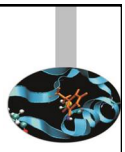
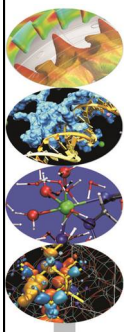
CINECA SCAI
SuperComputing Applications and Innovation



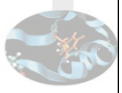
Performance in Python:
mixed Language Programming



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation



Sommario

1. Introduzione:
   a. elementi di *profiling* di codice Python
   b. Python vs (C/C++ & Fortran)
   c. Estendere Python
2. F2PY
3. Cython
4. Un caso reale: equazione delle onde in 2D

CINECA SCAI
SuperComputing Applications and Innovation

Python optimization strategy

Per migliorare le prestazioni di un codice Python esistono diverse strategie:

- Utilizzare la libreria NumPy e vettorizzare il codice
- Utilizzare pacchetti ottimizzati per task specifici
- **Integrazione con linguaggi più di basso livello: F2PY, Cython, …**
- Parellelizzazione
- Porting su GPGPU

3

CINECA SCAI
SuperComputing Applications and Innovation

Timing your code …

- Prima di ottimizzare è necessario rilevare gli hot-spot del nostro codice
- Il modulo *timeit:*
  - consente di misurare il tempo di esecuzione di una funzione o di una espressione;
  - È adatto per test molto piccoli e particolarmente veloci (micro-sec)

```
import timeit as t

def testF():
    a=[]
    for el in xrange(1000000): a.append(el)

if __name__ == '__main__':

   print t.Timer('for el in range(10000000):pass').timeit(1),'s'

   print t.Timer('for el in xrange(10000000):pass').timeit(1),'s'

   print t.Timer('testF()',
                 setup='from __main__ import testF').timeit(1),'s'
```

4

CINECA SCAI
SuperComputing Applications and Innovation

… Timing your code …

timeit output

```
1.120997905730 s
0.394932985306 s
0.228574037552 s
```

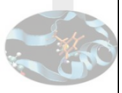
```
import timeit as t

def testF():
    a=[]
    for el in xrange(1000000): a.append(el)

if __name__ == '__main__':

   print t.Timer('for el in range(10000000):pass').timeit(1),'s'

   print t.Timer('for el in xrange(10000000):pass').timeit(1),'s'

   print t.Timer('testF()',
                 setup='from __main__ import testF').timeit(1),'s'
```

Il modulo *timeit* può anche essere utilizzato da *command-line*

```
$ python -m timeit -n 1 "for el in range(1000000):pass“

$ python -m timeit -n 1 ”for el in xrange(1000000):pass”

$ python -m timeit -n 1 -s “import timeit” -s “import mymodule”
“mymodule.testF”
```


CINECA SCAI
SuperComputing Applications and Innovation

Timing your code: `cProfile` …



- Il modulo `cProfile` è lo strumento standard di Python per il *profiling* di un codice.
- Vediamo un semplice esempio:

```
import cProfile

def testF():
   a=[]
   for el in xrange(1000000): a.append[el]

if __name__ == '__main__':
   cProfile.run('testF()')
```

CINECA SCAI
SuperComputing Applications and Innovation

… Timing your code: cProfile

OUTPUT:

```
1000003 function calls in 0.960 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.017    0.017    0.960    0.960 <string>:1(<module>)
      1    0.600    0.600    0.943    0.943 cProf.py:5(testF)
1000000    0.343    0.000    0.343    0.000 {method 'append' of ..}
      1    0.000    0.000    0.000    0.000 {method 'disable' of ..}
```

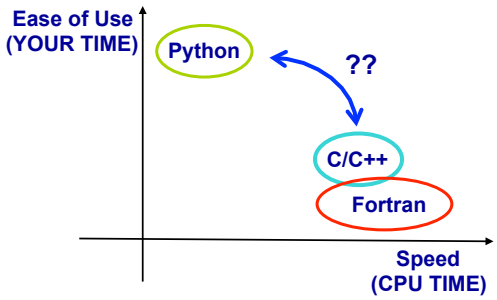
Come nel caso del modulo `timeit`, anche `cProfile` Può essere utilizzato da linea di comando:

$ python -m cProfile [-s order_string] [-o my_outfile] mymodule.py

7

CINECA SCAI
SuperComputing Applications and Innovation

Python vs. (C/C++ or Fortran)



C/C++ & Fortran:
- **High performance**
- **Basso livello**
- **Tipizzazione statica**

wrapper
(comm. layer)

Python:
- **Low performance**
- **Alto livello**
- **Tipizzazione dinamica**

11

CINECA SCAI
SuperComputing Applications and Innovation

Python & (*C/C++ or Fortran*)

