Cython

Xavier JUVIGNY

ONERA

3 Juillet 2017

Plan du cours

Prérequis et finalité du cours

Introduction

Utilisation basique de Cython et production

Prérequis et finalité de la présentation

Qui peut-être interessé par cette présentation

- Certaines parties critiques de votre codes ne peuvent être optimisées par numpy, scipy ou Panda;
- Nécessité d'un langage plus rapide mais avec la même maturité et les mêmes outils de support;
- Nécessité d'encapsuler une librairie C ou C++ indispensable pour votre application Python sans devoir s'investir dans l'API C de Python;

Prérequis

- Une bonne expérience en numpy
- Une bonne notion du C, voire du C++

Cython : l'essentiel

Caractéristiques de Cython

- *Cython* langage proche de Python permettant de mélanger du Python avec les déclarations statiques des types en C ou C++;
- *Cython* compilateur traduisant du code source Cython en C ou C++ efficace qui peut être compiler pour obtenir des modules Python optimisés en C

Pourquoi mélangé du C avec Python ?

- **Python** : langage de haut niveau, dynamique, facile à apprendre et flexible;
- **Python** : langage interprété et beaucoup plus lent que les langages statiques compilés
- **C** : Langage de bas niveau, compilateurs très optimisés donnant des exécutables très rapides;
- **C** : Pas de vérifications durant l'exécution, difficile à utiliser.

Cython : utilisation

Utilisation

- Permet avec “peu de modifications” de transformer du code Python en code Cython pour atteindre la vitesse du C;
- Permet d'interfacer facilement du code C ou C++ déjà existant pour l'utiliser dans un module ou un programme Python.

Quand ne pas utiliser Cython ?

Dans votre code :

- Si les boucles Python peuvent être remplacer par des vecteurs numpy;
- Si ce n'est pas une partie critique en temps du code;
- Si il existe déjà un module Python optimisé et proposant déjà la même chose que ce que vous voulez faire...
- Si la performance du code est déjà limité par l'accès réseau, les entrées–sorties, etc.

Un premier exemple “trivial”

Programme Python original

```
def fibonacci( n ) :
    """
    Calcul la suite de Fibonacci : u_{n+1} = u_{n} + u_{n-1}
    avec u_{0} = 0, u_{1} = 1
    """
    a = 0.0
    b = 1.0
    for i in range(n):
        a,b = a + b, a
    return a
```

Mesure de temps

Sur un Intel I7, 2GHz :

n	Temps (en μs)
90	4
180	8

Un premier exemple “trivial” (suite)

Programme C de comparaison

```
double fibonacci( int n )
{
    double a = 0.0, b = 1.0;
    for (std::size_t i = 0; i < n; ++i ) {
        double tmp = a;
        a = a + b;
        b = tmp;
    }
    return a;
}
```

Mesure de temps

Sur un Intel I7, 2GHz :

n	Temps (en μs)	Rapport temps (Python $= 1$)
90	0.06	66
180	0.13	62

Un premier exemple “trivial” (suite)

Programme Cython

```
def fibonacci( int n ) :
    """
    Calcul la suite de Fibonacci : u_{n+1} = u_{n} + u_{n-1}
    avec u_{0} = 0, u_{1} = 1
    """
    cdef double a = 0.0
    cdef double b = 1.0
    cdef int    i
    for i in range(n):
        a,b = a + b, a
        return a
```

Mesure de temps

Sur un Intel I7, 2GHz :

n	Temps (en μs)	Rapport temps (Python = 1)
90	0.03	132
180	0.11	72

Conclusion du premier exemple

Efficacité de Cython

- Un réel gain de performance !
- Peu de modification dans le code python existant !
- Attention cependant : fonction “cpu-bound” idéal pour obtenir des performances en Cython !

Optimisation du code cython

C–fication du code Cython

- Degré d'indépendance du code à l'API Python;
- Évocation : `cython -a file.pyx`
- Génère une page html
- But : générer du code C indépendant de l'API python pour les sections critiques.

Exemple d'optimisation

```
def integrate(a, b, f,N=2000):
    pas = (b-a)/(N-1.)
    x   = a
    s = 0.5*(f(a)+f(b))
    for i in range(N-1):
        x += pas
        s += f(x)
    return pas*s
```

```
@cython.cdivision(True)
def integrate( double a, double b,
               f, N=2000 ):
    cdef:
        double pas, x, s
        int i
    pas = (b-a)/(N-1.)
    x   = a
    s = 0.5*(f(a)+f(b))
    for i in range(N-1):
        x += pas
        s += f(x)
    return pas*s
```

Production avec Cython

Mode explicite

- Compilé explicitement avec toute utilisation du module Cython;
- À l'aide des distutils de Python;
- À l'aide d'un outil de production comme Make, CMake ou Scons;

Mode Implicite

- En l'utilisant de façon interactive avec IPython;
- En utilisant le module pyximport.

compilation en deux phases

- Génération d'un code C;
- Compilation du code C généré.

Mode explicite : distutils

Utilisation de distutils

Permet un contrôle total de la production

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

setup(ext_modules=cythonize('fib.pyx'))
```

A vous de jouer

- Tester le programme `mandelbrot.py`;
- Cythonizer le module `mandelbrot.py`;
- Modifier le setup.py pour prendre en compte votre module Cython;
- Lancer le setup.py puis tester de nouveau le programme `test_mandelbrot.py`;

Mode implicite : ipython

Cython en mode interactif !

- La commande `%%cython_magic` permet d'écrire du cython dans l'interpréteur;
- Après la fin du bloc cython, compilation automatique du source en module;
- Petite pause avant de rendre la main.

