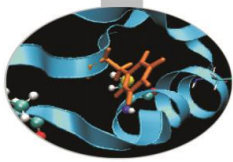
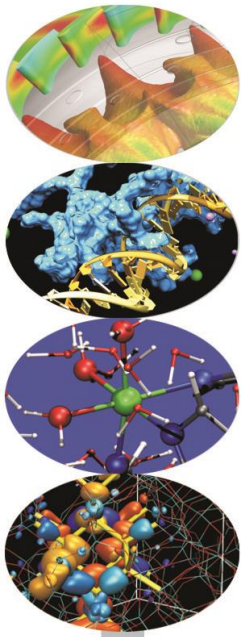
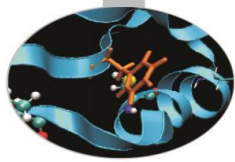


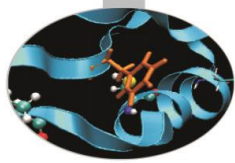
Mixed Language Programming



Indice

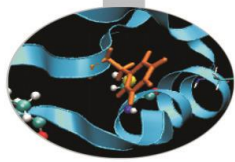


Python Optimization strategies

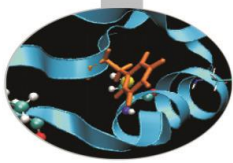


- Per migliorare le prestazioni di un codice Python esistono diverse strategie:
  - Sfruttare la libreria Numpy e vettorizzare il codice
  - Sfruttare pacchetti ottimizzati per task specifici
  - Cython,Swig,f2py e integrazione con linguaggi low level
  - Parellelizzazione
  - GPU porting

Python Optimization strategies



STEP 0: Prima di ottimizzare individuare gli hot-spot del codice



Timing your code

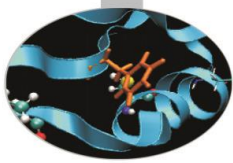
Con il modulo *timeit* è possibile misurare il tempo di esecuzione di una funzione o di una espressione.

E' particolarmente adatto per test molto piccoli e su test particolarmente veloci (micro-sec) e può essere richiamato da riga di comando.

Esempio:

```
import timeit
def test_func():
    a=[]
   for el in xrange(1000000): a.append(el)
if __name__==‘__main__’:
    print  timeit.Timer(‘for el in range(10000000):pass’).timeit(1), ‘s’
    print  timeit.Timer(‘for el in xarange(10000000):pass’).timeit(1), ‘s’
    print  timeit.Timer(‘test_func()’, ‘from __main__ import
    test_func()’).timeit() ,’s’
```

Timing your code



0.334960419911

0.947776416355

0.439117697007

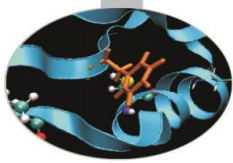
Oppure da linea di comando:

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.test_func”

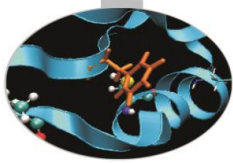
Timing your code



Il modulo cProfile è lo strumento standard di Python per il profiling di un codice.

```
import cProfile
def test():
  lista=[]
  for el in xrange(1000000):
      lista.append[el]

if __name__==‘__main__’:
   cProfile.run(‘test_func()’)
```

Timing your code



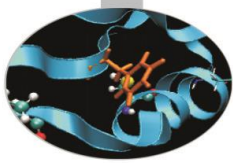
OUTPUT

1000003 function calls in 1.383 CPU seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:
1	0.015	0.015	1.383	1.383	<string>:1(<module>)
1	0.841	0.841	1.368	1.368	p.py:5(test_func)
1000000	0.526	0.000	0.526	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable', '_lsprof.Profiler' objects}

Timing your code



In maniera analoga ai metodi del modulo timeit, il profiling di un codice o di una parte di esso può essere effettuato direttamente da linea di comando.

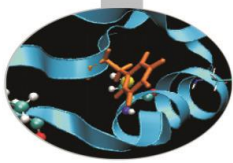
python -m cProfile mymodule.py

Specificando il metodo di ordinamento delle informazioni e reindirizzando l'output:

python –m cProfile –s 'cumulative' –o 'profile_file.txt' mymodule.py

L'ordinamento è basato sulla classe pstats.Stats.

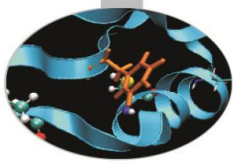
Python Optimization strategies



- Vantaggi di Numpy:
  - Contenitore dati multidimensionali
  - Accesso efficiente ai dati
  - Strumenti di manipolazione dati ottimizzati

- Si potrebbe migliorare:
  - Valutazione di espressioni (Numexpr)
  - Introdurre il supporto al multiprocessing
  - Contenitore dati più flessibile per ‘big data’ (Carray)

Numexpr



- Numexpr:
  - Modulo specializzato per la valutazione di espressioni
  - Uso della memoria efficiente
  - Multithreading integrati e semplice da usare

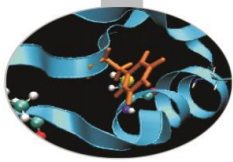
- **Example**

Valutazione del polinomio : 0,25x^3+0,75x^2+1,5x-2 in range[-1,1] con step size di 2*10^-7

```
x = numpy.linspace(-1, 1, 1000*10000)
y=.25*x**3 + .75*x**2 - 1.5*x - 2
y2=numexpr.evaluate(".25*x**3 + .75*x**2 - 1.5*x - 2")
```

Numexpr 10x Faster than Numpy

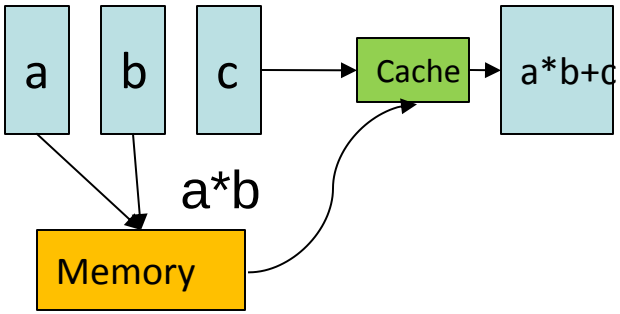
Numexpr



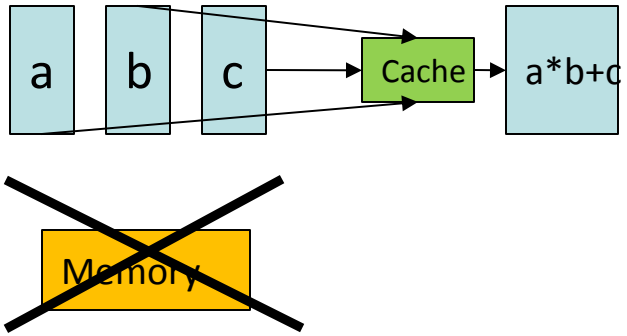
- Riscrittura del polinomio in forma:
  - **((0.25x + 0.75)x + 1.5)x – 2**