- Fare *Mixed Programming* in Python con codici C/C++ o Fortran è utile in almeno due casi:
  - *accesso a codici già esistenti*: durante lo sviluppo di un nuovo codice Python, sarebbe utile accedere facilmente a librerie (o comunque a frammenti di codice) già esistenti e scritti in Fortran e/o C/C++
  - *migrazione di codice lento*: nello sviluppo di nuovo codice, dopo aver realizzato un prototipo interamente scritto in Python, ne migriamo gli hot-spot in un linguaggio più adatto ad attività HPC (C/C++ o Fortran)

- Con questo "modello di sviluppo" è possibile sfruttare la potenza di ogni linguaggio nel proprio territorio più naturale:
  - la potenza di Python nella gestione della complessità (alto livello)
  - la potenza di C/C++ o Fortran nella gestione delle prestazioni (basso livello)

12

CINECA SCAI
SuperComputing Applications and Innovation

`ctypes`: utilizzare una libreria C

- Spesso dobbiamo importare una libreria (C) già esistente, e non pensata per essere importata in Python.
- Non è troppo complicato farlo, ci sono vari strumenti; il modulo `ctypes` consente di chiamare una funzione di una libreria C

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary("libc.so.6")

>>> libc.time(None)
1378848259

>>> libc.time(None)
1378848261

>>> libc.printf("%s\n", "Hello World!")
Hello, World!
14
```

13

CINECA SCAI
SuperComputing Applications and Innovation

"Estendere" Python

- È possibile creare moduli Python in due maniere:
  - scrivendo codice Python puro (come abbiamo fatto fino a questo momento)
  - scrivendo codice in altri linguaggi (C, C++, FORTRAN, ...) e facendone il *binding* in modo che sia importabile come modulo Python.
- Dal punto di vista dell'uso di questi moduli, non c'è alcuna differenza!

- La maniera nativa per scrivere un nuovo modulo Python in C consiste nell'usare le Python C-API (http://docs.python.org/c-api/)
  - si tratta di scrivere nuovo codice C attraverso l'uso delle API C di Python (creano la struttura necessaria ad importare gli oggetti e le funzioni come modulo all'interno di Python stesso)
  - **È abbastanza complicato!**

14

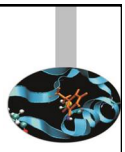
CINECA SCAI
SuperComputing Applications and Innovation

Alternative a Python C-API

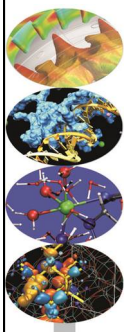
- F2PY:
  - è un *tool* che permette di costruire un modulo che interfaccia Python a funzioni scritte in Fortran e C.
  - è parte della libreria *NumPy* e, quindi, non necessità di installazione, se *NumPy* è già installato.

- Cython (http://www.cython.org/):
  - è una estensione di Python pensata per consentire facilmente di estendere Python con moduli scritti in altri linguaggi
  - aggiunge a Python una sintassi in cui è possibile dichiarare variabili, funzioni ed oggetti C/C++, ovvero è una sorta di *superset* di Python
  - crea automaticamente il *binding* di funzioni (oggetti) C (C++) usando le Python C-API;
  - produce codice C/C++ ottimizzato, che può essere compilato per generare un modulo Python.

CINECA SCAI
SuperComputing Applications and Innovation



Mixed language programming: introduzione a F2PY



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

f2py at work: routine Fortran (1)

```
! FILE: fib.f90

  subroutine fib(a,n)
     implicit none
     integer :: n, i
     real*8  :: a(*)
     do i=1,n
        if (i .eq. 1) then
           a(i) = 1.0
        elseif (i .eq. 2) then
           a(i) = 1.0
        else
           a(i) = a(i-1)+a(i-2)
        endif
     enddo
  end
```

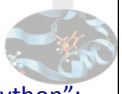
Ricetta per usarla in Python

```
$ f2py –m myFib fib.f90
$ python –c 'import myFib'

>>> import myFib
>>> print myFib.fib.__doc__
fib - Function signature  fib(a,n)
Required arguments:
a : input rank-1 array('d') with
    bounds (*)
n : input int

>>> import numpy as np
>>> a = np.zeros(4)
>>> myFib.fib(a,4)
>>> a
array([ 1.,  1.,  2.,  3.])
```

17

SCAI
SuperComputing Applications and Innovation

f2py at work: routine Fortran (2)

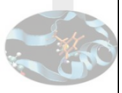


- Si può fare di meglio e generare un'interfaccia al modulo più "in stile Python":
  - "in stile Python" il modulo si dovrebbe usare così: `a=myFib.fib(4)`
  - Il comando f2py, utilizzato con l'opzione –h, genera un file di *signature* con la descrizione dell'interfaccia del modulo

