Rapport Projet Informatique

Localisation d'Orbitales - FL05

Table des matières

1 Introduction

L'étude des orbitales moléculaires constitue un outil essentiel en chimie quantique pour comprendre la structure électronique des molécules. Toutefois, les orbitales issues de la résolution de l'équation de Schrödinger sont généralement délocalisées, et cela se voit sur les représentations. Ceci les rend peu intéressante à interpréter chimiquement, notamment car elle ne représente pas les liaisons ou les doublets non liants. Pour pallier à cette difficulté, plusieurs méthodes de localisation des orbitales ont été développées. Parmi elles, l'approche POPLOC (*population localised orbitals*) utilisée dans l'article est celle que nous allons suivre, apparemment la plus efficace pour notre objectif. Elle permet de transformer les orbitales délocalisées en orbitales localisées (LMO), directement interprétables en fonction des liaisons chimiques, et ce sans recourir à des calculs lourds d'intégrales biélectroniques.

Dans ce travail, nous nous intéressons à l'implémentation de cette méthode à travers un programme écrit en C. Ce programme a pour objectif de résoudre le problème posé par l'algorithme POPLOC : à partir de la matrice des coefficients d'orbitales moléculaires (E), identifier les combinaisons linéaires menant aux orbitales localisées. L'intérêt de ce développement est double. D'une part, il offre une meilleure compréhension des résultats quantiques en permettant une interprétation chimique visuelle à l'aide de logiciels tel que JMOL à partir de la nouvelle matrice des coefficients. D'autre part, il permet une automatisation, une reproductibilité et une rapidité d'exécution adaptées à l'analyse de systèmes moléculaires de complexité variable.

Ceci dit, dans notre projet nous nous sommes penchés sur le cas de CH_4, une petite molécule sans doublets non liants afin de simplifier le problème. Nous avons donc uniquement quatre liaisons C-H à considérer dans notre programme.

Cependant avant de se pencher sur le programme et les résultats de celui-ci, il est important de rappeler les hypothèses utilisées. Tout d'abord, l'hypothèse de la combinaison linéaire des orbitales atomiques ($LCAO$). Cette hypothèse suppose que les OA forment une base complète pour décrire l'état électronique de la molécule, ce qui permet d'écrire chaque OM (ψ_{OM}) comme une combinaison linéaire d'OA (ϕ_i). Elle rend le problème calculable via le principe variationnel : les coefficients (c_i) se déterminent en minimisant l'énergie sous contrainte de normalisation.

$$\psi_{\text{OM}}(\mathbf{r}) = \sum_{i=1}^{N} c_i \, \phi_i(\mathbf{r})$$

De plus, on utilise le fait que les OM occupées peuvent être localisées afin qu'elles correspondent à des liaisons ou des doublets, plus proches de la représentation chimique intuitive. En effet, les orbitales moléculaires **occupées** peuvent subir une transformation unitaire sans altérer la densité électronique ni l'énergie totale du système (s'explique car l'état électronique d'une molécule est représenté par un **déterminant** de Slater formé à partir des orbitales **occupées**), car elles **décrivent le même espace vectoriel**. Ce qui n'est pas le cas des vacantes car les mélanger avec des OM occupées modifierait l'espace vectoriel et donc la densité et l'énergie du système.

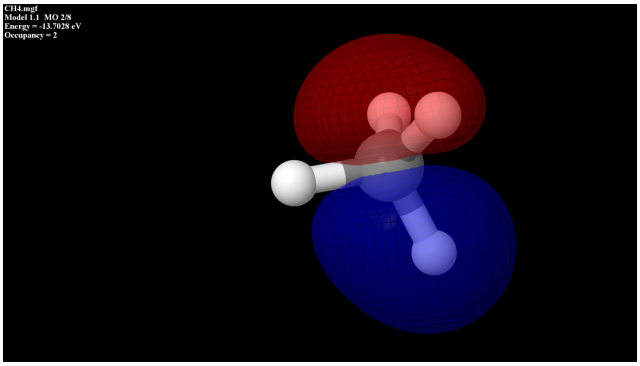


FIGURE 1 – Exemple de représentation d'un OM délocalisée de la liaison

2 Méthode d'approche de l'algorithme

Notre programme a pour objectif de localiser les orbitales moléculaires d'une molécule à partir d'un fichier `.mgf` dont les orbitales sont encore délocalisées.