```
[In 12]:%load_ext cython_magic
[In 13]:%%cython
    ...:def fib(int n):
    ...:    cdef int i
    ...:    cdef double a=0.0, b=1.0
    ...:    for i in range (n):
    ...:        a, b = a+b, a
    ...:    return a
[In 14]:fib (90)
[Out14]:2.880067194370816e+18
```

Mode implicite: compilation au vol avec pyximport

Module pyximport

- Permet d'utiliser des modules cython comme des modules python;
- Utile pour génération automatique et utilisation de modules cython;
- `import` différencie les modules cython (.pyx) des modules python (.py);
- Compilation automatique du module .pyx

```
import pyximport
pyximport.install () # À appeler avant d'importer des modules Cython
```

Mode implicite: compilation au vol avec pyximport (suite)

Utilisation de pyximport

```
>>> import pyximport
>>> pyximport.install ()
(None, <pyximport.pyximport.PyxImporter object at 0xf7174a30>)
>>> import fib
>>> fib.__file__
/home/juvigny/.pyxbld/lib-linux-x86_64-3.5/fib.so
>>> type (fib)
<class 'module'>
>>> fib.fib(90)
2.880067194370816e+18
>>> pyximport.uninstall() #Retourne à un comportement standard d'import
```

- pyximport gère les modifications : recompile un module si son `.pyx` est modifié
- Ne recompile pas à la prochaine utilisation du module dans le cas contraire.

Gestion des dépendances avec pyximport

Cas de dépendance

- Le fichier cython dépend d'autres fichiers cython;
- Le fichier cython dépend de fichiers C ou C++.
- Générer un fichier de dépendance : même nom de base que le fichier cython avec extension `.pyxdeps`
- Contient la liste des dépendances, une dépendance par ligne.

Cas de compilation avec des fichiers auxiliaires

- Le fichier Cython doit faire une édition de lien avec une bibliothèque extérieure;
- On doit compiler d'autres fichiers sources (cython/C/C++/Fortran/etc.) avec le fichier Cython;
- Créer un fichier portant le même nom de base que le fichier `.pyx` et l'extension `.pybld`.
- Ce fichier peut contenir la définition de deux fonctions :
  - `make_ext(modname, pyxfilename)` : appelée avant compilation du fichier `.pyx` avec deux chaînes de caractères en entrée : le nom du module et le nom du fichier `.pyx`. Doit retourner une instance d'un `distutils.extension.Extension`;
  - `make_setup_args` : Permet de lire un dictionnaire d'arguments supplémentaires à passer à `distutils.core.setup` pour contrôler les arguments passés au `setup`

Listing 1: fib.pyxdeps

```
c_fib.*
```

Listing 2: fib.pyxbld

```
def make_ext(modname, pyxfilename):
    from distutils.extension import Extension
    return Extension(modname,
                     sources=[pyxfilename,'c_fib.c'],
                     include_dirs=['.'])
```

Compilation d'un fichier Cython "à la main"

Génération du code C/C++

- `cython -3 fib.pyx` : Génère le fichier C `fib.c` pour compatibilité Python 3;
- `cython -3 --cplus fib.pyx` : Génère le fichier C++ `fib.cpp` pour compatibilité Python 3;
- `cython -3 -a fib.pyx` : Produit un fichier html `fib.html` annotant le fichier source.

Compilation du source C/C++ généré

```
CFLAGS=$(python-config --cflags)
LDFLAGS=$(python-config --ldflags)
gcc -c fib.c ${CFLAGS}
gcc fib.o -o fib.so -shared ${LDFLAGS}
```

Interaction de Cython avec d'autres outils de production

Cython et CMake

```
# Détecte et active Cython
include(UseCython)

# Spécifie que le source Cython devra générer du code C++
set_source_file_properties(${CYTHON_EXAMPLE_SOURCE_DIR}/src/file.pyx
                           PROPERTIES CYTHON_IS_CXX TRUE )

# Rajoute et compile le source Cython dans un module d'extension.
cython_add_module( modname fib.pyx c_fib.cpp )
```

Cython et Scons

Dans les outils proposés par défaut dans Scons, il existe *cython.py* et *pyext.py* pour étendre Scons avec un support Cython qui peut être incorporer dans votre système basé sur Scons.

Directives de compilation pour Cython

Directives globales

- directives globales dans un commentaire de directives en en-tête de module

```
# cython: noneCheck=True, boundscheck=False
```

- Ou en ligne de commande : dans ce cas, prévalent sur les commentaires de directives

```
cython --directive nonecheck=False source.pyx
```

Directives locales

- Par fonction, par bloc d'instruction ou par contexte
- Par fonction

```
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
def f():
  # ...
```

- Par contexte:

```
cimport cython
def f():
    with cython.boundscheck(False), cython.wraparound(False):
        ....
```

Options de compilation de Cython

- `binding(True/False)` : fonctions considérés comme fonctions CPython (False) ou instance de classe (True)
- `boundcheck(True,False)` : Vérifie (Vrai) ou Non (Faux) la validité des indices de tableaux
- `wraparound(True,False)` : Autorise les indices négatifs pour démarrer de la fin (Vrai)
- `initializedcheck(True,False)`: Vérifie si la memoryview est initialisée
- `nonecheck(True,False)` : Si vrai, vérifie que l'argument est None ou pas
- `overflowcheck(True,False)` : Vérifie le débordement d'entier pour les entiers C
- `overflowcheck.fold(True,False)`: Vérifie le flag de débordement d'entier du processeur
- `embedsignature(True,False)`: Rajoute la signature de la fonction dans la documentation
- `cdivision(True,False)` : Si False, jette une exception `ZeroDivisionError` et ajuste le modulo à celui de Python
- `cdivision_warnings(True,False)` : Si vrai, un warning est émis à l'exécution si le modulo utilise des valeurs négatives
- `profile(True,False)`: Rajoute du code dans le source C pour profiler le code avec Cython
- `linetrace(True,False)`: Rajoute des balises sur les lignes Cython pour le profiler ou le coverage
- `infer_types(True,False)`: Infère le type des variables à partir des valeurs données si True
- `language_level(2,3)` : Indique pour quel python (v2 ou v3) générer le code
- `c_string_type(bytes,str,unicode)` : permet de définir la coercion implicite entre `char*` et `std::string`
- `c_string_encoding(ascii,default,utf8)` : type d'encodage des caractères

Pourquoi Cython est-il plus rapide que Python ?

Interprété contre Compilé

- Python transforme le code python en du *bytecode* qu'il interprète à chaque exécution;
- Surcharge dûe à chaque passage à la conversion du bytecode en langage machine;
- À l'inverse, code C compilé directement en langage machine.
- Code compilé : gain de vitesse entre 10% et 30% entre un code python et son équivalent compilé.

Typage dynamique contre typage statique

- Langages à typage statique : type des variables fixé au moment de la compilation (C, C++, Fortran, etc.)
- Langages à typage dynamique: type des variables changent au cours de l'exécution (Python, Java, Ruby, etc.)
- Langage typage statique cinquante fois plus rapide que langage typage dynamique.

Pourquoi Cython est-il plus rapide que Python ?(suite)

Que fait python pour additionner deux valeurs a et b ?

- Interroge a pour connaître son type
- Cherche si une fonction `__add__` est définie pour ce type ou les types dont il dérive;
- Si trouvée, appelle la fonction `__add__` avec a et b en argument
- La fonction `__add__` extrait les informations nécessaires de a et b,
- Si réussi, effectue l'addition.
- Le résultat est stocké dans un objet python (nouveau ou non).

Que fait C pour additionner deux entiers a et b ?

- Retourne dans un entier le résultat de a+b

Déclaration de type statique avec cdef

Variables en cython

- Variables non typées équivalentes variables typées dynamiquement en Python.

Déclarations variables typées statiquement

- Mot clef `cdef` permet déclaration statique

```
cdef int i
cdef float k
```

- Typage proche du C
- Variables typées statiques se comportent comme variables C;
- L'opérateur = effectue des copies comme en C

Variables typées statiques

Déclaration

- Déclaration simple : `cdef double x`
- Déclaration simple avec initialisation : `cdef double y=0.`
- Déclarations multiples : `cdef int i, j = 0, k`
- Bloc de déclaration :

```
cdef:
    double x, y = 0
    long int i
    cdef size_t lgth
```

- Mot clef `static` non reconnu par Cython : en C, existence de la variable durant tout le programme...

Exemples de déclarations

Type C	Déclaration Cython
Pointeurs	`cdef int* pt_i`
Tableau statique	`cdef float tab[3][3]`
Structures et unions	`cdef tm time_struct`
Pointeurs de fonctions	`cdef void (*f)(int,double)`

Inférence automatique

Inférence des types

- Par défaut, Cython déduit le type des variables si cela ne change pas la sémantique du code;

```
def infere ():
    i=1 # pas d'inférence, peut être un entier long Python
        # si on a un overflow de l'entier long C
    d=2.0 # ok, infere comme un double
    c=3+4j # Objet python...
```

- Décorateur pour déduire avec moins de précaution :

```
cimport cython

@cython.infere_types (True)
cdef infere ():
    i=1 # entier long C. Responsabilité programmeur pour overflow
    d=2.0 # ok, double C
    c=3+4j # struct complexe C fournie par cython.
```

- Peut être mis en décorateur ou en en-tête de fichier.

Les pointeurs en Cython

Déclaration

- Similaire au C

```
cdef int *p, *q, r # Attention, r n'est pas un pointeur
cdef double** array;
```

Déférencement des pointeurs

- * impossible car utilisé pour `*tuple` et `**dict`
- On utilise à la place l'accès au premier élément d'un tableau

```
cdef double gold_number
cdef double *pt_gold
pt_gold = &gold_number # pt_gold pointe sur gold_number
pt_gold[0] = 3.0
```

Interopérabilité entre types dynamiques et statiques

Interopérabilité

Possilité d'utiliser des types statiques au sein de types dynamiques :

```
cdef int a,b,c
tuple_of_ints = (a, b, c)
```

Correspondance entre les types python et les types C

Type Python	Type C	Commentaire
bool	bint	`int` en C : faux si nul, vrai sinon
int	[unsigned] char	Vérification par défaut de l'overflow
long	[unsigned] short, int, long, long long	Vérification par défaut de l'overflow
float	float, double, long double	Conversion suit la norme IEEE 754
complex	float/double complex	Structure de deux réels, compatible C99 et C++
bytes str unicode	char* std::string (C++)	`c_string_type` et `c_string_encoding` activés
dict	struct	

Déclaration de type statique avec un type Python

Déclaration statique de types pythons

- Seulement si le type a été programmé en C;
- Ce qui est le cas des types builtins

```
cdef list particles, modified_particles
cdef dict names_from_particles
cdef str pname
cdef set unique_particles
particles = list(names_from_particles.keys())
other_particles = particles # other_particles -> variable typée dynamiquement
del other_particles[0] # Détruit le premier élément de particles !
```

Types Python par défaut supportés en statique

- `type`, `object`, `bool`, `complex`
- `basestring`, `str`, `unicode`, `bytes`, `bytearray`
- `list`,`tuple`,`dict`,`set`,`frozenset`
- `array`, `slice`
- `date`,`time`,`datetime`,`timedelta`,`tzinfo`

Compteur de référence et chaîne de caractères statiques

Garbage collector

- Compteur de référence;
- G.C détruit les objets sans références périodiquement.

Conséquence sur le code Cython

```
b1 = b"Tous les chats sont mortels"
b2 = b"Socrate est mortel..."
cdef char *buf = b1 + b2
# Ne marche pas car
# b1+b2 est un objet temporaire
# de type byte si bien que buf
# pointerait sur une zone invalide.
# Heureusement, dans ce cas, Cython
# génère une erreur !
```

```
cdef bytes tmp = s1 + s2
cdef char *buf = tmp
# Ca marche cette fois
# à condition que tmp ne
# soit pas détruit avant
#  le pointeur C !
# Le C n'a aucun moyen
# de savoir si un objet Python
# a encore une référence ou non...
```

Fonctions Python en Cython

Déclaration et définition

- On déclare la fonction à l'aide du mot clef `def`
- Les paramètres peuvent être déclarés statiquement dans la signature de la fonction
- Dans ce cas, on omet d'utiliser le mot clef `cdef`.