Numexpr is still 3x Faster than Numpy

Numpy



Numexpr

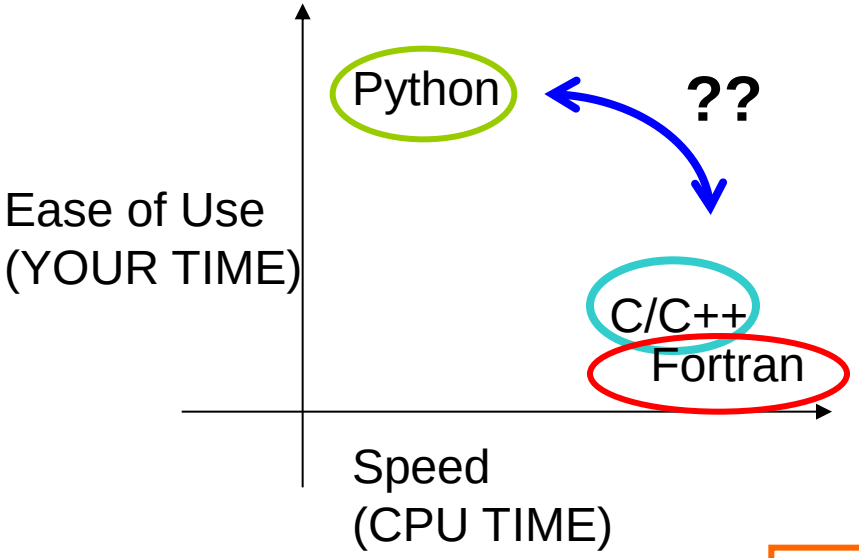


Numexpr ~ C/OpenMP

- Numexpr usa il multithreading in modo semplice:

```
numexpr.set_num_threads(nthreads)
```

Python and C/C++ or Fortran



C/C++ & Fortran:

- High performance
- Basso livello
- Tipizzazione statica

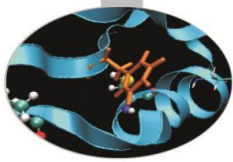
wrapper

(comm. layer)

Python:

- Low performance
- Alto livello
- Tipizzazione dinamica

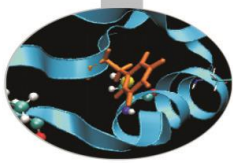
Python vs Fortran



- Python
- Si pensa sia lento
- E’ tutto ad oggetti
- Lento per il puro calcolo scientifico
- Numpy e altro per ottimizzare

- Fortran
- Pensato per il calcolo scientifico
- E’ veloce
- Le performance sono il suo punto di forza

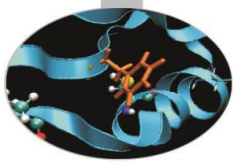
Python vs Fortran



- Coordinate comuni in un file di dati di questo tipo:

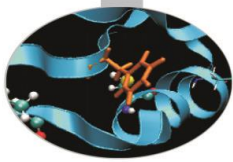
```
x                          y                  z
-0.930730079412    -2.5837626746    -1.54168368133
0.242648096556    -0.0356080560781    1.64592869632
1.08469025095    -0.0499861074276    0.850168034143
0.904693997768    0.331774955996    -0.435462229263
-0.30666600039    0.502104683611    -0.147720257532
0.67879008187    0.594205526229    -2.76854972969
0.346956345057    0.64044334422    -0.397530419261
-0.415357107828    0.431902374758    -0.96024517464
-0.0423741085785    0.97632569202    1.13665523717
-0.0890995975459    -0.0170642871372    -0.209950709345
```

Python vs Fortran



```
import sys

def read_file_short(file_name):
    with open(file_name) as fobj:
        next(fobj)
        return (tuple(float(entry) for entry in line.split()) for line in fobj)

def find_intersection(file_name1, file_name2):
    data1=set(read_file(file_name1))
    data2=set(read_file(file_name2))
    return data1.intersection(data2)

if __name__=='__main__':
    if (len(sys.argv)>1):
        dim=sys.argv[1]
        print find_intersection('value_'+dim+'_0.txt', 'value_'+dim+'_1.txt')
```

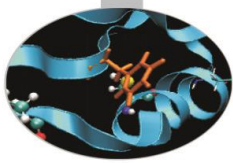
Python vs Fortran



Lettura dei dati

```
subroutine read_data(file_name, data_array, line, row)
        implicit none
        integer,intent(in)::line
        integer,intent(in)::row
        integer::status
        integer::l
        double precision, dimension(line,row)::data_array
        character(len=20),intent(in)::file_name
        Character(len=100)::dummy
        open(unit=99,file=file_name,status='old',action='read')
        read(99,*,iostat=status) dummy
        do l=1,line
                read(99,*,iostat=status) data_array(l,1), data_array(l,2),
data_array(l,3)
                if (status /=0) exit
        enddo
        close(99)
end subroutine read_data
```

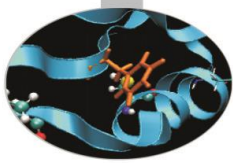
Python vs Fortran



Funzione di intersezione

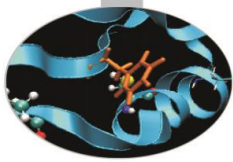
```
subroutine find_intersection(data_array1,data_array2,line1,line2,row)
        implicit none
        integer, intent(in)::line1,line2,row
        integer::status,counter=0,l1,l2,r,matches
        double precision, intent(in),dimension(line1,row)::data_array1
        double precision, intent(in),dimension(line2,row)::data_array2
        double precision, allocatable, dimension(:,:):: found_temp
        double precision, allocatable, dimension(:,:):: found
        allocate(found_temp(min(line1,line2),row),stat=status)
        do l1=1,line1
                do l2=1,line2
                matches=0
                        do r=1,row
                                if(data_array1(l1,r)==data_array2(l2,r)) then
                                        matches=matches+1
                                endif
                        enddo
                if (matches==3) then
                        counter=counter+1
                        do r=1,row
                                found_temp(counter,row) =data_array1(l1,r)
                        enddo
                endif
                enddo
```

Python vs Fortran



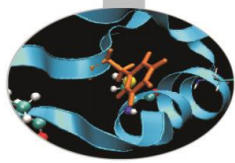
```
enddo
        allocate (found(counter,row),stat=status)
        if(status /=0) then
                write(*,*),'Error allocating'
        end if
        do l1=1,counter
                do r=1,row
                        found(l1,r)=found_temp(l1,r)
                end do
        end do
        write(*,*) counter
end subroutine find_intersection
```

Python vs Fortran



Programma chiamante

```
program common_coord
implicit none
integer::number_of_line1=0
integer::number_of_line2=0
integer::status
character(len=100)::dummy
double precision,allocatable, dimension(:,:) :: data_array1
double precision,allocatable, dimension(:,:) :: data_array2
character(len=100)::dimension
character(len=20) :: file_name1 = "value_10000_0.txt"
character(len=20) :: file_name2 = "value_10000_1.txt"
call count_line(file_name1,number_of_line1)
call count_line(file_name2,number_of_line2)
allocate(data_array1(number_of_line1,3),stat=status)
call read_data(file_name1,data_array1, number_of_line1,3)
allocate(data_array2(number_of_line2,3),stat=status)
if(status /=0) then
        write(*,*) 'Error allocating 2.'
end if
call read_data(file_name2,data_array2,number_of_line2,3)
call find_intersection(data_array1,data_array2,number_of_line1,number_of_line2,3)
end program common_coord
```

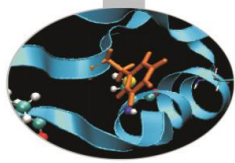


Python vs Fortran

- Python è più flessibile
- Python è più compatto
- 8 righe vs 107
- Python dispone di contenitori dati molto flessibili

Dimension	Python (s)	Fortran (s)
1000	0,092	0,051
10000	0,189	1,984
100000	0,876	238
1000000	29,34	??