```
$ f2py –h myFib.pyf –m myFib fib.f90
$ cat myFib.pyf
! Note: the context of this file is case sensitive.

python module myFib ! in
    interface  ! in :myFib
        subroutine fib(a,n) ! in :myFib:fib.f90
            real*8 dimension(*) :: a
            integer :: n
        end subroutine fib
    end interface
end python module myFib

! This file was auto-generated with f2py (version:2).
```


SCAI
SuperComputing Applications and Innovation

f2py at work: routine Fortran (3)

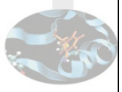


Interveniamo manualmente sulla descrizione dell'interfaccia generata da f2py, definendo quali variabili sono di input e di output e le relative dipendenze, attraverso le clausole `intent(in)`, `intent(out)` e `depend()`

```
$ cat myFib2.pyf

! Note: the context of this file is case sensitive.

python module myFib ! in
    interface  ! in :myFib
        subroutine fib(a,n) ! in :myFib:fib.f90
            real*8 dimension(n), intent(out), depend(n) :: a
            integer intent(in) :: n
        end subroutine fib
    end interface
end python module myFib
```

CINECA SCAI
SuperComputing Applications and Innovation

f2py at work: routine Fortran (4)



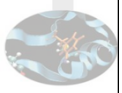
Dopo l'intervento sul file che descrive l'interfaccia del nuovo modulo Python, usiamolo per ri-compilare il sorgente Fortran:

```
$ f2py –c –m myFib myFib2.pyf fib.f90
$ python

>>> import myFib

>>> myFib.fib.__doc__
fib - Function signature:
a = fib(n)
Required arguments:
n : input int
Return objects:  a : rank-1 array('d') with bounds (n)

>>> myFib.fib(4)
array([ 1.,  1.,  2.,  3.])
```

20

CINECA SCAI
SuperComputing Applications and Innovation

f2py at work: routine Fortran (5)



Le direttive per f2py possono essere inserite direttamente nel sorgente Fortran

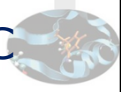
```
! FILE: fib2.f90

  subroutine fib(a,n)
     implicit none
     integer :: n,i
     real*8  dimension(n) :: a

     !f2py real*8 dimension(n), intent(out), depend(n) :: a
     !f2py integer intent(in) :: n

     do i=1,n
        if (i .eq. 1) then
           a(i) = 1.0
        elseif (i .eq. 2) then
           a(i) = 1.0
        else
           a(i) = a(i-1)+a(i-2)
        endif
     enddo
  end
```

Comando di compilazione:
$ f2py –c –m myFib fib2.f90

21

CINECA SCAI
SuperComputing Applications and Innovation

F2PY: modulo Python da funzioni C



- Seppur progettato per generare moduli Python a partire da funzioni e subroutine Fortran, F2PY può essere utilizzato anche per creare moduli da funzioni scritte in C.
- Vediamone un esempio d'uso; dato il sorgente C

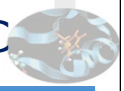
```
// File hwC.c
#include<stdio.h>
#include<math.h>

double hw1 (double r1, double r2){
     double s;
     s=sin(r1+r2);
     return s;
}

void hw2(double r1, double r2){
     double s;
     s=sin(r1+r2);
     printf("sin(%g+%g) = %g\n",r1,r2,s);
}
```

bisogna creare manualmente un *signature file*, che descriva le interfacce ed in cui le funzioni ed i suoi argomenti abbiano anche l'attributo `intent(c)`

22

CINECA SCAI
SuperComputing Applications and Innovation

F2PY: modulo Python da funzioni C



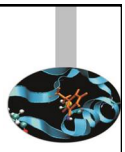
```
//! File hwC.pyf

python module hwC
  interface
    function hw1(r1,r2)
      intent(c) hw1
      intent(c) r1,r2
      real*8 r1,r2
    end function hw1
  end
  interface
    subroutine hw2(r1,r2)
      intent(c) hw2
      intent(c) r1,r2
      real*8 r1,r2
    end subroutine hw2
  end interface
end python module hwC
```

Compiliamo ed utilizziamo il nuovo modulo

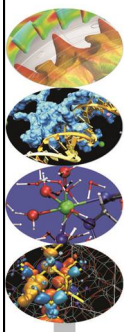
```
$ f2py -c -m hwC hwC.pyf hwC.c
$ python –c 'import hwC'

>>> import numpy as np
>>> import hwC

>>> np.sin(np.pi+0.5*np.pi)
-1.0

>>> a = hwC.hw1(np.pi,0.5*np.pi)

