# TD Calcul parallèle

MACS 2$^{\text{ième}}$ année

Octobre 2009

Ce TD est prévu pour deux séances. Il faudra rendre les programmes à la fin de la deuxième séance et une bref rapport d'une page sur papier ou sous format électronique avant la séance suivante...

Le but de ce TD est d'une part mettre en œuvre l'algorithme de Gram–Schmidt ( la version modifiée qui est plus stable que la version originale ) permettant de calculer une base orthonormale engendrant le même sous–espace qu'une famille $U$ de vecteurs $u_i$ donnée en entrée.

Pour mémoire, on rappelle ici l'algorithme de Gram–Schmidt modifié :

- En entrée : $U$ une famille libre de $m$ vecteurs de dimension $n$ ;
- Initialisation de la base orthonormée $V$ à une liste ne contenant aucun vecteur ;
- Pour chaque $u_i$ dans $U$ :
  - $\widetilde{v_i} = u_i - \sum_{j=1}^{i-1}(u_i, v_j).v_j$
  - Si $\|\widetilde{v_i}\|$ est nulle, $U$ est une famille liée. On lève une exception de type ValueError ;
  - Rajouter le vecteur $v_i = \frac{\widetilde{v_i}}{\|\widetilde{v_i}\|_2}$ à la base orthonormée $V$.

## 1 Création d'un module définissant une classe vecteur

Dans un premier temps, nous allons compléter le module `vector` avec les fonctions suivantes ( qu'on devra documenter ) :

- Compléter la fonction `__init__` permettant de créer un vecteur de dimension $n$
- Compléter la fonction qui renvoie l'addition et la soustraction de deux vecteurs $u$ et $v$ ;
- Compléter la fonction qui renvoie l'homothétie de rapport $\alpha$ d'un vecteur $u$ ;
- Compléter la fonction renvoyant le produit scalaire de deux vecteurs $u$ et $v$ ;
- Compléter la fonction renvoyant la norme L2 d'un vecteur $u$ ;

Dans chaque cas, s'assurer que les paramêtres données à la fonction sont compatibles entre eux ( par exemple, pour l'addition, vérifier que $u$ et $v$ ont même dimension ). Si le cas échéant, les deux paramêtres sont incompatibles, renvoyer une exception de type `ValueError` avec un message expliquant le problème.

Dans cette première version, on se contentera d'écrire des boucles remplissant une liste avec le résultat désiré.

Par exemple, si on devait écrire une méthode calculant la valeurs absolue des cœfficients du vecteur courant, on aurait :

Par exemple, si on avec une fonction `vect_abs` retournant la valeurs absolue des coefficients d'un vecteur, on l'écrira comme suit :

```
def abs( self ) :
    """
␣␣␣Retourne␣le␣vecteur␣contenant␣la␣valeur␣absolue
␣␣␣de␣chaque␣coefficient␣du␣vecteur␣courant
␣␣␣"""
    coefs = [None]*self.dim()
    for i in xrange(self.dim()) :
      coefs[i] = abs(self.coefs[i])
    return vector(coefs=coefs)
```

Tester la classe ainsi crée (le test est inclu dans le fichier `vector.py`).

# 2 Algorithme de Gram-Schmidt

Dans un deuxième fichier, en utilisant le module précédemment écrit, mettre en œuvre l'algorithme de Gram–Schmidt décrit au début du TP sous forme d'une fonction `GramSchmidt` qui prendra en paramêtre une famille de vecteur $U$ et retournera une base orthonormée $V$ ( sous la même forme que $U$ ). Cette fonction devra être écrite dans une fichier nommé `GramSchmidt.py`.

Tester la fonction avec les deux fichiers tests `TestGramSchmidt1.py` et `TestGramSchmidt2.py` et noter les temps affichés pour `TestGramSchmidt1.py`.

# 3 Améliration de la classe vecteur

Nous allons essayer d'optimiser notre classe vecteur afin d'avoir un algorithme beaucoup plus rapide.

## 3.1 Première amélioration

Copier `vector.py` dans un nouveau fichier `vector2.py` qui sera modifié pour remplacer, dès que faire ce peut, les boucles remplissant une liste par une liste définie à l'aide d'une boucle.

Par exemple, si on avec une fonction `vect_abs` retournant la valeurs absolue des coefficients d'un vecteur, on l'écrira comme suit :

```
def abs( self ) :
    """
␣␣␣Retourne␣le␣vecteur␣contenant␣la␣valeur␣absolue
␣␣␣de␣chaque␣coefficient␣du␣vecteur␣courant.
␣␣␣"""
    return vector(coefs=[abs(c) for c in self.coefs])
```

Tester la nouvelle version de la classe `vector`.

Modifier `TestGramSchmidt1.py` afin d'inclure le nouveau fichier `vector2.py` au lieu de `vector.py`. Relancer `TestGramSchmidt1.py` et comparer les temps d'exécution par rapport à la première version.

### 3.1.1 Deuxième amélioration

Pour améliorer la version précédente, nous allons utiliser les instructions `map` et `sum` de python ainsi que le module `operator`.

- `map` permet de générer une liste à partir d'une fonction à $n$ arguments appliquée en itérant sur les éléments de $n$ listes ;
- Le module `operator` définit des fonctions basiques comme l'addition ( `add` ), soustraction ( `abs` ), etc à utiliser avec `map` ;
- `sum` permet de calculer la somme des valeurs contenues dans une liste.