Benefici

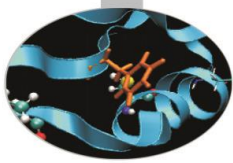


L'integrazione è interessante in almeno 2 contesti:

- *Migrazione di codice lento*: scrittura di un nuovo codice in Python migrando la parte di calcolo intensivo verso linguaggi HPC.
- *Accesso a codici già esistenti*: utilizzo di libreria e codici validati scritti in linguaggi HPC, direttamente in Python

In entrambi i casi Python è utilizzato per task non numerici, traendo benefici dalla semplicità nella gestione di:

- I/O
- interfacce
- gestione dell'applicazione
- reporting e post processing
- GUI



Strategie: C-API

Chiamare funzioni scritte in C/C++ e F77 in Python non è banale:

compilati	⟷	interpretati
fortemente tipizzati	VS	debolmente tipizzati

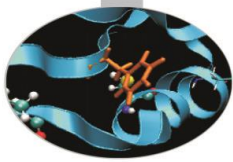
La maniera nativa per scrivere un modulo Python in C consiste nell'usare le C-API di Python.

ESEMPIO:

```
extern double f1(double a, double b); //f1 è presente in un modulo M1
from M1 import f1
a=10.0
b=20.0
#chiamata
c= f1(a,b)
```

deve esistere una funzione wrapper (scritta in C, che effettua la traslazione dei tipi di dato e che converta il risultato ritornato dalla funzione in un oggetto Python)

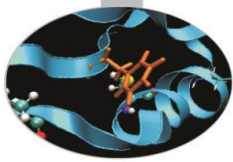
Strategie: C-API



In questo caso il wrapper è strutturato come segue:

```
//M1module.c
#include <Python.h>
static PyObject *_wrap_f1(PyObject *self, PyObject *args){
        double arg1, arg2, result;
        if (!PyArg_ParseTuple(args,"dd:f1,&arg1,&arg2)){
                return NULL;
        }
        result = f1 (arg1,arg2);
        return Py_BuiltValue("d",result);
}
```

Strategie: C-API



Il wrapper va compilato (*C-compiler*) e il codice oggetto va linkato al codice oggetto del modulo M1 (scritto in C che contiene la definizione della funzione f1 che vogliamo utilizzare).

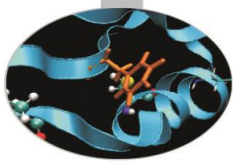
Viene generata una libreria (*extension module*) che può essere importata in Python con un *import* classico.

A questo livello in Python è IMPOSSIBILE distinguere tra una modulo *pure python* ed una *extension module*.

Possono essere scelti principalmente 2 approcci:

- utilizzo delle Python C API (Application Programming Interface)
- utilizzo di strumenti automatici

Python C-API: costi



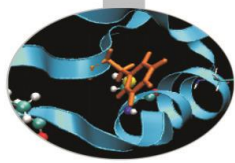
L'utilizzo delle Python C API comporta una grande molo di lavoro:

- Ogni funzione C che si vuole utilizzare DEVE ESSERE FORNITA DI WRAPPER
- Error-prone
- Conoscenza dettagliata dell'interfaccia C a Python.

Strumenti automatici:

- SWIG (Semplified Wrapper Interface Generator)
- F2PY
- Instant (usa SWIG) oggi Fenix
- Cython
- …

F2PY



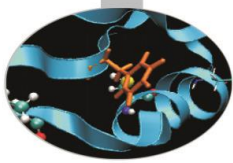
F2PY è un tool che permette di creare interfacce Python a funzioni scritte in Fortran e C.

F2PY è nativamente presente nella libreria Numpy e non necessità di installazione a parte se Numpy è già presente.

ESEMPIO:

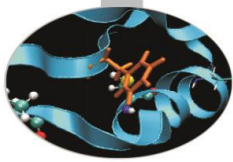
```
#!/usr/bin/env python
"“”hw: modulo in python"""“
import math


def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s


def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1, r2, s)
```

F2PY



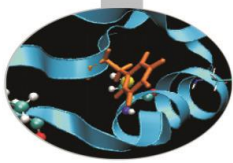
```
#!/usr/bin/env python
"""uso del modulo hw da python"""
import sys
from hw import hw1, hw2
try:
    r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
except IndexError:
        print 'Usage:', sys.argv[0], 'r1 r2'; sys.exit(1)

print 'hw1, result:', hw1(r1, r2)
print 'hw2, result: ',hw2(r1, r2)
```

F2PY: Fortran Automatic Wrapper



```
PROGRAM hwtest
      REAL *8 r1, r2 ,s
      r1 = 1.0
      r2 = 0.0
      s = hw1(r1, r2)
      WRITE(*,*) 'hw1, result:',s
      WRITE(*,*) 'hw2, result:'
      CALL hw2(r1, r2)
END
SUBROUTINE hw2(r1, r2)
      REAL *8 r1, r2, s
      s = sin(r1 + r2)
WRITE(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      RETURN
      END

REAL*8 FUNCTION hw1(r1, r2)
      REAL*8 r1, r2
      hw1 = sin(r1 + r2)
      RETURN
      END
```

F2PY



Per generare il wrapper:

```
f2py -m hw --fcompiler=gfortran -c hw.f90
```

`-m` specifica il nome del *extension module* che vogliamo creare

`-c` specifica che vogliamo compilare e linkare il modulo

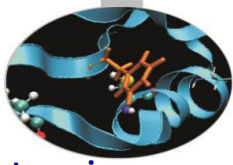
`--fcompiler` indica il compilatore da utilizzare

In output abbiamo l'extension module sotto la forma di un hw.so (.dll in win32; .dylib in Mac OS X)

Possiamo testare che il modulo sia stato generato con successo :

```
>>>import hw
>>> hw.hw1(1,2)
0.14112000805986721
>>> hw.hw2(1,2)
Hello, World! sin( 3.000)=0.141120
```

F2PY



Per moduli molto grandi è possibile rendere wrappate solo alcune funzioni, con la sintassi:

```
f2py -m hw -c --fcompiler=gfortran hw.f only: hw1 hw2:
```

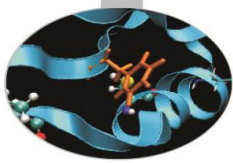
F2PY è stato inizialmente progettato per interfacciare funzioni fortran a Python ma può essere utilizzato anche per creare wrapper di funzioni scritte in C.