>>> a
-1.0

>>> hwC.hw2(np.pi,0.5*np.pi)
sin(3.14159+1.5708) = -1
```

23

CINECA SCAI
SuperComputing Applications and Innovation



Mixed language programming:
introduzione a Cython



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Velocizzare codice Python: l'approccio di Cython …

- il progetto Cython affronta il problema delle performance del codice Python per mezzo di un compilatore di codice sorgente che traduce il codice Python in codice C equivalente
- Il codice prodotto :
  - viene eseguito all'interno dell'ambiente di *runtime* di Python, ma alla velocità di un un codice C compilato e con la possibilità di chiamare direttamente in librerie C
  - mantiene l'interfaccia originale del codice sorgente Python, cosa che lo rende direttamente utilizzabile dal codice Python.
- Queste caratteristiche sono alla base dei 2 principali casi d'uso di Cython:
  - estensione dell'interprete Python con moduli binari veloci
  - interfacciamento di codice Python con librerie C esterne

26

SCAI SuperComputing Applications and Innovation

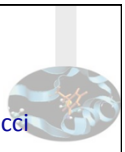
…Velocizzare codice Python: l'approccio di Cython

- Cython può compilare la maggior parte dei sorgenti Python
- il codice C generato presenta miglioramenti di velocità (talvolta impressionanti) prevalentemente a causa ***tipizzazione statica*** (opzionale) che può essere applicata, nel sorgente Cython, sia alle variabili C che a quelle Python
- Pertanto in Cython è possibile assegnare la semantica C a parti del proprio codice, in modo che queste possano essere tradotte in codice C estremamente veloce
- La *type declarations* può essere utilizzata per 2 scopi:
  - trasformare sezioni di codice dalla "semantica dinamica" di Python alla "semantica statica e veloce" del C
  - utilizzare i tipi definiti in librerie esterne

27

SCAI SuperComputing Applications and Innovation

Cython by example



La funzione che calcola il valore l'ennesimo numero della serie di Fibonacci

La versione Python

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

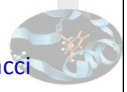
La versione C/C++

```
int fib(int n) {
    int tmp, i, a, b;
    a = b = 1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

28

CINECA SCAI SuperComputing Applications and Innovation

Cython by example



La funzione che calcola il valore l'ennesimo numero della serie di Fibonacci

La versione Python

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

La versione C/C++

```
int fib(int n) {
    int tmp, i, a, b;
    a = b = 1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

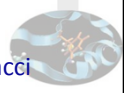
La versione Cython

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

29

CINECA SCAI SuperComputing Applications and Innovation

Cython by example



La funzione che calcola il valore l'ennesimo numero della serie di Fibonacci

La versione Python **1x**

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

La versione C/C++ **70x**

```
int fib(int n) {
    int tmp, i, a, b;
    a = b = 1;
    for(i=0; i<n; i++) {
        tmp = a; a += b; b = tmp;
    }
    return a;
}
```

La versione Cython **70x**

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

30

CINECA SCAI
SuperComputing Applications and Innovation

Workflow per "*velocizzare*" un codice Python

Python

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Cython

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Codice C generato da Cython

```
static PyObject
*__pyx_pf_5cyfib_cyfib(PyObject
*__pyx_self, int __pyx_v_n) {
  int __pyx_v_a; int __pyx_v_b;
PyObject *__pyx_r = NULL; PyObject
*__pyx_t_5 = NULL;
const char *__pyx_filename = NULL;
...
  for (__pyx_t_1=0;
__pyx_t_1<__pyx_t_2; __pyx_t_1+=1)
{
    __pyx_v_i = __pyx_t_1;
    __pyx_t_3 = (__pyx_v_a +
__pyx_v_b);
    __pyx_t_4 = __pyx_v_a;
    __pyx_v_a = __pyx_t_3;
    __pyx_v_b = __pyx_t_4;
  }
...
}
```

32

CINECA SCAI
SuperComputing Applications and Innovation

Wrapping di codice C / C++

C/C++

```
int fact(int n) {
    if (n <= 1)
      return 1;
    return n * fact(n-1);
}
```

Cython

```
cdef extern from "fact.h":
    int _fact "fact"(int)

