École Nationale Supérieure d'Électrotechnique, d'Électronique,
d'Informatique,
d'Hydraulique et des Télécommunications

H2Matrix

Stage 2A 2022

Table des matières

1 Introduction

L'objectif de ce stage est d'étudier et de comparer les performances de 3 librairies permettant de calculer des matrices H2. Ces 3 librairies sont :

- H2tools
- H2Pack
- H2lib

Avant de découvrir les librairies, il est intéressant de se documenter sur les matrices hériarchiques et les matrices H2. Pour cela 2 documents sont mis à disposition :

- Thèse de Priscillia Daquin
- Notes de l'école de Leipzig

2 Installation de la librairie H2tools

La documentation de la librairie h2tools est accessible via le lien suivant : Documentation.
Quant au code source disponible sur bitbucket est dans le lien suivant : Code Source

Avant d'installer le code disponible sur bitbucket, il faut tout d'abord trouver l'environnement de travail le plus adapté, j'ai alors testé atom, idle python et Pycharm. Après plusieurs tests, j'ai remarqué que la librairie h2tools utilise du code fortran et des fichiers utilisable sur jupyter notebook. En effet, certains codes (en particulier pypropack) de la librairie utilise du code Fortran interfacé avec python.
Le plus adéquat est alors d'utiliser linux sous window (WSL), car je suis sous window. Pour cela il faut ouvrir une invite de commande en mode administrateur est rentrée la commande :

wsl –install

L'environnement que j'ai décidé d'utiliser sur WSL est Ubuntu. Maintenant que l'environnement est prêt, il faut alors installer les packages nécessaires pour utiliser H2tools.
ATTENTION car sans les bonnes versions de packages l'installation de la librairie risque d'être compromise. Après plusieurs essais avec différentes versions, les packages et les versions à installer dans l'ordre sont :

- **pip** (dernière version) : python3 -m pip install –upgrade pip
- **numba** (dernière version) : pip3 install numba
- **numpy** (1.19.2) : pip3 install numba==1.19.2
- **maxvolpy** (0.3.8) : pip3 install maxvolpy==0.3.8
- **scipy** (dernière version) : pip3 install scipy
- **pypropack** : il faut l'installer avec git clone –> git clone pypropack
  Après l'avoir installé, il faut se placer dans le fichier et installer le fichier setup.py :

  python3 setup.py install

  Il faut ensuite lié le chemin de pypropack avec l'invite de commande.
  Pour cela, il faut ouvrir le fichier bashrc dans /wsl.localhost/Ubuntu/home/user et écrire export PYTHONPATH=/home/seb/pypropack. Sinon on peut juste l'écrire dans l'invite de commande mais il faudra l'écrire à chaque fois qu'on reouvre ubuntu.

Normalement tout a été bien installer sans problème, l'installation de la librairie H2tools est maintenant possible : pip3 install h2tools

3 Utilisation de H2tools

Maintenant que H2tools est bien installé, en allant dans le fichier examples, on peut remarquer que les fichiers sont lisibles sous jupyter. Il faut alors installer jupyter notebook :

pip3 install jupyter

Après avoir installé le notebook, il suffit d'entrer la commande "jupyter notebook" pour le lancer.

3.1 Examples

Les 3 codes que j'ai principalement étudié pour comprendre le fonctionnement de la librairie sont particles, triangle_surface et minimal_working_example.

3.1.1 Particles

Le dossier dossier Particles comprends 2 codes : log_distance et inv_distance. Ces deux codes calculent la distance avec le log ou l'inverse entre une liste de points généré avec la fonction *randn*. Les codes vaut alors créer la matrice H2 correspondant à ces opérations.