```
def fact(long n):
    """Computes n!"""
    if n <= 1 :
        return 1
    return n * fact(n-1)
# On ne gagne pas beaucoup de temps à typer n
# Car on retourne un objet python et donc on passe
# beaucoup de temps dans le retour de la fonction...
# Le problème vient donc du fait qu'on appelle
# récursivement une fonction Python.
```

Fonctions C en Cython

Déclaration et définition

- On déclare une fonction C à l'aide du mot clef `cdef`;
- Génère une fonction pure C
- On peut y manipuler des objets Python mais antagoniste à l'idée d'optimisation
- Ne peut pas être appelée par une fonction Python non définie via Cython
- Possibilité d'inline pour la fonction

```
cdef long c_fact(long n):
    """Computes n!"""
    if n<=1 :
        return 1
    return n * c_fact(n-1)

def wrap_c_fact( long n ):
    """Computes n!"""
    return c_fact(n)
```

Combiner fonction Python et fonction C

Déclarer une combinaison d'une fonction C/Python

- Une fonction qui se décline automatiquement en deux versions : une pure C et une wrappant la fonction C,
- Les deux fonctions générées portent le même nom.
- Possibilité d'inline pour la fonction (pour la version pure C)

```
cpdef inline long cp_fact(long n):
    """Computes n!"""
    if n <= 1 :
        return 1
    return n * cp_fact(n-1)
```

Gestion des exceptions dans les fonctions C et C/Python

Problèmatique

- Une fonction Python retourne toujours un objet Python : permet de gérer facilement les exceptions
- Pour les fonctions C ou C/Python : pas possible de remonter l'exception à l'appelant;
- Il faut utiliser une clause d'exception : soit retourner un entier particuliers soit Cython gère une exception (plus couteuse)

```
# Entier particuliers pour signaler l'erreur ( ici -1 )
# C'est Cython lui-même qui ensuite initialise cet entier
# pour provoquer une exception
cpdef int divide_ints(int i, int j) except ? -1:
    return i / j
# Ou Cython gère directement une exception :
cpdef int divide_ints(int i, int j) except *:
    return i / j
```

Embarquer signature de fonction dans documentation

Documentation engendrée par Cython

- Dans la documentation d'une fonction python pure, la signature de la fonction est donnée;

```
>>> help(pfib.fibonacci) # pfib est le module python pure de fibonacci
Help on function fibonacci in module pfib:

fibonacci(n)
    Calcul la suite de Fibonacci : u_{n+1} = u_{n} + u_{n-1}
    avec u_{0} = 0, u_{1} = 1
```

- Dans la documentation d'une fonction cython, elle n'est pas donnée par défaut

```
>>> help(cfib.fibonacci) # cfib est le module cython de fibonacci
Help on built-in function fibonacci in module cfib:

fibonacci(...)
    Calcul la suite de Fibonacci : u_{n+1} = u_{n} + u_{n-1}
    avec u_{0} = 0, u_{1} = 1
```

- Il suffit de mettre `embedsignature` à True pour que cela soit maintenant le cas.

```
>>> help(cfib.fibonacci) # cfib est le module cython de fibonacci
Help on built-in function fibonacci in module cfib:

fibonacci(...)
    fibonacci(int n)

    Calcul la suite de Fibonacci : u_{n+1} = u_{n} + u_{n-1}
    avec u_{0} = 0, u_{1} = 1
```

Coercion de type et convertion

Coercion de type

- Uniquement pour les types statiques : même règles qu'en C

Convertion entre types

- Opérateur de convertion similaire au C
- On remplace simplement les parenthèses par des crochets

```
cdef int *ptr_i = <int*> v
def print_address( a ):
    cdef void *v = <void*>a
    cdef long addr = <long>v
    print("Cython adress : {}".format(addr))
    print("Python id     : {}".format(id(a)))
```

- On peut convertir des objets en type python, déjà pre-existant ou défini par nous même

```
def cast_to_list(a):
    cdef list cast_list = <list>a # Si a n'est pas un sous-type de list
                                  # ou une liste, on a une erreur système !
    cast_list.append(1)
```

- On peut demander auparavant à Cython de vérifier le type de a :

```
def cast_to_list(a):
    cdef list cast_list = <list?>a # Si on ne peut convertir en liste,
                                   #   on a un TypeError
    cast_list.append(1)
```

Définir des structures et des unions en Cython

Déclaration d'une nouvelle structure/union

Syntaxe mixte entre du Python et du C :

```
ctypedef double reel
ctypedef struct icplx:
    int i_real
    int i_imag
ctypedef union repr_float:
    double val
    char   real_repr[8]
cdef icplx iz1(1,1)
cdef icplx iz2(i_real = 3, i_imag = -2)
iz1.i_real = -1
cdef icplx iz3 = { 'i_real' : -4, 'i_imag' = 3}
```

mais impossible de déclarer des structures emboîtées.

Déclaration d'énumérés

```
cdef enum PRIMARIES:
    RED    = 1
    YELLOW = 3
    BLUE   = 5
cdef enum SECONDARIES:
    ORANGE, GREEN, PURPLE
cdef enum: # Anonymous enum for constants
    GLOBAL_SEED = 37
```

Types fusionnés

Types fusionnés

- Permet de définir des types génériques regroupant plusieurs autres types;
- Cython prédéfinit trois types fusionnés : `integral`, `floating` et `numeric`
- On peut définir soi-même des types fusionnés.