ESEMPIO:

//File hw.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("Hello World !sin(%g+%g)=%g\n",r1,r2,s);
}
```

F2PY

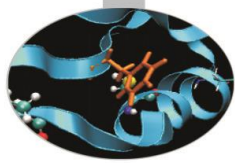


```
(mult.f90)
subroutine fmult(a,b,c,n)
implicit none
integer, parameter :: dim = 100
real*8 :: a(dim)
real*8 :: b(dim)
real*8 :: c(dim)
integer :: n, i
do i =1,n
  c(i) = a(i) * b(i)
enddo
end
```

```
$ f2py -c -m mult mult.f90
```

```
>>> import mult
>>> print mult.fmult.__doc__
fmult  Function signature:
  fmult(a,b,c,n)
Required arguments:
  a : input rank1 array('d') with bounds (100)
  b : input rank1 array('d') with bounds (100)
  c : input rank1 array('d') with bounds (100)
  n : input int
>>> a = np.ones(100)+ 3;
>>> c = a.copy()
>>> b = np.ones(100)+ 1.5
>>> mult.fmult(a,b,c,100)
```

F2PY



- Possiamo migliorare l'interfaccia prodotta e renderla più vicina allo 'stile Python'

```
$ f2py -h mult.pyf -m mult mult.f90
```

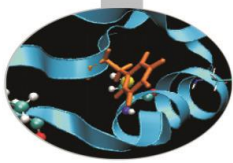
```
python module mult
  interface
    subroutine fmult(a,b,c,n)
      real*8 dimension(*) :: a
      real*8 dimension(*) :: b
      real*8 dimension(*) :: c
      integer :: n
    end subroutine fmult
  end interface
end python module mult
```

Generato in automatico

```
(mult.pyf)
python module mult
  interface
    subroutine fmult(a,b,c,n)
      real*8 dimension(n) :: a
      real*8 dimension(n) :: b
      real*8 intent(out), dimension
(n) :: c
      integer intent(hide), depend(
a) ::
n=len(a)
    end subroutine fmult
  end interface
end python module mult
```

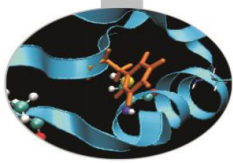
Intervento manuale

F2PY



```
>>> import numpy as np
>>> import mult
>>> a = np.array([1, 3, 4])
>>> b = np.array([2, 5, 1.5])
>>> c = mult.fmult(a, b)
>>> f = mult.fmult([3,4], [1.2,1.4])
>>> f
array([ 3.6,  5.6])
```

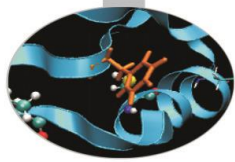
F2PY



Oppure inserendo le direttive come commento nei sorgenti

```
(mult2.f90)
subroutine fmult(a,b,c,n)
implicit none
integer, parameter :: dim=1000
real*8 :: a(n)
real*8 :: b(n)
real*8 :: c(n)
integer :: n, i
!f2py intent(hide), depend(a) :: n=len(a)
!f2py real*8                 :: a(n)
!f2py real*8                 :: b(n)
!f2py real*8, intent(out)    :: c(n)
do i =1,n
  c(i) = a(i) * b(i)
enddo
end
```

F2Py: C wrapping



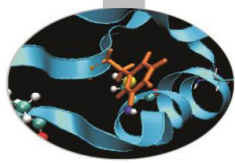
Per ogni funzione C è necessario creare un signature file, la funzione e i suoi argomenti devono avere attributo *intent(c).*

```
//! File m.pyf
python module hwC
interface
  function hw1(r1,r2)
    intent(c) foo
    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 m
```

- Per generare il wrapper utilizzare il comando:

```
f2py m.pyf foo.c -c
```

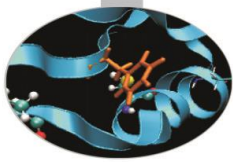
Swig



SWIG (**Simplified Wrapper and Interface Generator)** è un tool:

- permette di creare interfacce di codice scritto in C/C++ per linguaggi interpretati (Perl, Python,Tuby, Tcl)
- E' pensato per lavorare con codice C/C++ già esistente, usando solo una interfaccia di scripting.
- Deve essere installato a parte: http://www.swig.org/index.php
- Supporta le strutture C e le classi C++
- E' indipendente dal linguaggio
- Supporta una varietà di opzioni di customizzazione.

Swig



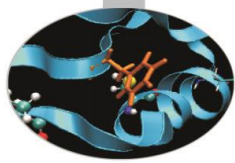
- Per generare un wrapper è necessario creare un file di 'interfaccia' per SWIG.
- La sintassi del file di interfaccia consiste in un mix delle direttive SWIG (identificate da %) , da istruzioni per il preprocessore C (identificate da #), da codice C e da commenti

%module: definisce il nome dell'extension module

%{…}% blocco utilizzato per inserire codice C necessario per rendere corretta la compilazione (header files, dichiarazione di funzioni e dipendenze)

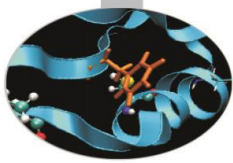
- infine vengono dichiare le funzioni che devono essere wrappate nell'interfaccia.

Esempio



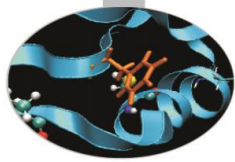
```
/* file: hw.c */
#include <stdlib.h>  /* need atof */
#include <stdio.h>
#include <math.h>

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("Hello, World! sin(%g+%g)=%g\n", r1, r2, s);
}
```

Esempio



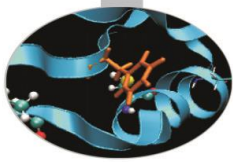
```
/* file: hw.c */
int main(int argc, char* argv[])
{
  double r1, r2, s;
  if (argc < 3) {  /* need two command-line arguments */
    printf("Usage: %s r1 r2\n", argv[0]);  exit(1);
  }
  r1 = atof(argv[1]);  r2 = atof(argv[2]);
  printf("hw1, result: %g\n", hw1(r1, r2));
  printf("hw2, result: ");
  hw2(r1, r2);
return 0;
}
```

Esempio



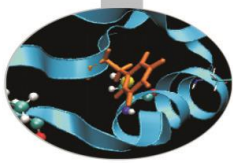
```
/* file: hw.h*/
#ifndef HW_H
#define HW_H
extern double hw1(double r1, double r2);
extern void   hw2(double r1, double r2);
#endif
```

Esempio



```
/* file: hw.i */
%module hw
%{
/* include C header files necessary to compile the interface */
#include "hw.h"
%}

double hw1(double r1, double r2);
void   hw2(double r1, double r2);
```

Swig steps



1. la generazione del codice wrappato avviene tramite la chiamata di swing:

```
swig –python –I/path_to_hw.h .. hw.i
```

L'output del processo è un file che contiene il C wrapper code (.c) e un file hw.py che costituisce l'interfaccia all'extension module

2. Il wrapper C code va compilato insieme al C source code che contiene le implementazioni delle funzioni e linkato ad esso in un object file (_hw.so) che costituisce l'extension module.

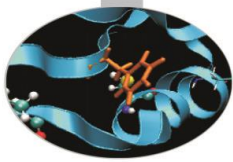
Creazione dell'object file compilato con l'opzione -fPIC