Dans un premier temps, un objet data est crée avec la class data définie dans particles.py. Il est initialisé avec la dimension des points (ndim), le nombre de points (count) et les coordonnées des points (vertices).
La création de cet objet data est nécessaire pour l'initialisation des clusters tree. En effet pour créer l'objet ClusterTree, il faut l'objet data car il comporte les coordonnées du problème, de plus pour créer cet objet il faut initialiser la variable "Block_Size" correspondant à la taille maximale accepté pour les feuilles. Cet objet permet de hiérarchiser la liste de points initialisés comme une liste de parents et d'enfants et donc d'obtenir le nombre de feuilles, de noeuds et la profondeur des clusters.
Si le problème n'était pas symétrique, il faudrait créer 2 Cluster Tree différents, un correspondant aux lignes du problème et l'autre correspondant aux colonnes. Ici le Cluster Tree crée est le même pour les lignes et les colonnes.
Il faut maintenant crée les blocs des clusters tree à l'aide des cluster tree et de la fonction correspondant au problème. Le code appel func = particles.log_distance pour l'exemple sur le log, une fonction qui lorsqu'on lui donne une liste de d'indices correspondant à des points du problème, créer la sous matrice correspondante. Ainsi un objet problem de la class Problem est crée et il affiche les données suivantes :

```
    Cluster trees are generated in 0.12835192680358887 seconds\\
Depth level of each cluster tree: 11\\
Row cluster tree\\
    nodes : 739\\
    leaves: 370\\
Column cluster tree\\
    nodes : 739\\
    leaves: 370\\
```

Nous remarquons que les cluster tree des lignes et colonnes sont bien identiques.

Maintenant que tout est bien initialisé, la matrice H2 peut être crée :

matrix = mcbh(problem, tau, iters=iters, onfly=onfly, verbose=verbose)

La structure mcbh est une structure de matrice H2 dont les matrices d'interactions lointaines sont des sous matrices crée avec une erreur relative tau. Cette routine va appelé plusieurs fois l'objet problem et pourra donc créer les blocs de sous matrices utile à la construction de la matrice H2. iters correspond aux nombre d'itérations de la routine mcbh et onfly permet de choisir si l'on désire enregistrer en mémoire la matrice H2 ou juste l'appelé sur demande.
En éxécutant le code, nous obtenons les informations suivantes :

```
    Far-field interactions (MCBH method):\\
    Function calls: 7796\\
    Function values computed: 11122524\\
    Function time, seconds: 0.12\\
    Average time per function value, seconds: 1.10e-08\\
    Maxvol time, seconds: 0.4349534511566162\\
Near-field interactions:\\
    Function calls: 1551\\
    Function values computed: 404438\\
    Function time, seconds: 0.01\\
    Average time per function value, seconds: 2.23e-08\\
Total time, seconds: 0.62\\
Memory:\\
    Basises, MB: 0.14\\
    Transfer matrices, MB: 1.27\\
    Far-field interactions, MB: 5.55\\
    Near-field interactions, MB: 3.45\\
Total memory, MB: 10.41\\
```

La librairie comporte aussi une compression SVD pour le format H2 avec une tolérance à imposer :

matrix.svdcompress(1e-4, verbose=1)

La compression permet d'alléger le coût mémoire de la matrice.
Il est possible grâce à la fonction diffnorm d'obtenir l'erreur relative entre la matrice H2 et la matrice complète.

3.1.2 Approximation of BEM matrix

Le fichier vortex_ring_dynamics utilise à peu près de la manière la librairie H2tools. Cet exemple calcule la matrice H2 correspondant à un problème d'intégrale volumique :

$$A_{ij} = \sigma_i \int_{\Omega_j} |\mathbf{r} - \mathbf{r}_i|^{-3} ds,$$

La différence majeur avec l'exemple précèdent est qu'il n'initialise pas la data mais il importe un fichier .dat avec le format voulu. Le fichier semble avoir les coordonnées des sommets des triangles correspondant à la surface volumique. Il serait intéressait peut être de vraiment comprendre le format du fichier .dat pour essayer d'en construire pour nos propres exemples.

3.1.3 Minimal_working_example

Ce fichier reprends l'exemple de inv_distance mais redéfinie directement la class data et la fonction générant les sous matrices avec un interfaçage avec cython.
Ce que je ne comprends pas dans cet exemple est qu'il génère la matrice entière du problème "dense_matrix", l'utilisation des matrices H2 est alors inutile si nous générons la matrice dense.