```
from cython cimport integral

cpdef integral i_max( integral a, integral b ):
    return a if a >= b else b

ctypedef fused ordered_num:
^^Icython.short
^^Icython.int
^^Icython.long
^^Icython.float
^^Icython.double
```

Cython et les types extensions

Types extensions

- Comme une classe Python mais écrite en C avec l'API CPython;
- Permet d'avoir une classe optimisée
- Demande une bonne connaissance du C et de l'API CPython !

Types extensions avec Cython

- Programmation proche d'une classe Python pure

```
class Particle(object):
    """Simple particle type"""
    def __init__(self, mass, pos, vel ):
        self.mass     = mass
        self.position = pos
        self.velocity = vel
    def comp_momentum(self):
        return self.mass * self.velocity
```

```
cdef class Particle:
    """Simple particle extension type"""
    # Accessibles dans les méthodes mais
    # pas à l'extérieur de la classe !
    cdef double mass, position, velocity

    def __init__(self, mass, pos, vel ):
        self.mass     = mass
        self.position = pos
        self.velocity = vel
    def comp_momentum(self):
        return self.mass * self.velocity
```

Contrôle d'accès aux attributs

Permission d'accès

```
cdef class Particle:
    """Simple particle extension type"""
    # Accessible en lecture/écriture
    cdef public double mass
    # Accessible en lecture seule !
    cdef readonly double velocity
    # Attribut non accessible
    cdef double position
    ...
```

Initialisation attributs C

- Méthode `__cinit__` responsable allocation et initialisation attributs C
- Évite des problèmes de double appel dans le cadre de l'héritage
- La méthode `__dealloc__` reponsable de déallouer attributs C

```
cdef class Matrix:
    cdef readonly unsigned  int nrows, ncols
    cdef           double* _matrix

    def __cinit__(self, nr, nc):
        self.nrows = nr; self.ncols = nc
        self._matrix = <double*>malloc(nr * nc * sizeof(double))
        if self._matrix == NULL:
            raise MemoryError()
    def __dealloc__(self):
        if self._matrix != NULL:
            free(self._matrix)
```

Extension des properties en Cython

Getter et Setter

- Possibilité de définir des attributs dérivés commen en Python
- Par contre, ne permet pas de définir une fonction C pour optimisation

```
cdef class Particle:
    """Simple particle extension type."""
    ...
    property momentum:
        """Momentum value of the particle"""
        def __get__(self):
            """Getter of the momentum value"""
            return self.mass * self.velocity
        def __set__(self, double m) :
            """Momentum setter"""
            self.velocity = m / self.mass
```

Opérateurs en Cython

Opérateurs arithmétiques

- Les types d'extension ne supportent pas `__radd__`
- C'est l'opérateur `__add__` qui doit gérer les deux cas de figure

```
cdef class E:
    cdef int data
    def __init__(self, int d):
       self.data = d
    def __add__(x,y):
        if isInstance(x, E):
            if isInstance(y, int): return (<E>x).data + y
        elif isInstance(y, E):
            if isInstance(x, int): return (<E>y).data + x
```

Opérateurs de comparaison

- Cython ne supporte qu'un opérateur de comparaison général `__richcmp__(x, y, op)`
- op donne le type de comparaison voulu comme dans le tableau suivant :

Argument entier	Py_LT	Py_LE	Py_EQ	Py_NE	Py_GT	Py_GE
Opérateur	$<$	$\leq$	$==$	$\neq$	$>$	$\geq$

```
cdef class E: ...
    def __richcmp__(x,y, int op):
        cdef E e
        e, y = (x,y) if isInstance(x, E) else (y,x)
        if op == Py_LT: return e.data < y
        elif op == Py_LE: ...
        else: assert False
```

Organisation en modules

Les modules

- Comme Python, Cython permet de gérer notre projet en modules;
- Permet aussi à deux modules Cython d'accéder à leur couche C (`cdef`, `cpdef`, `ctypedef`,...)
- Trois types de fichiers pour cela :
  - Les fichiers avec extension `.pyx` pour la mise en œuvre
  - Les fichiers avec extension `.pxd` pour les définitions
  - Les fichiers avec extension `.pxi` pour les inclusions
- La commande `cimport` permet d'accéder aux constructions C d'un autre module Cython à l'aide du fichier de définitions.
- Dès qu'il y a en jeu plus d'un module Cython, une organisation en fichiers de définitions, mise en œuvre est nécessaire.

Exemple d'organisation

Une simulation physique d'un nuage de particules

Un aspect de fichier de mise en œuvre initial

```
ctypedef double real_t

cdef class Particles:
    cdef:
        unsigned long nbParticles
        real_t *pos_x, *pos_y
        real_t *vel_x, *vel_y

    def __cinit__( ... ):
        ...
    def __dealloc__( ... ):
        ...
    cpdef real_t momentum(self):
        ...

def setup( input_fname ):
    ....
cpdef run(Particles p):
    ....
cpdef int step(Particles p, real_t timestep ):
    ....
def output(Particles p):
    ....
```

Exemple d'organisation (suite)

Organisation en fichier de définition et de mise en œuvre

Fichier de définition `Particles.pxd`

```
ctypedef double real_t

cdef class Particles:
    cdef:
        unsigned long nbParticles
        real_t *pos_x, *pos_y
        real_t *vel_x, *vel_y

    cpdef real_t momentum(self)

cpdef run(Particles p)

cpdef int step(Particles p,
               real_t timestep)
```

Fichier de mise en œuvre `Particles.pyx`

```
cdef class Particles:
    def __cinit__(...):
        ...
    def __dealloc__(...):
        ...
    cpdef real_t momentum(self):
        ...