```
gcc -fPIC -I/data/apps/bin/epd-5.0.0-rh5-
x86_64/include/python2.5/ -I/data/apps/bin/epd-5.0.0-
rh5-x86_64/lib/python2.5/config  -c ../hw.c hw_wrap.c
```

Creazione della shared lib

```
gcc -shared -o _hw.so hw.o hw_wrap.o
```

Swig



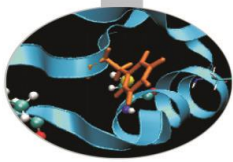
- In alternativa è possibile automatizzare le fasi di compilazione sfruttando il modulo *distutils.*
- Creare un file di setup.py:

```
import commands, os
from distutils.core import setup, Extension
name='hw'
version=1.0
swig_cmd = 'swig -python %s.i' %name
print 'Running Swig: ', swig_cmd
failure, output =commands.getstatusoutput(swig_cmd)
module1 = Extension('_hw',
                    sources = ['hw.c','hw_wrap.c'])
setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

Eseguire il comando:

```
python setup.py build
```

Example



Un esempio di utilizzo reale di swig:

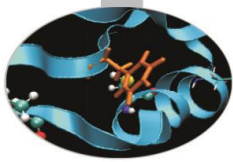
- pyMagma

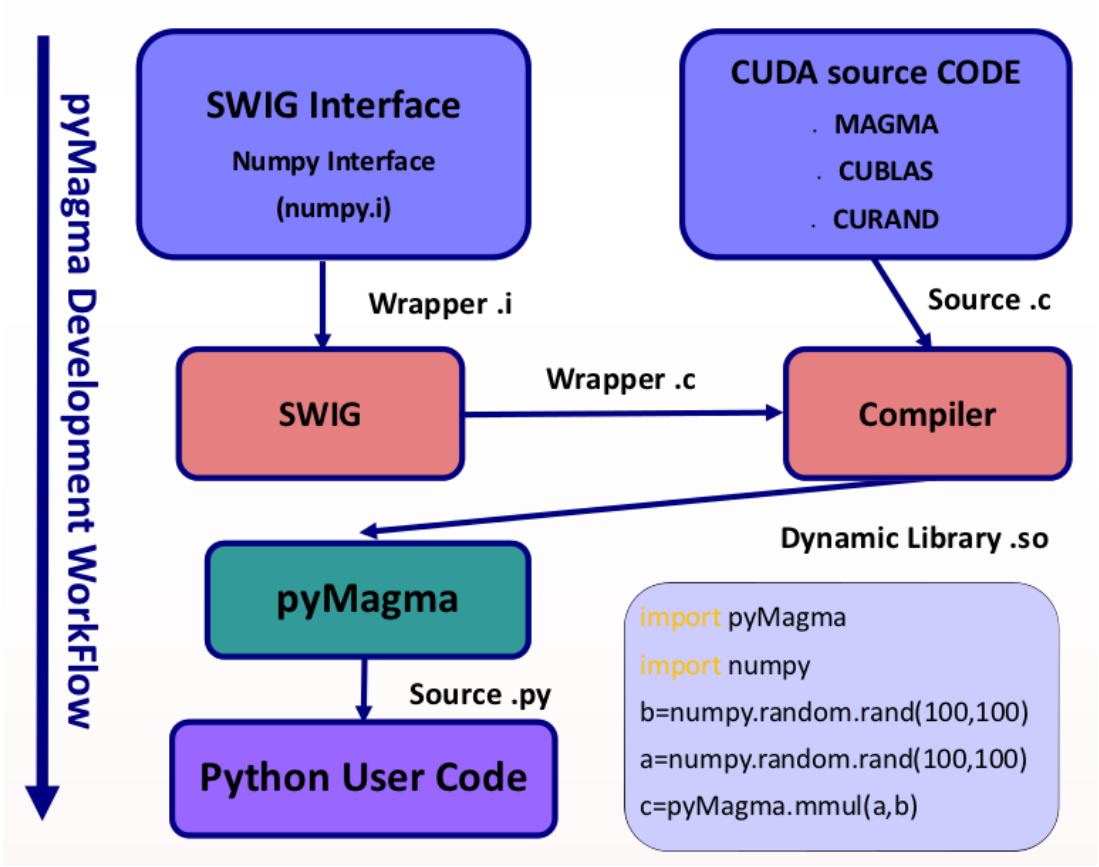
pyMagma is developed on MAGMA library and makes use of Numpy array as working data-strucures. In order to build a wrapper interface between modules written in C and Python, SWIG tool was used.

pyMagma Routines and Functionalities		
	Single GPU	Multi-GPUs
Cholesky Factorization	✓	✓
LU Factorization	✓	✓
QR Factorization	✓	✓
Linear System Solver	✓	✓ *
Matrix Inversion	✓	✓ *
EigenValue/Eigen Vector Problem	✓	
Random Number Generator	✓	

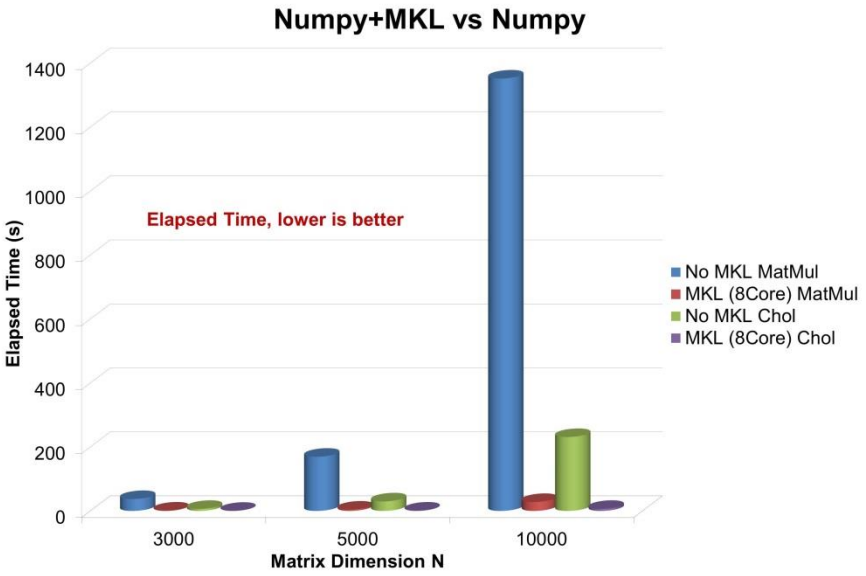
✓* Only some step of the algorithm ara available with Multi-GPUs support