3.2 Création de matrice H2

Maintenant que l'utilisation de la librairie est en majorité comprise, il est intéressant de créer mes propres exemples.

3.2.1 Intégrales Volumiques

Le code expliqué ci dessous est disponible en copie.

En récupérant l'example BEM, j'ai essayé de réutiliser le même problème mais en définissant une surface volumique différente. Dans un premier temps j'ai alors décidé de définir la surface d'une sphère.

La première difficulté fut la création de l'objet data, car en effet dans l'exemple fournit BEM, la data n'est pas initialisé mais elle est crée à l'aide d'un fichier qui est importé . Ainsi il fallait que je crée la data, pour cela le fichier triangular_surafce.py comporte la classe "TriangularSurface" qui permet d'initialiser un objet adapté pour un problème d'intégrale volumique. Pour crée l'objet data j'avais besoin des triangles qui définissent la surface et des coordonnées des sommets des triangles. Pour crée les triangles, j'ai alors utilisé la fonction Delaunay de *scipy.spatial* qui lorsque je lui donne des points va me crée des triangles et me donner toutes les informations nécessaires (indices, sommets, ...). Ainsi j'ai pu crée l'objet data :

data=trisurf.TriangularSurface(coordonne,triangle)

En gardant la surface correspondant à une sphère, je remarque à cette étape un problème de division par zéros. Après plusieurs tests de surface, je remarque que le problème provient d'un alignement des points de collocation avec les sommets des triangles. En prenant alors une demi sphère au lieu d'une sphère complète je remarque que le code s'éxécute sans problème. J'en déduit alors qu'il y avait un problème d'alignement avec les points aux extrémités hautes et basses de la sphère et le points de collocation crée par ces 2 points.

Maintenant nous pouvons créer la matrice H2 en réutilisant le même code que dans l'exemple de base.

3.2.2 Logarithme + interfaçage python/fortran

Maintenant on essaye sur un autre exemple le log de la différence entre des points d'un maillage généré par un code Fortran donné par M Poirier. Le code final est disponible sur logr.py.

Le dossier HMATDEV est disponible sur GitLab, je l'ai alors installé et placé dans home. Afin d'utiliser les fonctions disponible dans le dossier, il faut d'abord compiler le makefile dans l'invite de commande. Maintenant que tout est compilé, nous pouvons maintenant récupérer la fonction d'assemblage d'un coefficient et la fonction de création du cloud python.

J'ai eu des difficultés à utiliser la fonction générant le log (problème d'éxécution), j'ai alors décidé de créer ma propre fonction générant le logarithme en reprenant le même format que dans particles.py.

Intéressons nous maintenant à la génération de la matrice H2. Premièrement j'ai du définir la class data comme dans le code "minimal_working_example", mais j'ai supprimé dans l'initialisation le fait d'entrée la matrice entière du problème. Car en effet, en faisant cela on perd l'intérêt de la matrice H2 car on utilise de la mémoire inutilement.

```
class Data(object):
    """This is container for data for the problem."""
    """First of all, it requires methods check_far, compute_aux, divide and __len__.
    """

    def __init__(self, particles):
        """save particles and matrix"""
        self.particles = particles
        #self.matrix = matrix      # je n'ai pas compris l'utilite de cette ligne car la
    data ncessite seulement les coordonnes

        # main requirement here is to set self.count -- number of particles and self.dim
     -- dimensionality of the problem
        self.count, self.dim = particles.shape #attention la facon dont tu construis
     la data car ces 2 infos peuvent tre inverss

    """All other functions must have exactly the same parameters, as required."""

    def check_far(self, bb0, bb1):
        """checks if bounding boxes bb0 and bb1 do not cross each other."""
        mean0 = bb0.mean(axis=1)
        mean1 = bb1.mean(axis=1)
        dist = np.linalg.norm(mean1 - mean0)
        diag = max(np.linalg.norm(bb0[:, 1] - bb0[:, 0]), np.linalg.norm(bb1[:, 1] - bb1
    [:, 0]))
        return dist > diag

    def compute_aux(self, index):
        """computes bounding boxes, requires self.particles defined."""
        selected = self.particles[index]
        return np.hstack([selected.min(axis=0).reshape(2, 1), selected.max(axis=0).
    reshape(2, 1)])

    def divide(self, index):
        """divides cluster into subclusters."""
        vertex = self.particles[index].copy()
        center = vertex.mean(axis=0)
        vertex -= center.reshape(1, self.dim)
        normal = np.linalg.svd(vertex, full_matrices=0)[2][0]
        scal_dot = normal.dot(vertex.T)
```

```
        permute = scal_dot.argsort()
        scal_dot = scal_dot[permute]
        k = scal_dot.searchsorted(0)
        return permute, [0, k, permute.size]

    def __len__(self):
        return self.count
```