Pour cela, le programme doit remplir deux tâches distinctes :

- Être capable de lire et écrire un fichier `.mgf`
- Réaliser les opérations matricielle nécessaire à la résolution du problème.

Pour cette raison, le projet a été décomposé en 3 partie : `matrice.c` contenant l'ensemble des méthodes permettant les calculs matriciels nécessaire aux mathématiques du problème, `gestionfichier.c` contenant les méthodes pour lire et écrire un fichier `.mgf` et enfin un fichier `main.c` permettant de réaliser la localisation des orbitales grâce aux deux dépendances. De plus, le logiciel a besoin de deux fichiers de données : `molecule.mgf` (ici `CH4.mgf` et `liaison.csv` matrice de liaison renseignant les liaisons entre les différents atomes de la molécule. Enfin, le programme devra retourner un fichier `.mgf` contenant les orbitales délocalisées. L'arborescence du projet est donc la suivante :

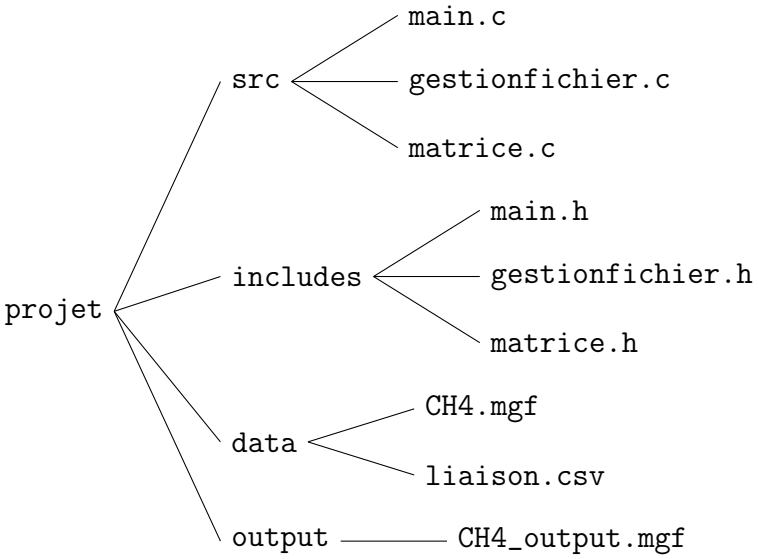


FIGURE 2 – Arborescence du projet C : structure des fichiers et répertoires.

2.1 Calcul Matriciel

Notre programme utilise un *type* personnalisée de variable appelée `Matrice` pour réaliser les calculs de la structure suivante :

Extraits 1 – `includes/matrice.h`

```c
//Définition d'une Matrice
typedef struct {
    int n;
    int m;
    long double** data;
} Matrice;
```

Ainsi, pour une matrice `Matrice M` donnée, il est possible de connaître en toute circonstance ses dimensions, via `M.n` et `M.m`. Pour lire ou écrire son contenu on utilise `M.data[i][j]` où `i` représente la ligne i et `j` la colonne j. Cette structure nous permet de simplifier grandement les entrées et sorties des fonctions que nous utiliserons, la structure `Matrice` renseignent d'elle-même la taille de la matrice qu'elle représente simplifiant l'implémentation des calculs mathématiques. Elle permet aussi un déboggage simplifié, puisqu'il est possible de connaître à chaque étape du programme la taille des matrices utilisées.
Le fichier `matrice.h` renseigne la totalité des méthodes permettant de réaliser les opérations matricielles dont le programme a besoin mais également les méthodes pratiques d'initialisation, d'affichages et de libération d'une variable `Matrice`. Cela est décrit dans l'Extrait 2.