Pour reprendre l'exemple de la fonction `abs`, on l'écrirait maintenant comme :

```
def abs( self ) :
    """
␣␣␣Retourne␣le␣vecteur␣contenant␣la␣valeur␣absolue
␣␣␣de␣chaque␣coefficient␣du␣vecteur␣courant
␣␣␣"""
    return vector(coefs=map(operator.abs, self.coefs))
```

Recopier `vector2.py` dans un nouveau fichier `vector3.py`, puis en s'aidant de l'aide en ligne ( `help` ) et en regardant le contenu du module `operator` ( pensez à l'instruction `dir` ), modifier la fonction de soustraction, d'homothétie et du calcul du produit scalaire.

Retester avec `TestGramSchmidt1.py` en ayant pris soin de remplacer `vector2.py` par `vector3.py` et comparer les temps d'exécution par rapport aux deux autres versions.

### 3.1.2 Dernière amélioration sans numpy

Le problème de `map` est que cette instruction génère effectivement une liste ce qui pose des problèmes de mémoire et parfois des problèmes de performance. Le module `itertools` permet de remédier à ce problème ainsi

qu'au problème de création d'une liste répétant $n$ fois un même objet ( ou valeurs ) comme dans la fonction calculant le produit scalaire en itérant sur une liste virtuelle...

Par exemple, pour reprendre la méthode `abs` :

```
def abs( self ) :
    """
␣␣␣Retourne␣le␣vecteur␣contenant␣la␣valeur␣absolue
␣␣␣de␣chaque␣coefficient␣du␣vecteur␣courant
␣␣␣"""
    return vector(coefs=list(itertools.imap(operator.abs,self.coefs)))
```

Noter l'emploie de l'instruction `list` afin de créer effectivement une liste à partir de la liste virtuelle générée par `itertools.imap(operator.abs,u)`.

Copier `vector3.py` dans un nouveau fichier `vector4.py` qu'on modifiera.

Après avoir remplacé `vector3.py` par `vector4.py`, tester à nouveau `TestGramSchmidt1.py` et comparer les temps d'exécution par rapport aux versions précédentes.

Quelles sont vos conclusions ?

# 4 Utilisation de numpy

## 4.1 Utilisation du module numpy

La première étape est de remplacer la manipulation de listes python par la manipulation de tableaux numpy.

Voici ici une brève description des fonctionnalités numpy qu'on utilisera pour cette partie :

- Création d'un tableau à partir d'une liste ou d'un tableau :

  ```
  a = numpy.array(l,type)
  ```

  où `l` est une liste ou un tableau ( numpy ), et type donne le type d'objets contenus par le tableau ( numpy.int, numpy.float, numpy.double, numpy.cfloat, numpy.cdouble )
  Exemples :

  ```
  A = numpy.array([[2.,-1.,0.,0.],[-1.,2.,-1.,0.],
                   [0.,-1.,2.,-1.],[0.,0.,-1.,2.]],numpy.double)
  u = numpy.array([1.,3.,5.,11.],numpy.double)
  ```

- Création d'un tableau à partir d'une liste ou d'un tableau sans recopie dans le cas du tableau :

  ```
  a = numpy.ascontiguousarray(l,type)
  ```

– Création d'un tableau en ne donnant que ses dimensions :

```
a = numpy.empty((d,type)
```

où `d` est un *tuple* contenant le nombre de dimensions
Exemples :

```
# Creation d'une 'matrice' 10x20 de type complexe double precision
A = numpy.empty( (10,20), N.cdouble)
# Creation d'un vecteur de dimension 3 de type reel double precision :
u = numpy.empty(3, N.double)
```

– Création d'un tableau en initialisant tous ses cœfficients à 0 :

```
u = numpy.zeros(d, type)
```

– Création d'un tableau en initialisant tous ses cœfficients à 1 :

```
u = numpy.ones(d, type)
```

– Les opérations algébriques pour les vecteurs se font avec les signes usuels Exemples :

```
u = numpy.array([1.,3.,5.],numpy.double)
v = numpy.array([2.,4.,6.],numpy.double)
w = u + v # w contient la somme de u et v
x = 2.*u # x contient 2u
```

Dans cet exemple, si u et v n'ont pas les mêmes dimensions, numpy renvoie une exception de type `ValueError`

– Produit scalaire de deux vecteurs :

```
prodScal = numpy.dot(u,v)
```

où u et v sont des tableaux numpy de même taille.

Transformer le fichier `vecteur.py` afin d'utiliser les tableaux et les fonctionnalités numpy.

Comparer le temps pris par `TestGramSchmidt1.py` par rapport aux versions précédentes...

## 4.2 Utilisation de f2py

On va utiliser les sources fortran données par `vecteurop.f90` pour en faire un module python (il serait possible également d'obtenir un module python à partir d'un fichier C, mais la manipulation en est moins aisée).

Pour cela, on va utiliser `f2py` qui est un programme fourni avec numpy. La syntaxe pour créer le module python est le suivant :

f2py -c -m <nom module python> <source fortran>

– Générer le module python associé à `vecteurop.f90`

– Créer une nouvelle version de la classe `vector` utilisant les fonctions définies dans le module créé à partir de `vecteurop.f90` (faire un help et un dir sur le module). Bien penser que les paramêtres de ce module doivent être des tableaux numpy contenant des réels double précision !
– Modifier `TestGramSchmidt1.py` en conséquence et comparez les temps de calcul par rapport aux versions précédentes...