J'ai récupéré les mêmes méthodes car elles sont utilisés pour la génération de la data et des cluster tree.

Avec la classe data j'ai pu alors crée l'objet data avec les points du maillage généré par le code provenant de HMATDEV.

```
# Cration data
m,n=ex.bem2d2py.pretr()

# definition de la fonction d'assemblage d'un coefficient (voir exemple.f90)
fun = lambda i,j : ex.bem2d2py.calczij(i,j,m,n) # fonction qui fait log(r) avec r =|x-y|

# Creation du cloud python ie tableau de taille nxd (ici d=2)
pcl=ex.bem2d2py.pcloud(m,n,2)

pcl[n-1,0]=1 #le premier et le dernier point n' tait pas bien dfinie
pcl[n-1,1]=0
#print(pcl)

data=Data(pcl)
```

J'ai ensuite généré le cluster tree car je prends un problème symétrique et donc il les lignes et les colonnes sont identiques. J'ai limité à 25 le nombre maximal de points dans un noeud, cet argument est important car influe sur la stockage maximal de la matrice H2.

```
# Initialize cluster trees with root node

block_size = 25
tree = ClusterTree(data, block_size)
```

Maintenant il faut crée l'objet permettant d'assembler les blocs cluster tree "problem". Pour cela il me faut définir la fonction qui lorsque lui donnant des datas et des listes d'indices va crée le bloc correspondant :

```
def func(data1, list1, data2, list2):

    ans = np.ndarray((list1.size, list2.size), dtype=np.complex64)
    return log_numba(data1.dim, data1, list1, data2, list2, ans)

from numba import jit
```

```
import math

def log_numba(ndim, data1, list1, data2, list2, ans):
    n = list1.size
    m = list2.size
    for i in range(n):
        for j in range(m):
            tmp_l = 0.0
            for k in range(ndim):
                #breakpoint() #fonction qui permet de debugger et afficher les variables pendant la boucle
                tmp_v = data1.particles[list1[i],k]-data2.particles[list2[j],k]
                tmp_l += abs(tmp_v)
            if tmp_l <= 0:
                ans[i, j] = 0
            else:
                ans[i, j] = math.log(tmp_l)
    return ans
```

L'objet problem permettant de crée des blocs de sous matrices peut être alors crée. J'ai essayé d'utiliser la fonctionnalité jit afin de créer la matrice dynamiquement mais malheureusement j'ai rencontré des problèmes lorsque je l'ai utilisé.

```
#Generate block cluster tree in variable 'problem'

symmetric = 1
verbose = True
problem = Problem(func, tree, tree, symmetric, verbose)
```

Tous les objets étant bien initialisés, la matrice H2 peut alors être crée.

```
#print('Computing MCBH, relative error parameter tau set to 1e-4')
matrix = mcbh(problem, tau=1e-4, iters=1, onfly=0, verbose=1)
# tau = Spectral error tolerance for SVD decompressions of each block row and block column.
```

Pour 1000 points grâce à la fonction diffnorm, nous obtenons une erreur relative de 0.08909502 avec un tau de 10^{-4}.