def setup(input_fname):
    ...
cpdef run(Particles p):
    ...
cpdef int step(Particles p,
               real_t timestep):
    ...
def output(Particles p):
    ...
```

Que contient un fichier de définition ?

Ce qu'il peut contenir

Qu'on veut mettre en public :

- Des déclarations de type C (structure, union, enum)
- Déclarations pour des librairies externes C/C++
- Déclarations pour des fonctions du module définies `cpdef` ou `cdef`
- Déclaration d'un type d'extension (`cdef class`)
- Les attributs C du type d'extension (`cdef`)
- La déclaration des méthodes déclarées `cdef` ou `cpdef`
- La mise en œuvre de fonctions ou méthodes C inline

Ce qu'il ne peut pas contenir

- La mise en œuvre des fonctions et méthodes Python et C non inlinées
- La définition des classes Python pure
- Du code exécutable Python en dehors de macros

Un autre module Cython peut appeler les services du module à l'aide de `cimport`

Définition d'une bibliothèque C externe

Exemple (partiel) sur CBlas

Définition de `CBlas.pxd`

```
cdef extern from "cblas.h":
    # Blas 1
    # Produit scalaire
    float  cblas_sdsdot(const int N, const float alpha, const float *X,
                        const int incX, const float *Y, const int incY)
    double cblas_dsdot(const int N, const float *X, const int incX, const float *Y,
                       const int incY)
    float  cblas_sdot(const int N, const float  *X, const int incX,
                      const float  *Y, const int incY)
    double cblas_ddot(const int N, const double *X, const int incX,
                      const double *Y, const int incY)
    ...
```

Utilisation de CBlas dans un fichier cython :

```
from CBlas cimport cblas_ddot
...
x = cblas_ddot( ... )
```

Fichiers de définition prédéfinis

En-têtes C/C++ définis en Cython

- Cython livré avec plusieurs fichiers pxd définissant des entêtes communs C/C++ ou CPython
- Contient un package *libc* pour les entêtes standards : *stdio*, *math*, *string*, *stdint...*
- Contient un package *libcpp* contenant des entêtes de la STL : *string*, *vector*, *list*, *map*, *pair et set*
- Contient un package *cpython* contenant la définition des fonctions servant à l'API Python.

Exemples

- Utiliser `cimport` avec un module dans un package

```
from libc cimport math as cmath
cmath.sin(3.14)
```

item Utiliser `cimport` avec plusieurs objets d'un module

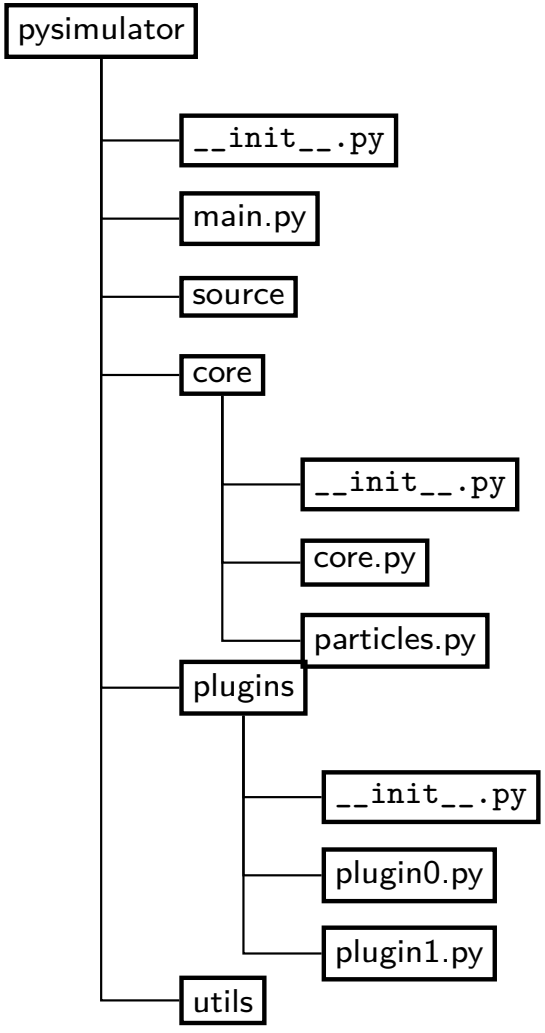
```
from libc.stdlib cimport rand, srand, qsort, malloc, free
cdef int* a = <int*>malloc(10 * sizeof(int))
```

- Utiliser `cimport` avec un classe C++ de la STL

```
from libcpp.vector cimport vector
cdef vector[int] *vi = new vector[int](10)
```

Organisation et compilation de modules Cython au sein d'un package Python

Mise en œuvre initiale (Python pure)



Profiling du code

Après profiling, on voit qu'il faut :

- Optimiser les fichiers `core.py`, `particle.py` et `plugin0.py`;
- `particles.py` contient la classe `Particles` qu'il faudra convertir en type extension;
- `core.py` contient les fonctions `run` et `step` à convertir en `cpdef` fonctions;
- `plugin0.py` contient une fonction `run` à convertir en `cpdef` fonction.

Travail à prévoir

- Transformer tout ces fichiers en fichiers de mise en œuvre cython;
- Extraire leurs déclarations publics Cython pour les mettre dans des fichiers de définition cython;

Organisation et compilation de modules Cython au sein d'un package Python (Définitions)

particules.pxd

```
ctypedef double real_t

cdef class Particules:
    cdef:
        unsigned long nbParticules
        real_t *x, *y
        real_t *vx, *vy

    cpdef real_t momentum(self)
```

core.pxd

```
from simulator.core.particules cimport Particules, real_t

cpdef int run(Particule, list plugins=None)
cpdef step(Particules p, real_t dt)
```

plugin0.pxd

```
from simulator.core.particules cimport Particules

cpdef run(Particules p)
```

Production du package

setup.py

```
from disutils.core import setup
from Cython.build  import cythonize

setup(name="simulator",
      packages = ["simulator", "simulator.core", "simulator.utils",
                  "simulator.plugins"],
      ext_modules=cythonize("**/*.pyx"),
     )
```

- `cythonize` est appelé avec un pattern global pour chercher récursivement tous les fichiers de mise en œuvre pyx et les compiler si nécessaire;
- `cythonize` gère les dépendances et recompile si besoin;
- Il détecte les interdépendances entre les fichiers de mise en œuvre et de définition, et recompile tout fichier de mise en œuvre dépendant.

Déclarer du code C externe en Cython

Canevas de déclaration