def fact(int n):
    return _fact(n)
```

Codice C generato da Cython

```
static PyObject
*__pyx_pf_5cyfib_cyfib(PyObject
*__pyx_self, int __pyx_v_n) {
  int __pyx_v_a; int __pyx_v_b;
PyObject *__pyx_r = NULL; PyObject
*__pyx_t_5 = NULL;
const char *__pyx_filename = NULL;
...
  for (__pyx_t_1=0;
__pyx_t_1<__pyx_t_2; __pyx_t_1+=1)
{
    __pyx_v_i = __pyx_t_1;
    __pyx_t_3 = (__pyx_v_a +
__pyx_v_b);
    __pyx_t_4 = __pyx_v_a;
    __pyx_v_a = __pyx_t_3;
    __pyx_v_b = __pyx_t_4;
  }
...
}
```

33

SCAI
SuperComputing Applications and Innovation

Cython + IPython

All'interno di *IPython* è possibile utilizzare i *magic command* per *Cython*, il più utile dei quali è senz'altro **%%cython**

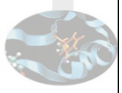
Usare *Cython* all'interno di *IPython*

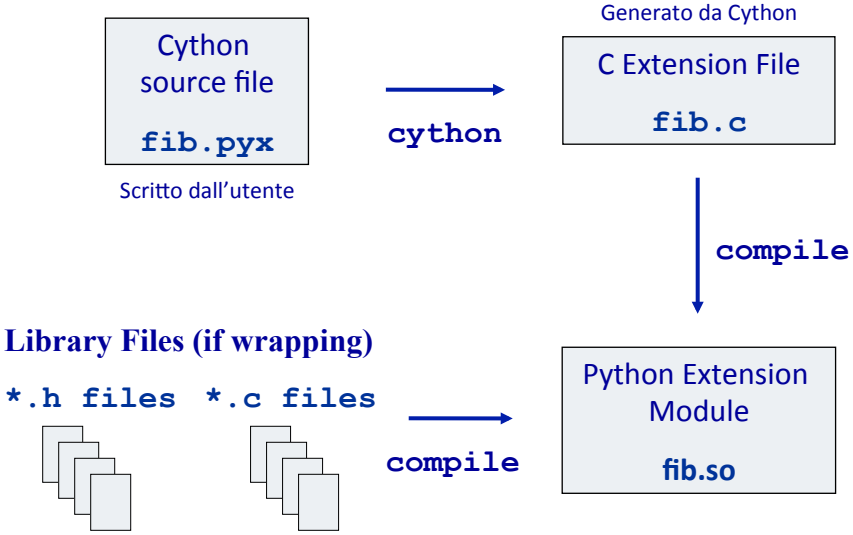
```
In [10]: %load_ext cythonmagic

In [11]: %%cython
    ....: def cyfib(int n):
    ....:     cdef int a, b, i
    ....:     a, b = 1, 1
    ....:     for i in range(n):
    ....:         a, b = a+b, a
    ....:     return a
    ....:

In [12]: cyfib(10)
Out[12]: 144
```

34

SCAI
SuperComputing Applications and Innovation

Cython pyx file





35

SCAI SuperComputing Applications and Innovation

Compilare Cython con *distutils*

1. Definiamo una funzione, con le necessarie *type declaration*

Il file `fib.pyx`

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

2. Costruiamo un opportuno file per il setup del modulo (Cython ha il proprio *extension builder*, che è in grado di costruire un modulo Python a partire dal sorgente Cython)

Il file `setup_fib.py`

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

ext = Extension("myFib",
                sources=["fib.pyx"])

setup(ext_modules=[ext],
      cmdclass={'build_ext': build_ext})
```

36

SCAI SuperComputing Applications and Innovation

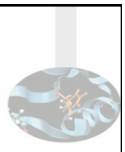
Compilare ed utilizzare l'*extension module*

```
$ python setup_fib.py build_ext --inplace

$ python

>>> import myFib

>>> myFib.fib()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)

>>> myFib.fib("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required

>>> myFib.fib(10)
144
```

37

CINECA SCAI
SuperComputing Applications and Innovation

Un utile strumento: `pyximport`



- `pyximport` consente di importare un sorgente Cython come se fosse un modulo Python puro
- `pyximport` rileva eventuali cambiamenti nel file Cython, lo ricompila se necessario, altrimenti lo legge dalla cache dei moduli
- E' molto utile nei casi semplici

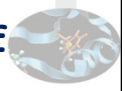
`pyximport` at work

```
>>> import pyximport

>>> pyximport.install()     # prepara l'environment

>>> from fib import fib     # trova fib.pyx e lo compila

>>> fib(10)
144
```

38

CINECA SCAI
SuperComputing Applications and Innovation

Dichiarare "oggetti" C in Cython : `cdef`



Dichiarare variabili locali

```
def fib(int n):
    cdef int i, a, b
    ...
```

Dichiarare funzioni C

```
cdef float distance(float *x, float *y, int n):
    cdef:
       int i
       float d = 0.0
    for i in range(n):
       d += (x[i] - y[i])**2
    return d
```

NB: Gli argomenti delle *typed function* sono dichiarati senza `cdef`

39

SCAI SuperComputing Applications and Innovation

Funzioni `def`, `cdef` e `cpdef`

```
def distance(x, y):
    return np.sum((x-y)**2)
```

Una **`def`** ***function*** è una funzione Python utilizzabile sia in Python che in Cython

```
cdef float distance(float *x, float *y, int n):
    cdef:
        int i
        float d = 0.0
    for i in range(n):
        d += (x[i] - y[i])**2
    return d
```