Maintenant que la matrice H2 est crée il est intéressant de calculer le produit de la matrice avec un vecteur donné A*x et de comparé le résultat du même produit mais avec la matrice complète. La classe H2matrix dans h2matrix.py possède la fonction réalisant le matrice matrice vecteur à gauche et à droite. Ainsi en prenant un vecteur avec seulement des 1 et en créant la matrice complète, avec 1000 points j'obtiens l'erreur relative suivante :

erreur relative Matvec avec norm inf 0.115758464
erreur relative Matvec avec norm 2 1.2515833

L'erreur étant trop grande, il faut alors diminuer la tolérance de la décompression SVD. Pour un tau de 10^{-6}, je trouve :

erreur relative Matvec avec norm inf 0.009033699
erreur relative Matvec avec norm 2 0.057470914

Le résultat semble satisfaisant.

3.3 Méthode intégrale

La méthode intégrale est une méthode étudié en 3eme année dans le parcours physique numérique. Cette méthode est particulièrement utile pour les problèmes sans frontière, comparé aux éléments finis cette résolution comporte des matrices pleines et taille de plus faible. Il est alors intéressant d'utiliser les matrices H2 pour optimiser la résolution. Je me suis alors basé sur le TP qui souhaite résoudre le problème suivant avec les méthodes intégrales

$$-\int_{\Gamma} G(r,r')q(r)d\gamma(r) = u^i(r')\forall r' \in \Gamma$$

Définissons Γ comme la surface d'une cercle. Comme pour le code précèdent les objets sont définis de la même façon, ce qui va changer est la fonction qui crée les blocs de sous matrices. J'ai alors repris le code de la fonction S0 du TP et je le l'ai modifié pour qu'il sorte ce dont on a besoin.

```
def S0(data1, list1 , data2, list2 , ans):
    # Description des parametres physiques
    f=0.6
    phi=0
    kk=2*math.pi*f/0.3
    ndim = data1.dim
    ky=kk*math.cos(phi)
    kx=kk*math.sin(phi)
    n = list1.size
    m = list2.size
    xm=[]
    ym=[]
    lx=[]
    ly=[]
    for i in range(n):
        for v in range(m):
            #print('i  :', i)
            #for k in range(ndim):
            if  list1[i]==int(data1.particles.shape[0]-1) and list2[v]==int(data2.
    particles.shape[0]-1):

                xm=0.5*(data1.particles[list1[i],0]+data1.particles[0,0])
                ym=0.5*(data2.particles[list2[v],1]+data2.particles[0,1])
                lx=data1.particles[list1[i],0]-data1.particles[0,0]
                ly=data2.particles[list2[v],1]-data2.particles[0,1]

            elif  list1[i]==int(data1.particles.shape[0]-1):

                xm=0.5*(data1.particles[list1[i],0]+data1.particles[0,0])
                lx=data1.particles[list1[i],0]-data1.particles[0,0]
```

```
                ym=0.5*(data2.particles[list2[v],1]+data2.particles[int(list2[v]+1),1])
                ly=data2.particles[list2[v],1]-data2.particles[int(list2[v]+1),1]

            elif list2[v]==int(data2.particles.shape[0]-1):

                ym=0.5*(data2.particles[list2[v],1]+data2.particles[0,1])
                ly=data2.particles[list2[v],1]-data2.particles[0,1]
                xm=0.5*(data1.particles[list1[i],0]+data1.particles[int(list1[i]+1),0])
                lx=data1.particles[list1[i],0]-data1.particles[int(list1[i]+1),0]

            else :

                xm=0.5*(data1.particles[list1[i],0]+data1.particles[int(list1[i]+1),0])
                ym=0.5*(data2.particles[list2[v],1]+data2.particles[int(list2[v]+1),1])
                lx=data1.particles[list1[i],0]-data1.particles[int(list1[i]+1),0]
                ly=data2.particles[list2[v],1]-data2.particles[int(list2[v]+1),1]

            D=kk*cmath.sqrt(xm**2+ym**2) # Distance des milieux
            l=cmath.sqrt(lx**2+ly**2) # longueur des segments.
            e=cmath.exp(1)
            if i!= v :
                ans[i,v]=-1j*sp.hankel1(0,D)/4*l*l;

            else :
                    #from IPython.core.debugger import Pdb; Pdb().set_trace()
                    #print(data1.particles[37,k])
                    #print(data1.particles[38,k])
                    #print('lx', lx)
                    #print('ly', ly)

                    #print('k',k)
                aii=-1j/4*(1+2*1j/math.pi*cmath.log((2*math.pi*f/0.3)*e**0.5772/(4*e)
*l))*l**2
                ans[i,i]=aii
    return ans
```