```
cdef extern from "header_name":
    indented declaration from header file
```

Ce qui change par rapport à un header C

- Remplacer `typedef` par `ctypedef`;
- Enlever des mots clefs non nécessaires ou non supportés comme `restrict` ou `volatile`;
- S'assurer que le type de retour de la fonction soit sur la même ligne que le nom de la fonction;
- Enlever à la fin de la ligne le point-virgule;

Listing 3: header.h

```
#define M_PI 3.1415926
#define MAX(a,b) ((a)>=(b) ? (a):(b))

double hypot(double, double);
typedef int    integral;
typedef double real_t;

void func(integral, integral, real);
real*
func_arrays(integral[],integral[][10],
            real **);
```

Listing 4: header.pxd

```
cdef extern from "header.h":
    double M_PI
    float MAX(float a, float b)

    double hypot(double x, double y)
    ctypedef int    integral
    ctypedef double real_t

    void func(integral a,integral b,
              real c)
    real* func_arrays(integral[] i,
                      integral[][10] j,
                      real **k)
```

Encapsulation des fonctions et structures C

Déclarer et encapsuler des structures, unions ou énumérés

```
cdef extern from "header_name":
    struct struct_name:
        struct_members

    ctypedef struct struct_alias:
        struct_members

    union union_name:
        union_members

    enum enum_name:
        enum_members
```

Encapsuler des fonctions C

Listing 5: mt_random.pyx

```
cdef extern from "mt19937ar.h":
    void init_genrand(unsigned long s)
    double genrand_real1()

def init_state(unsigned long s):
    init_genrand(s)

def rand():
    return genrand_real1()
```

Listing 6: setup.py

```
from distutils.core import setup,Extension
from Cython.Build import cythonize
ext = Extension(
    "mt_random",
    sources = ["mt_random.pyx",
               "mt19937ar.c"])
setup(
    name ="mersenne_random",
    ext_modules = cythonize([ext]) )
```

Encapsuler dans des extensions de type

Header C et définition cython

Listing 7: mt19937ar-struct.h

```
typedef struct _mt_state mt_state;

mt_state *make_mt(unsigned long s);
void free_mt(mt_state *state);
double genrand_real1(mt_state* state);
```

Listing 8: mt_struct.pxd

```
cdef extern from "mt19937ar-struct.h":
  ctypedef struct mt_state
  mt_state *make_mt(unsigned long s)
  void free_mt(mt_state *state)
  double genrand_real1(mt_state *state)
```

Encapsulation dans une extension de type

Listing 9: mt_random_type.pyx

```
cimport mt_struct
cdef class MT:
    cdef mt_struct.mt_state *_thisPtr
    def __cinit__(self, unsigned long s):
        self._thisPtr = mt_struct.make_mt(s)
        if self._thisPtr == NULL: raise MemoryError("Insufficient memory")
    def __dealloc__(self):
        if self._thisPtr != NULL: mt_struct.free_mt(self._thisPtr)
    cpdef double rand(self):
        return mt_struct.genrand_real1(self._thisPtr)
```

Encapsulation d'une classe C++

Définition de la classe C++ en Cython

Listing 10: mt19937.h

```
namespace mtrandom {
    const static unsigned int N = 624;
    class MT_RNG {
    public:
        MT_RNG();
        MT_RNG(unsigned long s);
        MT_RNG(unsigned long init_key[],
               int key_length);
        void init_genrand(unsigned long s);
        unsigned long genrand_int32();
        double genrand_real1();
        double operator()() {
                return genrand_real1();
        }
    private:
        unsigned long mt[N];
        int mti;
    };
}
```

```
cdef extern from "mt19937.h"
            namespace "mtrandom":
    unsigned int N
    cdef cppclass MT_RNG:
        MT_RNG() except +MemoryError
        MT_RNG(unsigned long s) except +
        MT_RNG(unsigned long init_key[],
               int key_length) except +
        void init_genrand(unsigned long s)
        unsigned long genrand_int32()
        double genrand_real1()
        double operator()()
```

Encapsulation d'une classe C++ (suite)

Définition d'une extension de type

```
cdef class RNG:
    cdef MT_RNG *_thisptr
    def __cinit__(self, unsigned long s):
        self._thisptr = new MT_RNG(s)
        if self._thisptr == NULL:
            raise MemoryError()
    def __dealloc__(self):
        if self._thisptr != NULL:
            del self._thisptr
    cpdef unsigned long randint(self):
        return self._thisptr.genrand_int32()
    cpdef double rand(self):
        return self._thisptr.genrand_real1()
    def __call__(self):
        return self._thisPtr[0]()
```

Production

Listing 11: setup.py

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

ext = Extension("RNG",
                sources=["RNG.pyx", "mt19937.cpp"],
                language="c++")

setup(name="RNG", ext_modules=cythonize(ext))
```

Cython et les templates C++

Template et fused types

Listing 12: algorithm

```
template<typename T> const T& min(const T& a, const T& b);
template<typename T> const T& max(const T& a, const T& b);
```

- Déclaration proche en Cython

Listing 13: _algorithm.pxd

```
cdef extern from "<algorithm>" namespace "std":
    const T min[T](T a, T b) except+
    const T max[T](T a, T b) except+
```

- Fused types idéal pour les templates :

```
cimport cython
cimport _algorithm

ctypedef fused long_or_double:
    cython.long
    cython.double

def min( long_or_double a, long_or_double b ):
    return _algorithm.min(a,b)
def max( long_or_double a, long_or_double b ):
    return _algorithm.max(a,b)
```

Cython et la STL

Librairie STL supportée par Cython

- string
- pair
- vector
- list
- map
- queue
- set
- priority_queue
- unordered_map
- deque
- unordered_set
- stack

