei (.so) kann aus Python importiert und verwendet werden, sodass wir jetzt die `test.py` :

```
$ python test.py
(23.0 ^ 2) + 23.0 = 552.0
```

Erste Schritte mit Cython online lesen: https://riptutorial.com/de/cython/topic/2925/erste-schritte-mit-cython

Kapitel 2: C-Code einwickeln

Examples

Verwenden von Funktionen aus einer benutzerdefinierten C-Bibliothek

Wir haben eine C-Bibliothek namens `my_random` , die Zufallszahlen aus einer benutzerdefinierten Verteilung erzeugt. Es bietet zwei Funktionen, die wir verwenden möchten: `set_seed(long seed)` und `rand()` (und viele mehr brauchen wir nicht). Um sie in Cython verwenden zu können, müssen wir das tun

1. Definieren Sie eine Schnittstelle in der .pxd-Datei und
2. Rufen Sie die Funktion in der .pyx-Datei auf.

Code

test_extern.pxd

```
# extern blocks define interfaces for Cython to C code
cdef extern from "my_random.h":
    double rand()
    void c_set_seed "set_seed" (long seed) # rename C version of set_seed to c_set_seed to
avoid naming conflict
```

test_extern.pyx

```
def set_seed (long seed):
    """Pass the seed on to the c version of set_seed in my_random."""
    c_set_seed(seed)

cpdef get_successes (int x, double threshold):
    """Create a list with x results of rand <= threshold

    Use the custom rand function from my_random.
    """
    cdef:
        list successes = []
        int i
    for i in range(x):
        if rand() <= threshold:
            successes.append(True)
        else:
            successes.append(False)
    return successes
```

C-Code einwickeln online lesen: https://riptutorial.com/de/cython/topic/3626/c-code-einwickeln

Kapitel 3: Cython-Bündelung

Examples

Cython-Programm mit pyinstaller bündeln

Beginnen Sie mit einem Cython-Programm mit einem Einstiegspunkt:

```
def do_stuff():
    cdef int a,b,c
    a = 1
    b = 2
    c = 3
    print("Hello World!")
    print([a,b,c])
    input("Press Enter to continue.")
```

Erstellen Sie eine `setup.py` Datei im selben Ordner:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    name = "Hello World",
    ext_modules = cythonize('program.pyx'),
)
```

`python setup.py build_ext --inplace` es mit `python setup.py build_ext --inplace` , `.pyd` `python setup.py build_ext --inplace` eine `.pyd` Bibliothek in einem Unterordner.

Anschließend erstellen Sie ein Vanilla-Python-Skript mit Hilfe der Bibliothek (z. B. `main.py`) und platzieren die `.pyd` Datei daneben:

```
import program
program.do_stuff()
```

Verwenden Sie PyInstaller, um es `pyinstaller --onefile "main.py"` zu bündeln. Dadurch wird ein Unterordner erstellt, der die ausführbare Datei mit einer Größe von mindestens 4 MB enthält, die die Bibliothek sowie die Python-Laufzeit enthält.

Build automatisieren (Windows)

Zur Automatisierung des obigen Verfahrens in Windows verwenden Sie eine `.bat` mit ähnlichen Inhalten:

```
del "main.exe"
python setup.py build_ext --inplace
del "*.c"
rmdir /s /q ".\build"
pyinstaller --onefile "main.py"
```

```
copy /y ".\dist\main.exe" ".\main.exe"
rmdir /s /q ".\dist"
rmdir /s /q ".\build"
del "*.spec"
del "*.pyd"
```

Numpy zum Bundle hinzufügen

Um dem Paket Numpy hinzuzufügen, ändern Sie die `setup.py` mit `include_dirs` Schlüsselwort `include_dirs` und importieren Sie den numpy-Code im Wrapper-Python-Skript, um Pyinstaller zu benachrichtigen.

`program.pyx` :

```
import numpy as np
cimport numpy as np


def do_stuff():
    print("Hello World!")
    cdef int n
    n = 2
    r = np.random.randint(1,5)
    print("A random number: "+str(r))
    print("A random number multiplied by 2 (made by cdef):"+str(r*n))
    input("Press Enter to continue.")
```

`setup.py` :

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

setup(
    ext_modules=cythonize("hello.pyx"),
    include_dirs=[numpy.get_include()]
)
```

`main.py` :

```
import program
import numpy
program.do_stuff()
```

Cython-Bündelung online lesen: https://riptutorial.com/de/cython/topic/6386/cython-bundelung

Kapitel 4: Umschließen von C ++

Examples

Umschließen einer DLL: C ++ in Cython in Python

Dies zeigt ein nicht triviales Beispiel für das Umschließen einer C ++ - DLL mit Cython. Es wird die folgenden Hauptschritte abdecken:

- Erstellen Sie eine Beispiel-DLL mit C ++, die Visual Studio verwendet.
- Wickeln Sie die DLL mit Cython, damit sie in Python aufgerufen werden kann.