A l'aide de la classe gmres forunit par M Poirier et en posant le second membre par $e^{j(k_x+k_y)}$, nous pouvons résoudre le problème. Ainsi afin de vérifier si le code marche, la marche complète est aussi crée afin de comparer la solution avec la matrice complète et avec la matrice H2. En calculant l'erreur relative du vecteur solution des 2 méthodes est bien trop grande. Le code est donc à reprendre afin de comprendre l'origine de cette grande erreur.

4 Installation de la librairie H2Pack

La documentation et le code source de la librairie H2Pack est accessible via le lien suivant : H2Pack. Comparé à la librairie H2tools, H2Pack est mis à jour plus régulièrement et donc plus facile d'installation car mieux expliqué. La majorité de cette librairie est codé en C mais elle possède un interfaçage avec Python.

Avant d'installer la librairie, il faut installer OpenBlas :

- cd OpenBLAS
- make
- make USE_OPENMP=1
- make PREFIX=/home/local/ install

En faisant ces commandes, un dossier local va se créer dans home, ce dossier comporte OpenBLAS compilé et installé.

Maintenant que OpenBLAS est bien installé, il faut installer H2Pack en faisant un git clone :

- git clone –recurse-submodules https ://github.com/scalable-matrix/H2Pack.git
- cd H2Pack/src

Avant de compiler les makefile, il faut ouvrir les makefile dans les fichiers example, src et pyh2pack et rajouter CC= gcc et rajouter la direction de OPENBLAS dans les lignes correspondantes :

OPENBLAS_INSTALL_DIR = /home/local/

Attention à la version de gcc, pour ma part la version qui fonctionne est gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0.

Les makefile peuvent maintenant être compilé, il faut d'abord se placer dans le fichier src et compiler le makefile : make -f GCC-OpenBLAS.make
Compiler de la même façon le makefile dans le dossier examples.

5 Utilisation de H2PAck

5.0.1 Example_H2.py

Dans un premier temps, j'ai étudié le code example_H2.py afin de comprendre comment fonctionne la librairie. En éxécutant l'exemple nous obtenons :

```
Generating random coordinates in a scaled cubic box... done.
coordonneH2Pack generate numerical proxy points used 0.006 (s)...
Calculating direct n-body reference result for points 0 -> 39999
x : Direct n-body for 40000 points takes 0.914 (s)
Full H2 matvec takes 0.417 (s)
========== H2Pack H2 tree info ==========
  * Number of points                 : 40000
  * Kernel matrix size               : 40000
  * Maximum points in a leaf node    : 400
  * Maximum leaf node box size       : 0.000000e+00
  * Number of levels (root at 0)     : 4
  * Number of nodes                  : 585
  * Number of nodes on each level    : 1, 8, 64, 512
  * Number of nodes on each height   : 512, 64, 8, 1
  * H2Pack running mode              : H2
  * Minimum admissible pair level    : 2
  * Number of reduced adm. pairs     : 28224
  * Number of reduced inadm. pairs   : 5068
========== H2Pack storage info ==========
  * Just-In-Time B & D build         : Yes (B & D not allocated)
  * H2 representation U, B, D        : 64.75, 1355.55, 261.85 (MB)
  * Matvec auxiliary arrays          : 10.05 (MB)
  * Max / Avg compressed rank        : 152, 82
========== H2Pack timing info ==========
  * H2 construction time (sec)    = 0.352
      |----> Point partition      = 0.033
      |----> U construction       = 0.319
      |----> B construction       = 0.001
      |----> D construction       = 0.000
  * H2 matvec average time (sec) = 0.417, 4.29 GB/s
      |----> Forward transformation      = 0.110, 0.58 GB/s
      |----> Intermediate multiplication = 0.095, 26.32 GFLOPS
      |----> Backward transformation     = 0.065, 0.98 GB/s
      |----> Dense multiplication        = 0.040, 11.91 GFLOPS
      |----> OpenMP vector operations    = 0.106, 0.12 GB/s
==========================================================================

For 40000 validation points: ||y_{H2} - y||_2 / ||y||_2 = 1.766021e-07
The specified relative error threshold is 1.000000e-06
```

```
Store H2 matrix data to file? 1--yes, 0--no : 0
```