Extraits 2 – `includes/matrice.h`

```c
//Fonctions pratiques
Matrice createMatrice(int n, int m); //Renvoie la matrice initialisée de taille n*m
void freeMatrice(Matrice *M); //Libère de la mémoire la matrice
Matrice copieMatrice(Matrice M); //Renvoie une copie dure de la Matrice M
void afficheMatrice(Matrice mat); //Affiche la matrrice dans la console
void testMatrice(Matrice M); // Transforme la matrice M en une matrice de test C(i, j) = i+j
Matrice Identite(int n); //Renvoie une Matrice identité de taille n

//Fonctions de calculs matricielles
void pScaMat(Matrice M, long double a); // a*M (sur place)
```

```
long double pScalaire(Matrice X, Matrice Y); //Produit scalaire X·Y
Matrice addMat(Matrice M, Matrice N, long double a); // M + aN
long double norme(Matrice M); //Norme d'un vecteur M
Matrice transpose(Matrice M); //Transposée de M
Matrice produit(Matrice M, Matrice N); //Produit matriciel M*N
Matrice vpDiag(Matrice D); //Vecteur des coefficients diagonaux de D
long double minVP(Matrice colVP, int *pos_out); //Valeur min et position
long double maxVP(Matrice colVP, int *pos_out); //Valeur max et position
Matrice getCol(Matrice M, int pos); //Colonne n-ième de M
int areColinear(Matrice u, Matrice v); //Colinéarité de u et v
void vpJacobi(Matrice a, Matrice b); //Jacobi: a diagonalisée, b=VP
void compareVect(Matrice u, Matrice v); // u·v / (||u||·||v||)
```

2.2 Gestion des fichiers

Le programme doit être capable d'ouvrir un fichier `.mgf` de le parcourir et d'en tirer les variables nécessaires au calcul. Le fichier est constitué d'une entête suivi de plusieurs ensemble de ligne correspondant à des informations sur les atomes puis les orbitales atomiques. Aucune entête ne permet de retrouver facilement chacune des sections, il est donc nécessaire de naviguer méthodiquement dans le document, en prenant en compte les informations précédemment acquise (nombre d'orbitales atomiques, nombre d'atomes) afin de sauter un nombre correct de ligne. Pour cela la méthode commence d'abord par obtenir le nombre d'atomes composant la molécule. Cela lui permet ensuite d'itérer le nombre de boucle nécessaire pour parvenir à lire l'élément chimique de chacun d'entre eux. En supposant que la molécule est exclusivement constitué de C,O,N,ou H il calcule le nombre d'orbitales atomiques dans la molécule. Cela lui permet de construire la matrice E, représentation des contributions des orbitales atomiques dans les orbitales moléculaires. Au final, à partir de ce fichier le programme est capable de constituer la `Matrice E`, ainsi que le nombre d'atomes et d'électrons présent dans la molécule.