Exemple d'utilisation

```
from libcpp.string cimport string
from libcpp.map cimport map
from libcpp.pair cimport pair

def periodic_table():
  cdef map[string,int] table
  cdef pair[string,int] entry
  # Insérer l'hydrogène:
  entry.first = b"H"; entry.second = 1
  table.insert(entry)
  # Insérer l'Hélium:
  entry.first = b"He"; entry.second = 2
  table.insert(entry)
  ...
  # Plus facile :
  table = { "H":1, "He":2, "Li":3}
  ...
  return table
```

memoryview en python

Définition

- Permet de voir les données internes d'un objet Python qui supporte le protocole buffer sans les copier
- Exemple :

```
>>> b = b"Buffer froid"
>>> v = memoryview(b)
>>> v[0]
66
>>> v[2:4]
<memory at 0x7f6b595d51c8>
>>> bytes(v[2:4])
b'ff'
```

- Le type array de Python vérifie également le protocole :

```
>>> import array
>>> a = array("d")# Tableau de double
>>> a.fromlist([1,3,5,7,11])
>>> av = memoryview(a)
>>> av[2]
5.0
```

- Il en est de même pour les tableaux numpy

```
>>> import numpy as np
>>> a = np.array([1,3,5,7,11],dtype=np.double)
>>> av = memoryview(a)
>>> av[2]
5.0
```

Cython et memoryview

Memoryview dans Cython

- Cython a un memoryview typé au niveau C
- Étend la notion de memoryview de Python
- Permet de traiter tout objet python ayant le protocol buffer et les tableaux C ou Fortran
- Déclaration d'un memoryview, contient :
  - Son type (dont fused type)
  - Ses dimensions (exemple `double[:,:,:]`)
  - Contigües ou par stride
  - Fortran ou C convention
- Exemples de déclarations :

```
cdef float[:,::1] c_contig_mv # Vue contigue
```

- Exemple d'utilisation :

```
def summer(double[:] mv):
    """Somme les valeurs contenues dans l'argument"""
    cdef:
        int i, N
        double ss = 0.0
    N = mv.shape[0]
    for i in range(N):
        ss += mv[i]
    return ss
```

Utilisation des memoryview avec des tableaux C

```
cdef int a[3][5][7]
cdef int[:,:,::1] mv = a
mv[...] = 0 # Met tous les coefficients de a à zéro

from libc.stdlib cimport malloc
def dynamic(size_t N, size_t M):
    cdef long *arr = <long*>malloc(N * M * sizeof(long))
    cdef long[:,::1] mv1 = arr # ERREUR, ne compile pas, arr pas vu en 2D
    cdef long[:,::1] mv2 = <long[:N,:M]>arr
    ...
```

Gestion correcte des tableaux C en Cython (avec Numpy)

Problématique

- Supposons que nous avons une fonction C qui retourne une matrice qu'il alloue :

```
cdef extern from "matrix.h":
    float* make_matrix_c( int nrows, int ncols )
import numpy as np
def make_matrix(int nrows, int ncols):
    cdef float[:,::1] mv = <float[:nrows,:ncols]>make_matrix_c(nrows,ncols)
    return np.asarray(mv)
```

- Le problème de ce code est qu'il ne libère jamais la mémoire prise par la matrice
- Créer un objet se chargeant de libérer place mémoire en utilisant fonction `set_array_base` de l'API numpy

```
import numpy as np
cimport numpy as cnp
cdef class _finalizer:
    cdef void* _data
    def __dealloc__(self):
        if self._data is not NULL:
            free(self._data)
cdef void set_base(cnp.ndarray arr, void* carr):
    cdef _finalizer f = _finalizer()
    f._data = <void*>carr
    cnp.set_array_base(arr,f)
def make_matrix(int nrows, int ncols):
    cdef float* mat = make_matrix_c(nrows,ncols)
    cdef float[:,::1] mv = <float[:nrows,:ncols]>mat
    cdef cnp.ndarray arr = np.asarray(mv)
    set_base(arr,mat)
    return arr
```

Parallélisme multithreadé et le GIL

Problématique de GIL

- Le GIL (Global Interpreter Lock) oblique qu'un seul thread principal exécute du Bytecode Python;
- Le GIL est nécessaire seulement pour aider à la gestion mémoire des objets python;
- Du code C ne travaillant pas avec des objets Python peuvent être exécuter sans le GIL;
- Le GIL est spécifique à CPython. Il n'existe pas dans d'autres Python : Jython, IronPython et Pypy
- On peut demander à Cython d'outrepasser le GIL pour du parallélisme dans certaines parties du code;
- Pour cela, il faut s'assure de ne pas utiliser ou retourner des objets Python.

Enlever le GIL dans Cython

- Déclarer une fonction multithreadable :

```
cdef int kernel(double complex z, double z_max, int n_max ) nogil:
    ...
```

- Dans la déclaration de fonctions externes C :

```
cdef extern from "math.h" nogil:
    double sin(double x)
    double cos(double x)
```

- Pour un bloc d'instruction :

```
with nogil: # On désactive la gil
    result = kernel(z, z_max, n_max)
    ...
print result # gil de nouveau activé
```

Boucles parallèles

Parallélisation de boucle

- Utilise la fonction prange : `from cython.parallel cimport prange`
- Exemple :

```
# distutils: extra_compile_args = -fopenmp
# distutils: extra_link_args    = -fopenmp
from cython.parallel cimport prange

def calc_julia(...):
    # ...
    for i in prange(resolution+1, nogil=True ):
        real = bound + i * step
        for j in range(resolution+1):
            ....
```

- Options pour prange :
  - Comme range, on peut spécifier le début, la fin et le pas
  - Changer le schedule pour la boucle comme OpenMP : `static, guided, dynamic`
- Les réductions se font de manière automatique:

```
def calc_julia(...):
    # ...
    for i in prange(resolution+1, nogil=True ):
        real = bound + i * step
        for j in range(resolution+1):
            ....
            if iter == iterMax : total += 1
    return total/float(resolution*resolution)
```