Es wird davon ausgegangen, dass Sie Cython installiert haben und es erfolgreich in Python importieren können.

Für den DLL-Schritt wird außerdem davon ausgegangen, dass Sie mit dem Erstellen einer DLL in Visual Studio vertraut sind.

Das vollständige Beispiel umfasst die Erstellung der folgenden Dateien:

1. `complexFunLib.h` : Header-Datei für die C ++ - DLL-Quelle
2. `complexFunLib.cpp` : CPP-Datei für die C ++ - DLL-Quelle
3. `ccomplexFunLib.pxd` : Cython "Header" -Datei
4. `complexFunLib.pyx` : Cython-Wrapper-Datei
5. `setup.py` : Python-Setup-Datei zum Erstellen von `complexFunLib.pyd` mit Cython
6. `run.py` : Beispiel für eine Python-Datei, die die kompilierte, mit Cython `run.py` DLL importiert

C ++ - DLL-Quelle: `complexFunLib.h` und `complexFunLib.cpp`

Überspringen Sie dies, wenn Sie bereits eine DLL- und Header-Quelldatei haben. Zuerst erstellen wir die C ++ - Quelle, aus der die DLL mit Visual Studio kompiliert wird. In diesem Fall möchten wir schnelle Array-Berechnungen mit der komplexen Exponentialfunktion durchführen. Die folgenden zwei Funktionen führen die Berechnung `k*exp(ee)` für die Arrays `k` und `ee` , wobei die Ergebnisse in `res` gespeichert werden. Es gibt zwei Funktionen, um sowohl einfache als auch doppelte Genauigkeit zu ermöglichen. Beachten Sie, dass diese Beispielfunktionen OpenMP verwenden. Stellen Sie daher sicher, dass OpenMP in den Visual Studio-Optionen für das Projekt aktiviert ist.

H Datei

```
// Avoids C++ name mangling with extern "C"
#define EXTERN_DLL_EXPORT extern "C" __declspec(dllexport)
#include <complex>
#include <stdlib.h>

// Handles 64 bit complex numbers, i.e. two 32 bit (4 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c4(std::complex<float>* k,
```

```
                                    std::complex<float>* ee,
                                    int sz,
                                    std::complex<float>* res,
                                    int threads);

// Handles 128 bit complex numbers, i.e. two 64 bit (8 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c8(std::complex<double>* k,
std::complex<double>* ee,
                                    int sz,
                                    std::complex<double>* res,
                                    int threads);
```

CPP-Datei

```
#include "stdafx.h"
#include <stdio.h>
#include <omp.h>
#include "complexFunLib.h"

void mp_mlt_exp_c4(std::complex<float>* k,
                   std::complex<float>* ee,
                   int sz,
                   std::complex<float>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}

void mp_mlt_exp_c8(std::complex<double>* k,
                   std::complex<double>* ee,
                   int sz, std::complex<double>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}
```

Cython Source: `ccomplexFunLib.pxd` und `complexFunLib.pyx`

Als Nächstes erstellen wir die Cython-Quelldateien, die zum Einschließen der C ++ - DLL erforderlich sind. In diesem Schritt treffen wir folgende Annahmen:

- Sie haben Cython installiert
- Sie besitzen eine Arbeits-DLL, zB die oben beschriebene

Das ultimative Ziel besteht darin, diese Cython-Quelldateien in Verbindung mit der Original-DLL zu erstellen, um eine `.pyd` Datei zu erstellen, die als Python-Modul importiert werden kann und die in

C ++ geschriebenen Funktionen `.pyd` macht.