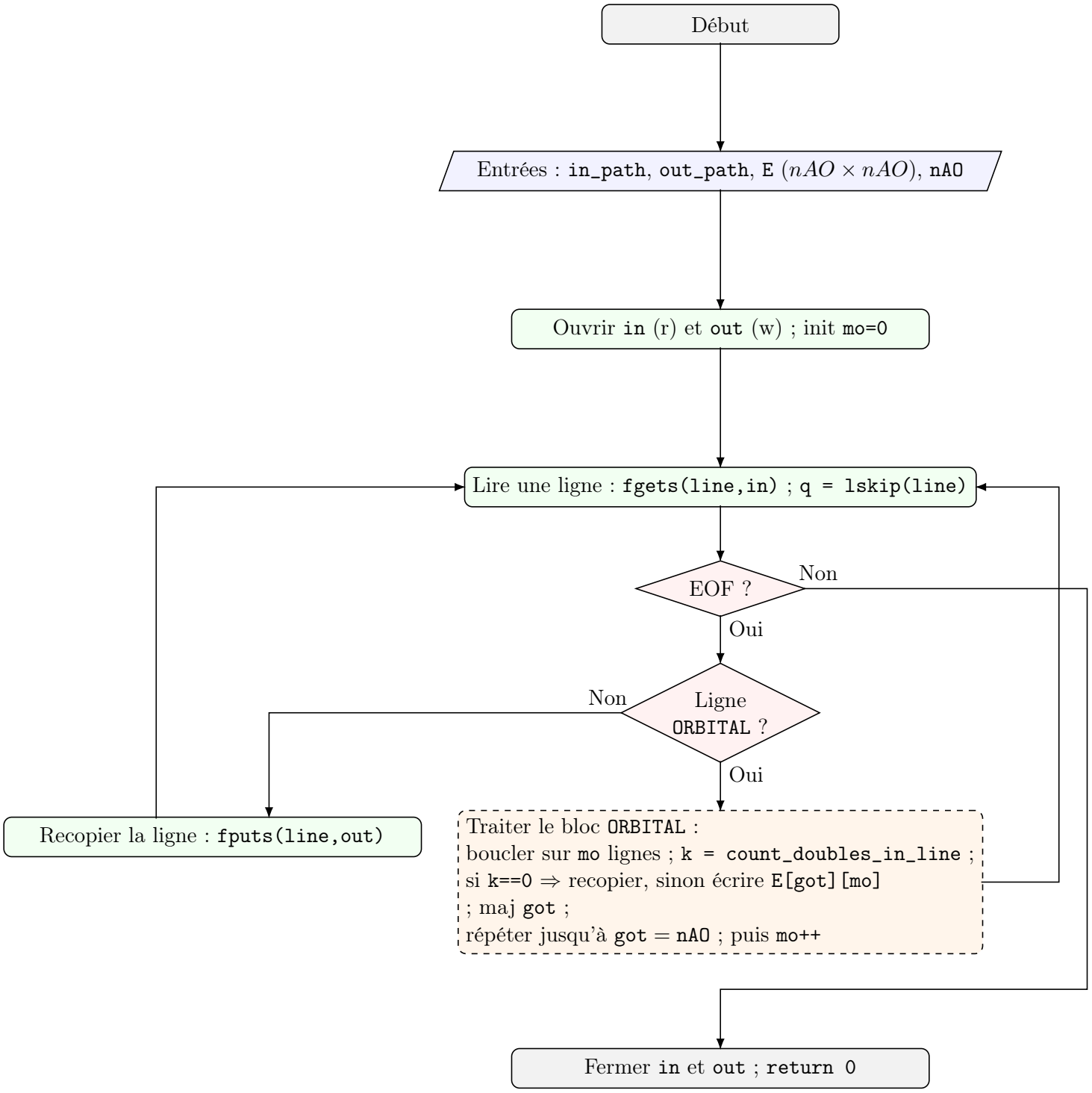


FIGURE 3 – Organigramme vertical de `ecrireMGF(...)` — réécriture des blocs `ORBITAL` depuis `E`.