Una **`cdef`** ***function*** è una funzione tradotta in C ed utilizzabile solo all'interno del file Cython (non direttamente da Python)

```
cpdef float distance(float[:] x, float[:] y):
    cdef int i
    cdef int n = x.shape[0]
    cdef float d = 0.0
    for i in range(n):
        d += (x[i] - y[i])**2
    return d
```

Una **`cpdef`** ***function*** è una funzione tradotta in C ed è utilizzabile sia all'interno del file Cython sia nel codice Python che importa il modulo che la contiene

40

SCAI SuperComputing Applications and Innovation

`def` *function*: un esempio

Il sorgente cython: `def` — Funzioni Python

```
# funzione chiamabile da Python
def inc(int num, int offset):
   return num + offset

# Chiamare la funzione inc su una sequenza di valori
def inc_seq(seq, offset):
   result = []
   for val in seq:
       res = inc(val, offset)
       result.append(res)
   return result
```

Usiamo il modulo `inc` all'interno di Python

```
# la funzione inc è direttamente utilizzabile
>>> inc.inc(1,3)
4
>>> a = range(4)
>>> inc.inc_seq(a, 3)
[3,4,5,6]
```

41

SCAI SuperComputing Applications and Innovation

`cdef` *function*: un esempio

Il sorgente cython: `cdef` — Funzioni C

```
# con l'uso di cdef diventa una funzione C
cdef int fast_inc(int num, int offset):
    return num + offset

# # Chiamare fast_inc su una sequenza di valori
def fast_inc_seq(seq, offset):
   result = []
   for val in seq:
       res = fast_inc(val, offset);
       result.append(res)
   return result
```

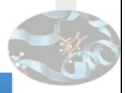
Usiamo il modulo `inc` all'interno di Python

```
# fast_inc non è chiamabile in Python
>>> inc.fast_inc(1,3)
Traceback: ... no 'fast_inc'
# Ma fast_inc_seq è 2 volte più veloce per grandi array
>>> a = range(4); inc.fast_inc_seq(a, 3)
[3,4,5,6]
```

42

SCAI SuperComputing Applications and Innovation

`cdef` *function*: un esempio



Il sorgente cython: `cpdef` — Funzioni sia C che Python

```
# con l'uso di cdef diventa una funzione sia C che Python
cpdef int fast_inc(int num, int offset):
    return num + offset

# All'interno del file cython, viene chiamata la versione C
def fast_inc_seq(seq, offset):
   result = []
   for val in seq:
       res = fast_inc(val, offset);
       result.append(res)
   return result
```

Usiamo il modulo `inc` all'interno di Python

```
# fast_inc ora è chiamabile attraverso il wrapper Python
>>> inc.fast_inc(1,3)
4
# Ma non c'è perdità di velocità nella versione per sequenze
>>> a = range(4); inc.fast_inc_seq(a, 3)
[3,4,5,6]
```

43

CINECA SCAI SuperComputing Applications and Innovation

Accesso alle funzioni della *C standard library*: `cimport`

Supponiamo che nel nostro codice Cython si debba calcolare il seno di uno scalare:

- Se si usa l'implementazione Pyhton della funzione seno

```
from math import sin as pysin
```

  si incorre nell'*overhead* di chiamata di una funzione Python

- La ufunc di NumPy

```
from numpy import sin as npsin
```

  è veloce per gli array, ma lenta se utilizzata su uno scalare

- Se si usa l'implementazione del seno da math.h

```
from libc.math cimport sin
```

  si dispone di una funzione veloce e non si paga alcun *overhead* di chiamata

44

CINECA SCAI SuperComputing Applications and Innovation

Profiling con annotazioni

`fib_orig.pyx`: Cython file senza cdef

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Creazione del sorgente annotato

```
$ cython -a fib_orig.pyx

$ open fib_orig.html
```

Un frammento del file `fib_orig.html`

Raw output: fib_orig.c

```
1: def fib(n):
2:     a,b = 1,1
3:     for i in range(n):
4:         a, b = a+b, a
5:     return a
```

- Sono evidenziate in giallo le linee di codice in cui sono necessari interventi in C per migliorare le prestazioni
- Il tono del giallo via via più scuro, indica il grado di necessità dell'intervento

45

CINECA SCAI
SuperComputing Applications and Innovation

Profiling con annotazioni

`fib.pyx`: Cython file con `cdef`

```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Creazione del sorgente annotato