L'exemple crée une liste de 40000 points en 3D et la rempli avec des points répartis uniformément dans l'intervalle [0.0,1.0]. La création de la matrice H2 est différente de la librairie h2tools car en effet ici le choix du problème à résoudre se fait par le kernel. Dans cet exemple le kernel utilisé est "Coulomb_3D", de plus dans la librairie plusieurs kernel sont définies :

- 2D Laplace : $K(x, y) = log(r)$
- 3D Coulomb : $K(x, y) = \frac{1}{r}$
- 2D - 3D Gaussian : $K(x, y) = e^{-lr^2}$
- 2D - 3D Exponential : $K(x, y) = e^{-lr}$
- 2D - 3D Matern 3/2 : $K(x, y) = (1 + \sqrt{3}rl)e^{-\sqrt{3}lr}$
- 2D - 3D Matern 5/2 : $K(x, y) = (1 + \sqrt{5}rl + \frac{5}{3}l^2r^2)e^{-\sqrt{5}lr}$
- 2D - 3D Quadratic : $K(x, y) = (1 + cr^2)^a$
- 3D Stokes : $K(x, y) = \frac{1}{r}I + \frac{(x-y)(x-y)^T}{r^3}$

Avec $r = |x - y|$ et l,c,a des paramètres scalaires.

Il est possible de créer notre propre noyau dans H2Pack, il faut écrire une fonction d'évaluation de la matrice du noyau (KME) qui prend deux ensembles de points X0, Y0 et génère le bloc de noyau K(X0, Y0). Une fonction KME prend deux ensembles de coordonnées de points et un tableau de paramètres en entrée et retourne la KME. Par manque de temps, je n'ai pas pu créer mon propre kernel mais il serait intéressant de créer un kernel qui n'est pas définie dans la librairie. Les kernel sont définies dans les fichier H2Pack_2D_kernels.h et H2Pack_3D_kernels.h dans le dossier src.

Pour revenir à l'affichage de l'exécution, les informations récupérés sont a peu près les mêmes que celle dans h2tools. Le seul point flou est la construction U,B,D que je ne comprends pas et que je n'ai pas trouvé expliqué dans la librairie.

5.0.2 Interfaçage avec Python

La librairie comporte un interfaçage avec Python qui facilite son utilisation, ainsi pour faciliter l'étude de la librairie j'ai décidé de travailler sur python. Le code example.py dans le dossier est le suivant :

```
1  import pyh2pack
2  import numpy as np
3
4  # generate random points in a 3D box
   N = 80000
6  coord = np.random.uniform(0, 5, size=(3, N))
7
8  # dimension and parameters of the kernel function
   krnl_dim = 1
10 krnl_param = np.array([1., -2.]) # please use double format but not integers
11
12 # build the H2 matrix
```

```
A = pyh2pack.H2Mat(kernel="Quadratic_3D", krnl_dim=krnl_dim, pt_coord=coord,
    pt_dim=3, JIT_mode=1, rel_tol=1e-3, krnl_param=krnl_param)

# H2 matvec
x = np.random.normal(size=(krnl_dim*N))
y = A.h2matvec(x)

# direct matvec for a subset of rows
start_pt = 8000
end_pt = 10000
z = A.direct_matvec(x, start_pt, end_pt)

# error of H2 matvec
print(np.linalg.norm(y[(start_pt-1)*krnl_dim:end_pt*krnl_dim] - z) / np.linalg.
    norm(z))

# print out statistic info of this h2 matrix representation
A.print_statistic()

#. destroy the h2 data structure (optional)
A.clean()
```