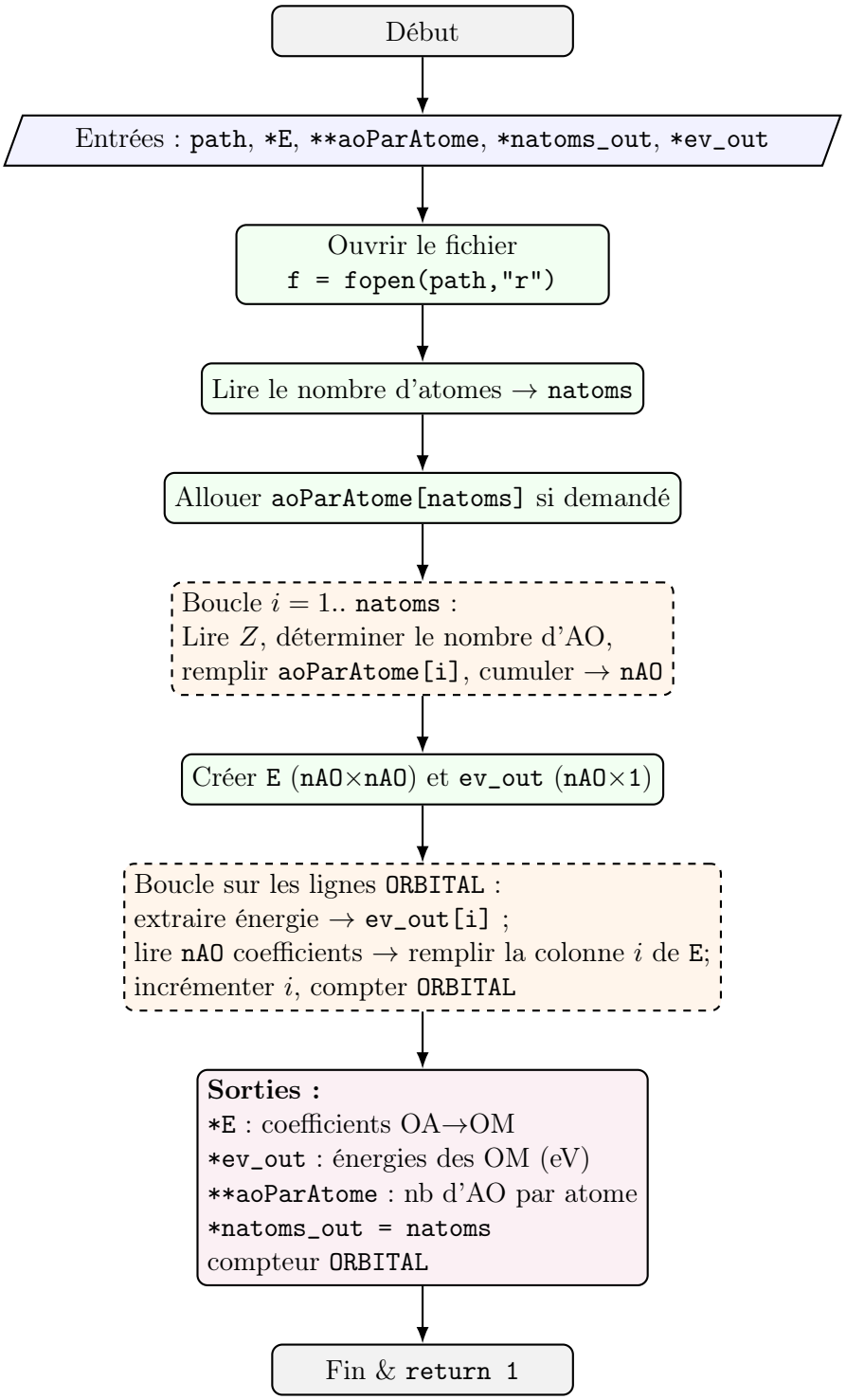


FIGURE 4 – Organigramme vertical de `lireMGF(...)`.

Afin de constituer la matrice R, le programme doit également être au fait de la répartition des liaisons dans la molécule. Pour cela nous avons crée un fichier CSV représentant la matrice du graphe des liaisons présentes dans la molécule. La fonction `lireCSV(...)` permet de récupérer cette matrice à partir du fichier CSV.
Enfin, le programme doit être capable de rédiger un fichier `.mgf` neuf (appelé ici `out.mgf`) contenant les orbitales moléculaires localisées calculées. Pour cela, il utilise comme base un fichier `.mgf` d'entrée(qu'on

appelera `in.mgf`)qui est celui utilisé précédemment pour acquérir les valeurs des orbitales moléculaires délocalisées. La méthode se contente de recopier identiquement ligne par ligne le fichier `in.mgf` a moins que la ligne lu commence par la chaine `"ORBITAL"`. Dans ce cas, le programme rédige une ligne correspondant aux nouveaux coefficients calculés. Enfin, la méthode termine lorqu'elle parviens à la fin du document.