PXD-Datei

Diese Datei entspricht der C ++ - Headerdatei. In den meisten Fällen können Sie die Kopfzeile mit geringfügigen Cython-spezifischen Änderungen kopieren und in diese Datei einfügen. In diesem Fall wurden die spezifischen Cython-Komplextypen verwendet. Beachten Sie das Hinzufügen von `c` am Anfang von `ccomplexFunLib.pxd` . Dies ist nicht notwendig, aber wir haben festgestellt, dass eine solche Namenskonvention zur Aufrechterhaltung der Organisation beiträgt.

```
cdef extern from "complexFunLib.h":
    void mp_mlt_exp_c4(float complex* k, float complex* ee, int sz,
                       float complex* res, int threads);
    void mp_mlt_exp_c8(double complex* k, double complex* ee, int sz,
                       double complex* res, int threads);
```

PYX-Datei

Diese Datei entspricht der C ++ `cpp` - Quelldatei. In diesem Beispiel übergeben wir Zeiger auf Numpy `ndarray` Objekte an die Import-DLL-Funktionen. Es ist auch möglich, das eingebaute Cython- `memoryview` Objekt für Arrays zu verwenden, seine Leistung ist jedoch möglicherweise nicht so gut wie bei `ndarray` Objekten (die Syntax ist jedoch wesentlich sauberer).

```
cimport ccomplexFunLib  # Import the pxd "header"
# Note for Numpy imports, the C import most come AFTER the Python import
import numpy as np  # Import the Python Numpy
cimport numpy as np  # Import the C Numpy

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *

# Python wrapper functions.
# Note that types can be delcared in the signature

def mp_exp_c4(np.ndarray[np.complex64_t, ndim=1] k,
              np.ndarray[np.complex64_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex64_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    # Call the imported DLL functions on the parameters.
    # Notice that we are passing a pointer to the first element in each array
    ccomplexFunLib.mp_mlt_exp_c4(&k[0], &ee[0], sz, &res[0], threads)

def mp_exp_c8(np.ndarray[np.complex128_t, ndim=1] k,
              np.ndarray[np.complex128_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex128_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    ccomplexFunLib.mp_mlt_exp_c8(&k[0], &ee[0], sz, &res[0], threads)
```

Python-Quelle: `setup.py` und `run.py`

setup.py

Diese Datei ist eine Python-Datei, die die Cython-Kompilierung ausführt. Der Zweck besteht darin, die kompilierte `.pyd` Datei zu generieren, die dann von Python-Modulen importiert werden kann. In diesem Beispiel haben wir alle erforderlichen Dateien (z. B. `complexFunLib.h` , `complexFunLib.dll` , `ccomplexFunLib.pxd` und `complexFunLib.pyx`) im gleichen Verzeichnis wie `setup.py` .

Nachdem diese Datei erstellt wurde, sollte sie über die Befehlszeile mit den Parametern ausgeführt werden: `build_ext --inplace`

Nach dem `.pyd` dieser Datei sollte eine `.pyd` Datei erstellt werden, ohne dass Fehler auftreten. Beachten Sie, dass in einigen Fällen, wenn ein Fehler `.pyd` die `.pyd` kann, aber ungültig ist. `setup.py` Sie sicher, dass bei der Ausführung von `setup.py` keine Fehler `setup.py` bevor Sie das generierte `.pyd` .

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy as np

ext_modules = [
    Extension('complexFunLib',
              ['complexFunLib.pyx'],
              # Note here that the C++ language was specified
              # The default language is C
              language="c++",
              libraries=['complexFunLib'],
              library_dirs=['.'])
    ]

setup(
    name = 'complexFunLib',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules,
    include_dirs=[np.get_include()]  # This gets all the required Numpy core files
)
```

run.py

Jetzt kann `complexFunLib` direkt in ein Python-Modul importiert und die `complexFunLib` DLL-Funktionen aufgerufen werden.

```
import complexFunLib
import numpy as np

# Create arrays of non-trivial complex numbers to be exponentiated,
# i.e. res = k*exp(ee)
k = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
ee = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
sz = k.size  # Get size integer
res = np.zeros(int(2.5e5), dtype='complex64')  # Create array for results
```

```
# Call function
complexFunLib.mp_exp_c4(k, ee, sz, res, 8)

# Print results
print(res)
```

Umschließen von C ++ online lesen: https://riptutorial.com/de/cython/topic/3525/umschlie-en-von-c-plusplus

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Cython	Community, J.J. Hakala, Keith L, m00am
2	C-Code einwickeln	m00am
3	Cython-Bündelung	Andrii Magalich
4	Umschließen von C ++	J.J. Hakala, Keith L, Kevin Pasquarella