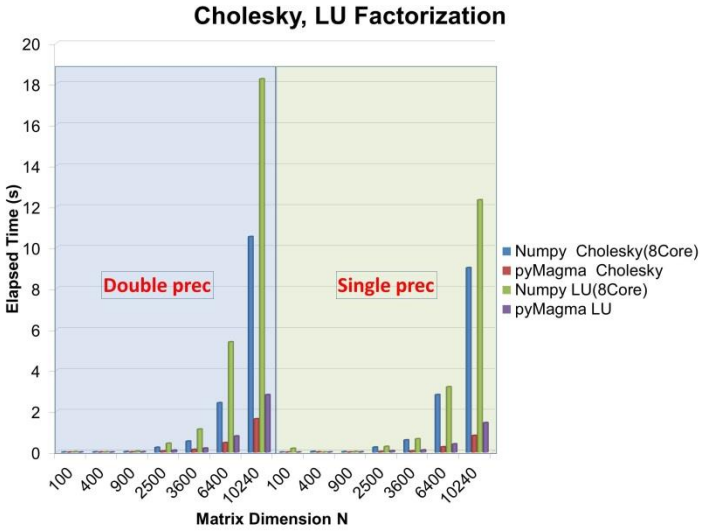
Example

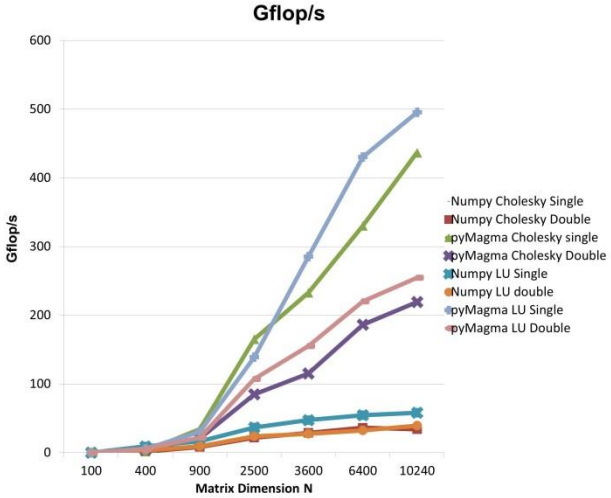



pyMagma Development WorkFlow
SWIG Interface
Numpy Interface
(numpy.i)
CUDA source CODE
. MAGMA
. CUBLAS
. CURAND
Wrapper .i
Source .c
SWIG
Wrapper .c
Compiler
Dynamic Library .so
pyMagma
Source .py
Python User Code
import pyMagma
import numpy
b=numpy.random.rand(100,100)
a=numpy.random.rand(100,100)
c=pyMagma.mmul(a,b)

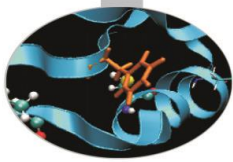
Example


Numpy+MKL vs Numpy
Elapsed Time, lower is better
Elapsed Time (s)
1400
1200
1000
800
600
400
200
0
3000
5000
10000
Matrix Dimension N
No MKL MatMul
MKL (8Core) MatMul
No MKL Chol
MKL (8Core) Chol


Cholesky, LU Factorization
Elapsed Time (s)
20
18
16
14
12
10
8
6
4
2
0
Double prec
Single prec
100
400
900
2500
3600
6400
10240
Matrix Dimension N
Numpy Cholesky(8Core)
pyMagma Cholesky
Numpy LU(8Core)
pyMagma LU


Gflop/s
Gflop/s
600
500
400
300
200
100
0
100
400
900
2500
3600
6400
10240
Matrix Dimension N
Numpy Cholesky Single
Numpy Cholesky Double
pyMagma Cholesky single
pyMagma Cholesky Double
Numpy LU Single
Numpy LU double
pyMagma LU Single
pyMagma LU Double

Cython



- *Cython* è un'estensione di Python pensata per consentire di estendere facilmente Python con moduli scritti in altri linguaggi.
- Scaricabile: http://cython.org/
- E' un'evoluzione di *pyrex*
- *Cython* crea automaticamente il binding di oggetti/funzioni C/C++ usando le python C-API.
- Cython produce un codice C/C++ ottimizzato che può essere compilato per creare un modulo aggiuntivo di Python.
- Il codice generato può essere compilato ed eseguito senza avere Cython installato.

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.

Cython può essere utilizzato per :

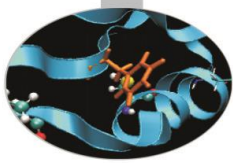
- L' estensione dell'interprete Python con moduli binari veloci
- L'interfacciamento di codice Python con librerie C esterne

Cython può compilare la maggior parte dei sorgenti Python

Il codice C generato presenta miglioramenti di velocità 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

Cython



Alcune regole di sintassi di base:

- **Definizione di variabili**: lo statement *cdef* permette di definire delle variabili C:

```
cdef int a
cdef struct my_struct:
    int a;
    double b;
```

•**Definizione di funzioni**: ci sono tre tipi di funzioni in Cython, funzioni Python definite con lo statement *def* che ricevono/restituiscono Python objects, funzioni C definite con lo statement *cdef* che ricevono/restituiscono Python objects o dati C, funzioni definite con lo *cpdef*

```
def my_function(a, b)        #Python function
cdef double my_function2(int a, double b) #C function
cdef my_function3(a,b) #C function with Python object
cpdef my_function(a,b)
```

Cython

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

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

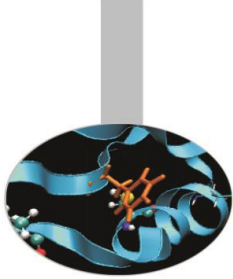
```
cdef float distance(float *x, float *y, int n):
    cdef:
    int i
    float d = 0.0
for i in range(n):
    d += (x[i] - y[i])**2
return d
```