3 Organigramme & Détail des fonctions

3.1 Organigramme

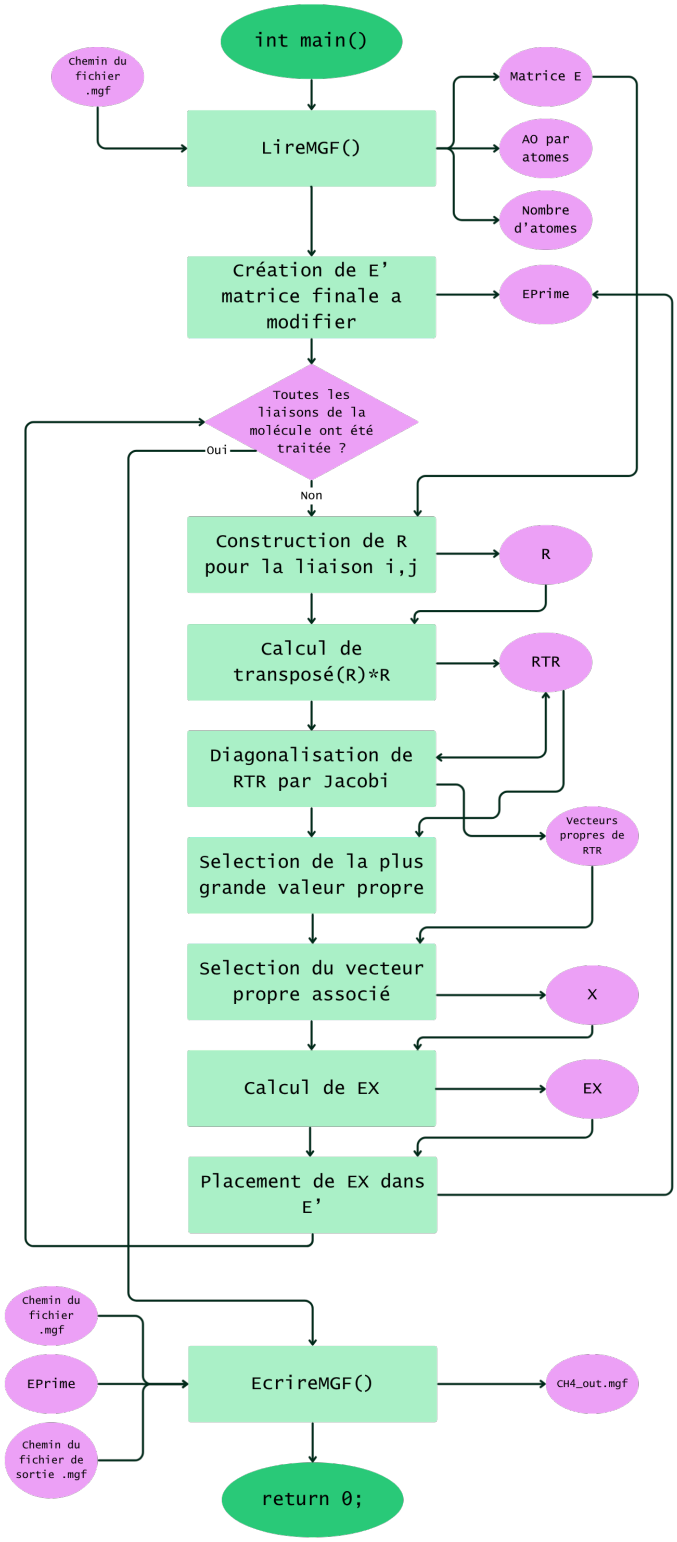


FIGURE 5 – Organigramme du Code

L'objectif du programme est, à partir d'une matrice $E_{[OA][OM]}$ récupérée à partir un fichier `.mgf` de constituer une matrice des orbitales localisées $E'_{[OA][OM]}$.
En prenant l'exemple du méthane, la matrice E aura l'allure suivante.

$$E[\text{OA}][\text{OM}] = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} & p_{16} & p_{17} & p_{18} \\ p_{21} & p_{22} & p_{23} & p_{24} & p_{25} & p_{26} & p_{27} & p_{28} \\ p_{31} & p_{32} & p_{33} & p_{34} & p_{35} & p_{36} & p_{37} & p_{38} \\ p_{41} & p_{42} & p_{43} & p_{44} & p_{45} & p_{46} & p_{47} & p_{48} \\ p_{51} & p_{52} & p_{53} & p_{54} & p_{55} & p_{56} & p_{57} & p_{58} \\ p_{61} & p_{62} & p_{63} & p_{64} & p_{65} & p_{66} & p_{67} & p_{68} \\ p_{71} & p_{72} & p_{73} & p_{74} & p_{75} & p_{76} & p_{77} & p_{78} \\ p_{81} & p_{82} & p_{83} & p_{84} & p_{85} & p_{86} & p_{87} & p_{88} \end{bmatrix}$$

À partir de celle-ci, on peut construire, pour chaque liaison C–H, un sous-ensemble d'orbitales locales en supprimant les lignes correspondant aux orbitales atomiques (OA) situées sur les deux atomes concernés. On obtient ainsi une matrice

$$R = P_{\mathcal{N}} E,$$

où $P_{\mathcal{N}}$ est le projecteur qui conserve uniquement les OA dites locales. Physiquement, cela revient à isoler la contribution des orbitales moléculaires occupées par des électrons : la matrice R mesure la population électronique des orbitales délocalisées dans la région où l'on cherche à localiser la densité électronique. Comme l'espace engendré par les orbitales occupées est invariant par rotation unitaire, il est légitime de chercher, dans cet espace, la combinaison linéaire X qui maximise cette population non locale, c'est-à-dire le vecteur propre associé à la plus grande valeur propre de la matrice symétrique positive

$$R^{\mathrm{T}} R.$$

La colonne correspondante

$$L = EX$$

représente alors l'orbitale la plus confinée sur la liaison considérée, et l'ensemble des vecteurs propres ainsi obtenus pour les quatre liaisons C–H définit la matrice transformée

$$E' = E\,\widetilde{X},$$

dont les colonnes constituent les orbitales localisées du méthane.