Les paramètres de la création de la matrice H2 sont :

- kernel : chaîne de caractère avec le nom du noyau
- kernel_dim : entier correspondant à la dimension du tableau sortant de la fonction kernel
- pt_coord : tableau correspondant aux coordonnées des points
- pt_dim : entier correspondant à la dimension des points
- JIT_mode (optionnel) : entier pour choisir d'exécuter un produit matrice vecteur en JIT mode, le mode JIT permet d'éxécuter dynamiquement le produit et donc utilise moins de stockage pour temps plus long
- rel_tol : reél correspondant à la tolérance de la matrice H2
- krnl_param (optionnel) : liste avec les paramètres de la fonction kernel
- max_leaf_points (optionnel) : entier correspondant au nombre maximal dans chaque noeuds

Puisque le nombre de points et très grand (N=80000), afin de comparer le produit matvec avec la matrice H2 et la matrice complète pour un vecteur donné. L'exemple calcul le produit matvec avec la "matrice complète" et une partie des points du vecteur avec la fonction direct_matvec. Et il calcul le produit matvec de la matrice H2 avec la fonction matvec.
Ainsi grâce à ces fonctions, on peut obtenir l'erreur relative des produits matvec. Sinon afin de comparer entièrement le produit, il est possible de simplement baisser le nombre de points et de calculer la matrice complète.

5.1 Exemple Logarithme

Afin de comparer les performances entre h2tools et H2PAck nous utilisons le même exemple du logarithme. L'avantage de H2Pack est que le kernel du logarithme est déjà définie, la création de la matrice H2 est alors plus simple.

```

krnl_dim = 1
pt_dim = 2
A = pyh2pack.H2Mat(kernel="Laplace_2D", krnl_dim=krnl_dim, pt_coord=pcl.T,
    pt_dim=pt_dim, JIT_mode=0, rel_tol=1e-4)
```

Le Kernel "Laplace_2D" correspond à la fonction log(r).

6 Comparaison des performances entre H2tools et H2Pack sur un exemple

Il est intéressant maintenant que nous avons le même exemple pour les 2 librairies différentes de comparer les performances et l'erreur relative après un produit matrice vecteur avec un vecteur donné.

Par exemple, pour 10000 points, avec une tolérance de compression de 10^{-6} de la SVD pour H2tools et un seuil de tolérance de 10^{-6} , j'obtiens les erreurs relatives suivantes :

Norm inf h2tools : 0,017260287
Norm inf H2Pack : 9,00E-07

En exécutant le programme et en modifiant plusieurs variables j'obtiens les données de ce sheets. Nous pouvons remarquer que la librairie H2Pack fait preuve de meilleures performances en terme de stockage, de temps de compilation et aussi à une erreur relative plus faible de la librairie H2Tools.

7 Conclusion

Par manque de temps et quelques pertes de temps sur l'installation des librairies, je n'ai pu que étudier 2 librairies sur les 3. En conclusion, je dirai qu'il est plus intéressant de continuer de travailler sur la librairie H2Pack car ses performances sont clairement meilleures et de plus elle est régulièrement mise à jour ce qui est très avantageux. Aussi L'interfaçage du code en C sur python est claire et simplifié ce qui rend son utilisation plus agréable.
Il serait intéressant dans la suite de prendre des exemples plus complexe et de comparer toutes les librairies.