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

Cython workflow per velocizzare Python



Python

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

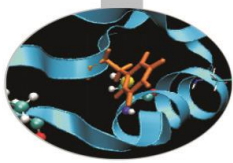
Cython

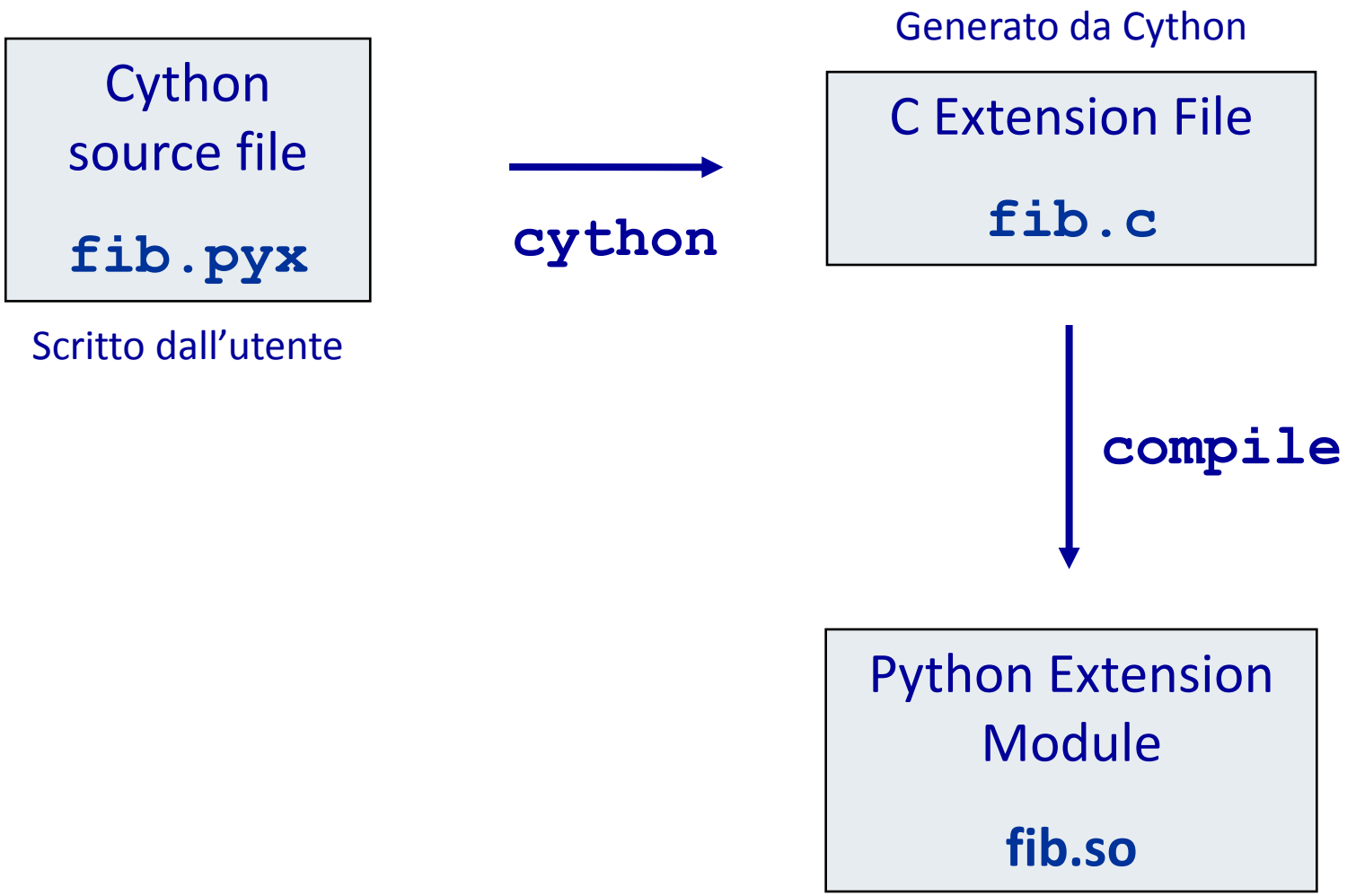
```
def fib(int n):
    cdef int i, a, b
    a,b = 1,1
    for i in
range(n):
        a, b = a+b,
               a
    return a
```

Codice C generato da Cython

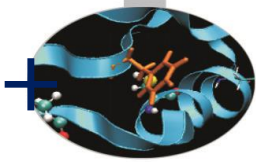
```
static PyObject
*__pyx_pf_5cyfib_cyfib(PyObject
*__pyx_self, int __pyx_v_n) {
  int __pyx_v_a; int __pyx_v_b;
PyObject *__pyx_r = NULL;
PyObject *__pyx_t_5 = NULL;
const char *__pyx_filename =
NULL;
...
  for (__pyx_t_1=0;
__pyx_t_1<__pyx_t_2;
__pyx_t_1+=1) {
    __pyx_v_i = __pyx_t_1;
    __pyx_t_3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

Cython pyx file





Wrapping di codice scritto in C/C++



C/C++

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

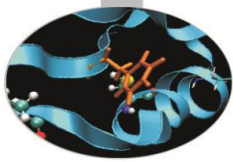
Cython

```
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;
  }
...
}
```

Cpython pyx file



Cython source file

fib.pyx

Scritto dall'utente

cython

Generato da Cython

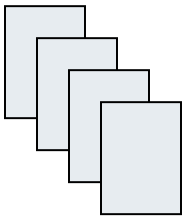
C Extension File

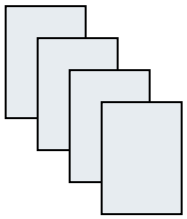
fib.c

compile

Library Files (if wrapping)

***.h files** ***.c files**





compile

Python Extension Module

fib.so

Cython

- Uno dei principali utilizzi di Cython è per il wrapping di librerie C già esistenti. Cython può essere utilizzato come ponte per consentire a Python di chiamare funzioni C.

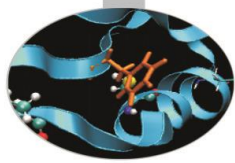
Il file len_extern.pyx

```
# 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
```

Cython



Source file e compilazione:

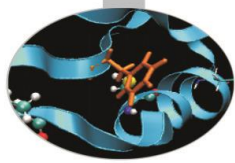
- I codici sorgenti devono essere salvati in un file con estensione .pyx

- Per la generazione dell'extension module è possibile operare in due modi:

  - Compilare i *.pyx* files con il *Cython compiler*, ottenendo dei *.c* file sorgenti e compilare il codice C in maniera appropriata, per generare un eseguibile, con i compilatori disponibili sulla propria piattaforma di lavoro.

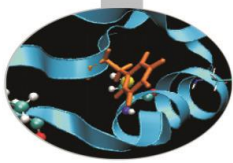
  - Utilizzare il modulo *Cython.Distutils* scrivendo un opportuno file di *setup.py*.

Esempio



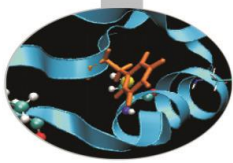
```
//mylib.h
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

double function(double x);
double integrate_trap(double a, double b, int N);

//mylib.c
#include "mylib.h"

double function(double a){
        return pow(a,2);
        }

double integrate_trap(double a, double b, int N){
        double s=0.0;
        double dx=fabs((b-a))/N;
        int i=0;
        double xa=a;
        for (i=0;i<N;i++){
                s+=((function(xa)+function(xa+dx))*dx)/2;
                xa=xa+dx;}
        return s;}
```

Esempio



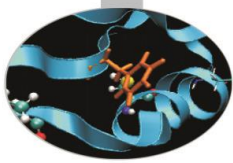
```
#Definizione delle librerie C:
cdef extern from "mylib.h":
        double function(double a)
        double integrate_trap(double a, double b, int N)

#Definizione della funzione Python
def integrate_rect(a,b,N):
        s=0.0;
        dx=float(b-a)/N;
        for i in range(N):
                s+=function(a+i*dx)
        return s*dx

print 'Result ', s
```

Esempio



```
cdef double s=0.0
print 'Compute integration of function x^2'
print 'Between interval a, b with N points'
a=float(raw_input("Insert a:"))
b=float(raw_input("Insert b:"))
N=int(raw_input("Insert N:"))

print 'Choose integration method‘
print '1- Rectangular method'
print '2- Trapez method'
c=raw_input('Choice= ')
if c=='1':
      s=integrate_rect(a,b,N)
if c=='2':
      s=integrate_trap(a,b,N)
print 'Result ', s
```

Esempio



Per generare un extension module:

1. Compilare .pyx:

```
cython myscript.pyx
```

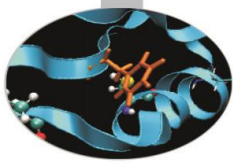
2. Compilare i codici sorgenti e produrre una shared lib:

```
 gcc -I/data/apps/bin/epd-5.0.0-rh5-x86_64/include -fPIC
-I/data/apps/bin/epd-5.0.0-rh5-x86_64/include/python2.5
-c mylib.c myscript.c

gcc -pthread -shared mylib.o myscript.o -L/data/apps/bin/
/epd-5.0.0-rh5-x86_64/lib -lpython2.5 -o myscript.so
```

3. A questo punto è possibili utilizzare in Python il modulo appena creato:

```
import myscript
Compute integration of function x^2
Between interval a, b with N points
Insert a:1
Insert b:2
Insert N:10
Choose integration method
1- Rectangular method
2- Trapez method
Choice= 1
Result  2.185
```

Esempio



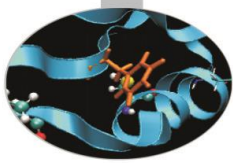
- E' possibile generare in automatico un *extension module,* utilizzando il modulo *Cython.Distutils* e generando un file di *setup.py*

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

sourcefiles = ['myscript.pyx', 'mylib.c']
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("myscript", sourcefiles)])
```

Da lanciare con il comando:

```
python setup.py build
```

Esempio



Tipicamente Cython viene utilizzato per creare degli *extension module* per Python. E' possibile tuttavia generare dei programmi standalone, includendo l'interprete Python nell'eseguibile prodotto.

Per generare un eseguibile:

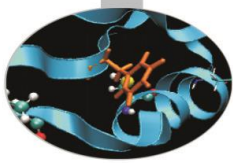
1. Compilare *.pyx* con il flag `--embed`

```
cython myscript.pyx –embed
```

2. Compilare i sorgenti e generare l'eseguibile:

```
gcc -I/data/apps/bin/epd-5.0.0-rh5-x86_64/include
-I/data/apps/bin/epd-5.0.0-rh5-
x86_64/include/python2.5 -c mylib.c myscript.c


gcc myscript.o mylib.o  -L/data/apps/bin/epd-
5.0.0-rh5-x86_64/lib -lpython2.5  -o my_exe
```

Benchmark



- Consideriamo l'equazione delle onde in un mezzo eteronegeneo, con velocità *k*, nel caso bidimensionale:

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

- Consideriamo un dominio rettangolare $\Omega = (0,1)\mathrm{X}(0,1)$, con *u=0* sull'intero bordo e con le seguenti condizioni iniziali:

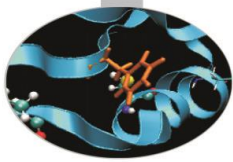
$$u(x,y,0) = A\exp\left(\left(\frac{x-x_c}{2\sigma_x}\right)^2 + \left(\frac{y-y_c}{2\sigma y}\right)^2\right)$$

$$\frac{\partial u}{\partial t} = 0$$

$$A = 2 \qquad x_c = y_c = 0.5 \qquad \sigma_x = \sigma_y = 0.15$$

$$k(x,y) = \max(x,y)$$

Benchmark



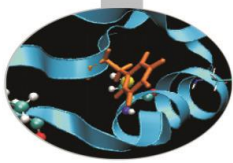
- Il problema viene discretizzato attraverso uno schema alle differenze finite:

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

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

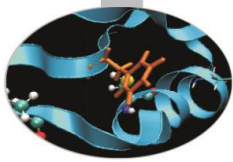
- Confrontiamo i tempi di calcolo ottenuti con diverse implementazioni dello stesso problema

Python+Numpy



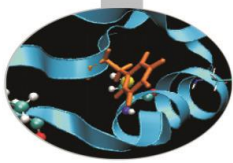
```
def calculate_u(dt, dx, dy, u, um, up, k):
    hx = (dt/dx)**2
    hy = (dt/dy)**2
    for i in xrange(1, u.shape[0]-1):
        for j in xrange(1,u.shape[1]-1):
            k_c = k[i,j]
            k_ip = 0.5*(k_c + k[i+1,j])
            k_im = 0.5*(k_c + k[i-1,j])
            k_jp = 0.5*(k_c + 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
```

Python+Numpy



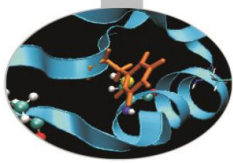
```
#Definizione delle varibili
m = 250; n = 250 # grid size
dx = 1.0/m
dy = 1.0/n
k = zeros((m+1, n+1))
up = zeros((m+1, n+1))
u = zeros((m+1, n+1))
um = zeros((m+1, n+1))
A=2
xc=yc=0.5
sx=sy=0.15
```

Python+Numpy



```
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])
x,y=meshgrid(x,y)
I=A*numpy.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())
print dt
t=0;t_stop=1.0
print 'Start'
start=time.clock()
while t <= t_stop:
        t += dt
        up = calculate_u(dt, dx, dy, u, um, up, k)
        um[:] = u
        u[:] = up
print 'Stop'
stop=time.clock()
print 'Elapsed time ', stop-start, 's'
```

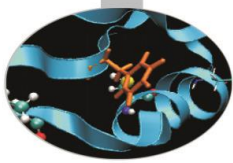
Python+Numpy+vettorizzazione



Vettorizzazione

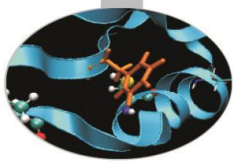
```
def calculate_u(dt, dx, dy, u, um, up, k):
    hx = (dt/dx)**2
    hy = (dt/dy)**2
    k_c = k[1:m,1:n]
    k_ip = 0.5*(k_c + k[2:m+1,1:n])
    k_im = 0.5*(k_c + k[0:m-1,1:n])
    k_jp = 0.5*(k_c + k[1:m,2:n+1])
    k_jm = 0.5*(k_c + k[1:m,0:n-1])
    up[1:m,1:n] = 2*u[1:m,1:n] - um[1:m,1:n] +hx*(k_ip*(u[2:m+1,1:n] -
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
```

F2PY



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

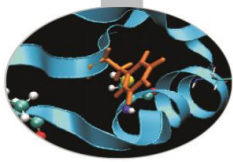
F2PY



```
import numpy
from numpy import *
import time
import mymod

 ...
print 'Start'
start=time.clock()
while t <= t_stop:
      t += dt
      up = mymod.calculate_u(dt, dx, dy, u, um, k)
      um[:] = u
      u[:] = up
print 'Stop'
stop=time.clock()
print 'Elapsed time ', stop-start, 's'
```

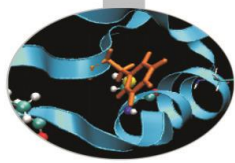
Fortran Function

Cython



```
#Compute.pyx
import numpy as np
cimport numpy as np
cimport cython
DTYPE=np.float
ctypedef np.float_t DTYPE_t
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
    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
```

Contronto



Implementazione	Elapsed Time (s)
Python + Numpy	382,32
Python+ Numpy+vect	1.52
F2PY	0.26
Cython	0.25

Griglia di calcolo 250 X 250

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