```
$ cython -a fib.pyx

$ open fib.html
```

Lo stesso frammento del file `fib.html`

Raw output: fib.c

```
 1: def fib(int n):
 2:     cdef int a, b, i
 3:     a, b = 1, 1
 4:     for i in range(n):
 5:         a, b = a+b, a
 6:     return a
```


CINECA SCAI
SuperComputing Applications and Innovation

Cython in modalità Python puro

Il file fib.pyx

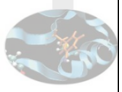
```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Il file fib.py

```
import cython

@cython.locals(n=cython.int)
def fib(n):
  cython.declare(a=cython.int,
                 b=cython.int,
                 i=cython.int)
  a,b = 1,1
  for i in range(n):
    a, b = a+b,
  return a
```

SCAI SuperComputing Applications and Innovation

Wrapping di funzioni C esterne



Il file `len_extern.pyx`

```
# Prima di tutti includiamo l'header file di cui abbiamo bisogno
cdef extern from "string.h":
    # Descriviamo l'interfaccia delle funzioni da utilizzare
    int strlen(char *c)

# La funzione strlen può essere usata in Cython, ma non in Python.
def get_len(char *message):
   return strlen(message)
```

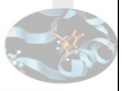
Importiamo il modulo in Python ed usiamo la funzione `get_len`

```
>>> import len_extern
>>> len_extern.strlen
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'strlen'
>>> len_extern.get_len("ciao!")
5
```

49

SCAI SuperComputing Applications and Innovation

Wrapping di strutture C esterne …



Il file `extern_time.pyx`

```
cdef extern from "time.h":
    # Dichiarazione di quanto è usato nella struttura tm
    struct tm:
        int tm_mday  # Giorno del mese: 1-31
        int tm_mon   # Mesi dopo gennaio: 0-11
        int tm_year  # Anni dopo il 1900

    ctypedef long time_t
    tm* localtime(time_t *timer)
    time_t time(time_t *tloc)

def get_date():
    """ Return a tuple with the current day, month, and year."""
    cdef time_t t
    cdef tm* ts
    t = time(NULL)
    ts = localtime(&t)
    return ts.tm_mday, ts.tm_mon + 1, ts.tm_year + 1900
```

50

CINECA SCAI
SuperComputing Applications and Innovation

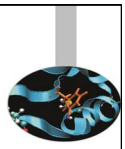
… *Wrapping* di strutture C esterne

Compiliamo il Cython file `extern_time.pyx` e utilizziamo il modulo ottenuto nella *shell* Python

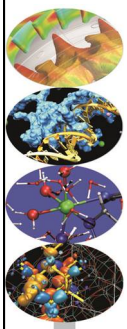
```
>>> import module extern_time as cTime

>>> cTime.get_date()
(13, 9, 2013)
```

51

CINECA SCAI
SuperComputing Applications and Innovation



Mixed language programming:
un caso reale - 2D wave equation



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Un caso reale: *wave equation …*

- Per verificare l'efficacia delle tecniche di *mixed programming* che abbiamo descritto, applichiamole ad un caso reale.
- Consideriamo l'equazione delle onde in un mezzo eterogeneo, con *k* velocità locale dell'onda

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

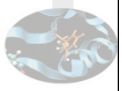
- Per il nostro scopo, è sufficiente limitarci al caso bidimensionale

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

53

CINECA SCAI
SuperComputing Applications and Innovation

… Un caso reale: *wave equation*



- Il dominio d'integrazione è $\Omega = (0,1)\mathrm{X}(0,1)$ e sul bordo *u=0*
- All'istante *t=0* il valore della u all'interno del dominio è pari alla funzione di Gauss in 2D

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

con *A=2*, $x_c = y_c = 0.5$ e $\sigma_x = \sigma_y = 0.15$

- La funzione *k* è definita come *k(x,y) = max(x,y)*

54

SCAI
SuperComputing Applications and Innovation

wave equation: discretizzazione

- Risolviamo l'equazione con il seguente schema alle differenze finite:

$$u_{i,j}^{l} = \left(\frac{\Delta t}{\Delta x}\right)^2 \left[k_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - k_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})\right]^{l-1} + \left(\frac{\Delta t}{\Delta y}\right)^2 \left[k_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - k_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1})\right]^{l-1}$$

dove $u^{l}_{i,j}$ rappresenta la *u* nel punto di griglia (x_i, y_j) al tempo t_l, in cui:

$$x_i = i\Delta x, i = 0, \ldots, n$$
$$y_i = j\Delta y, j = 0, \ldots, m$$
$$t_l = l\Delta t,$$

e $k_{i+1/2,j}$ è $k(x_{i+1/2}, y_j)$ in rappresentazione compatta

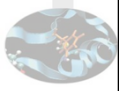
- Lo schema è esplicito e stabile sotto certe condizioni; scegliamo il massimo *time-step* possibile:

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

55

SCAI
SuperComputing Applications and Innovation

Applicazione ad un caso reale



- Il problema viene discretizzato attraverso uno schema alle differenze finite:

$$u_{i,j}^{l} = \left(\frac{\Delta t}{\Delta x}\right)^2 \left[k_{i+0.5,j}(u_{i+1,j} - u_{i,j}) - k_{i-0.5,j}(u_{i,j} - u_{i-1,j})\right]^{(l-1)} + \left(\frac{\Delta t}{\Delta y}\right)^2 \left[k_{i,j+0.5}(u_{i,j+1} - u_{i,j}) - k_{i,j-0.5}(u_{i,j} - u_{i,j-1})\right]^{(l-1)}$$

- Confrontiamo i tempi di calcolo ottenuti con diverse implementazioni dello stesso problema

56

CINECA SCAI
SuperComputing Applications and Innovation

Wave eq.: la versione Python+NumPy …

La funzione `calculate_u` evolve la soluzione di un *time-step*

```
# file Compute.py
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
```


CINECA SCAI
SuperComputing Applications and Innovation

… *Wave eq.*: La versione Python+Numpy …

```
# Definizione delle varibili
m = 250; n = 250 # grid size
dx = 1.0/m
dy = 1.0/n
k  = np.zeros((m+1, n+1))
up = np.zeros((m+1, n+1))
u  = np.zeros((m+1, n+1))
um = np.zeros((m+1, n+1))
A = 2
Xc = yc = 0.5
Sx = sy = 0.15

# Inizializzazione del campo di velocità
x=linspace(0,1,m+1)
y=linspace(0,1,n+1)
for i in xrange(0,m+1):
    for j in xrange(0,m+1):
        k[i,j]=max(x[i],y[j])
```

CINECA SCAI
SuperComputing Applications and Innovation

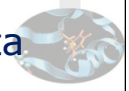
…*Wave eq.*: la versione Python+NumPy

```
x,y = np.meshgrid(x,y)
I = A*np.exp(((x-xc)/2*sx)**2-((y-yc)/2*sy)**2)
u = I
dt = float(1/sqrt(1/dx**2 + 1/dy**2)/k.max())

t=0; t_stop=1.0
print 'Start'          # loop per il calcolo della soluzione
start = time.clock()   # Start del contatore di elapsed time
while t <= t_stop:
        t += dt
        up = calculate_u(dt, dx, dy, u, um, up, k)
        um[:] = u
        u[:] = up
stop = time.clock()    # Stop del contatore di elapsed time
print 'Stop'
print 'Elapsed time: ', stop-start, 'sec.'
```

59

CINECA SCAI
SuperComputing Applications and Innovation

Wave equation: la versione Python + Numpy vettorizzata



Ovviamente ha senso vettorizzare solo la funzione `calculate_u`

```
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
```

60

CINECA SCAI
SuperComputing Applications and Innovation

La versione F2PY: il codice Fortran …

```
! file Compute.f90
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
    !f2py intent(in) u,um,k,n,m
    !f2py intent(out) up
    integer i,j
    hx=(dt/dx)*(dt/dx)
    hy=(dt/dy)*(dt/dy)
```

61

CINECA SCAI
SuperComputing Applications and Innovation

… La versione F2PY: il codice Fortran

```
   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   ! file Compute.f90
```

62

CINECA SCAI
SuperComputing Applications and Innovation

La versione F2PY: il codice Python

Con `f2py` generiamo il modulo `Compute`, che conterrà la funzione `calculate_u`, il *wrapper* Python alla relativa routine Fortran

```
import numpy as np
import time
import Compute
 …
print 'Start'
start=time.clock()
while t <= t_stop:
   t += dt
   up = Compute.calculate_u(dt, dx, dy, u, um, k)
   um[:] = u
   u[:] = up
stop=time.clock()
print 'Stop'
print 'Elapsed time: ', stop-start, 'sec.'
```

63

CINECA SCAI
SuperComputing Applications and Innovation

La versione in Cython

```
#File Compute.pyx
import numpy as np
cimport numpy as np
cimport cython
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] u,  \
        np.ndarray[DTYPE_t, ndim=2, negative_indices=False] um, \
        np.ndarray[DTYPE_t, ndim=2, negative_indices=False] up, \
        np.ndarray[DTYPE_t, ndim=2, negative_indices=False] 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
```

64

CINECA SCAI
SuperComputing Applications and Innovation

La versione in Cython

```
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
```

65

CINECA SCAI
SuperComputing Applications and Innovation

I risultati delle misure

Implementazione	Elapsed Time (sec)
Python + NumPy	**382.32**
Python + NumPy (con vettorizzazione)	**1.52**
F2Py	**0.26**
Cython	**0.25**

Griglia di calcolo 250 X 250

Processore: Intel(R) Xeon(R) CPU X5460 @ 3.16GHz

66