3.2 Fonctions principales

En ce qui concerne les fonctions centrales de notre programme, nous allons les détailler dans la suite :
Fonction `LireMGF` (cf Figure 4) :

- Prototype :
  ```
  int lireMGF(const char *path, Matrice *E, int **aoParAtome, int *natoms_out);
  ```
- Entrée : Le fichier, une matrice, un pointeur vers un tableau contenant le nombre d'Orbitales atomiques (OA) qu'un atome possède et un pointeur entier pour le nombre d'atomes de la molécule.
- Sortie : Renvoie une matrice E contenant les 8 orbitales moléculaires (OM) sur les colonnes en fonction des 8 OA sur les lignes. Renseigne dans les pointeurs idoines le nombre d'atomes et d'orbitales atomiques dans la molécule.

- Rôle : Cette fonction lit le fichier `CH4.mgf` et extrait les données des orbitales moléculaires (ici 8 mais fonctionne pour toutes les molécules de base [contenant O,N,C,H]) puis les renvoie sous forme de matrice (E).

Fonction `EcrireMGF` :

- Prototype :
```
int ecrireMGF(const char *in_path, const char *out_path,const Matrice *E, int nAO);
```
- Entrée : Le chemin vers le fichier MGF a modifier, le chemin vers le nouveau fichier MGF qui va sera crée, la matrice calculée contenant les nouveaux coefficients, le nombre d'OA de la molécule.
- Sortie : Modifie le fichier, renvoie 0 si le fichier a bien été modifié.
- Rôle : Cette fonction lit le fichier `CH4.mgf` et remplace les coefficients des OM du fichier par les nouveaux coefficients à l'aide de la nouvelle matrice E' (pour CH_4, les 4 nouvelles colonnes de L' et les 4 autres colonnes de E).

Fonction Jacobi :

- Prototype :
```
void vpJacobi(Matrice a, Matrice b);
```
- Entrée : a = matrice carré **symétrique réelle**; `b = vecteur`.
- Sortie : Diagonalise la matrice `a`; `b` devient la matrice des vecteurs propres.
- Rôle : Elle permet de donner la **matrice des vecteurs propres** et **leurs valeurs propres associées**, indispensable pour la résolution du problème.

Fonction `maxVP` :

- Prototype :
```
double maxVP(Matrice colVP, int *pos_out);
```
- Entrée : une matrice vecteur colonne, un pointeur vers un entier.
- Sortie : min = la valeur de la plus petite valeur propre. `pos_out` correspond désormais à la position de cette valeur propre.
- Rôle : renvoie la plus petite valeur propre (min) sa position dans la matrice.

Fonction CreateMatrice :

- Prototype :
```
Matrice createMatrice(int n, int m);
```
- Entrée : n = entier des lignes; m = entier des colonnes.
- Sortie : matrice remplie de zeros de taille n*m.
- Rôle : Crée des matrices de taille n*m indispensables pour le problème et utilisée très souvent.

Fonction FreeMatrice :

- Prototype :
```
void freeMatrice(Matrice *M);
```
- Entrée : une matrice M.
- Sortie : Rien.

- Rôle : Permet de supprimer l'espace alloué par la fonction "malloc" (utilisée dans la fonction "CreateMatrice") pour une matrice. Puisque meme si la matrice n'est plus utile, ou que le programme crash, l'espace reste alloué et cela permet de le libérer.

Fonction vpDiag :

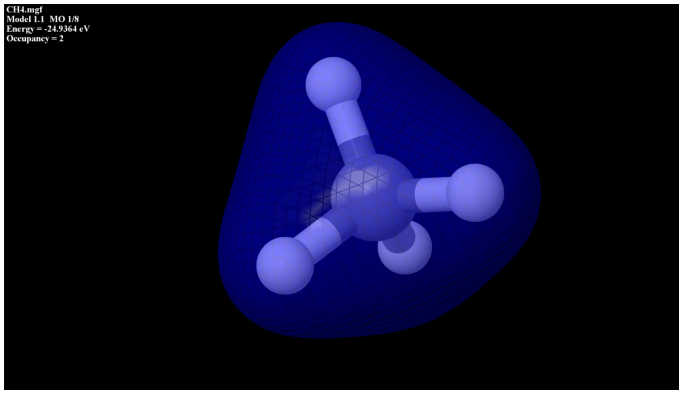
- Prototype :
  ```
  Matrice vpDiag(Matrice D);
  ```
- Entrée : une matrice Diagonale (D).
- Sortie : un vecteur colonne (res).
- Rôle : Donne un vecteur colonne des valeurs propres d'une matrice diagonale.

Fonction GetCol :

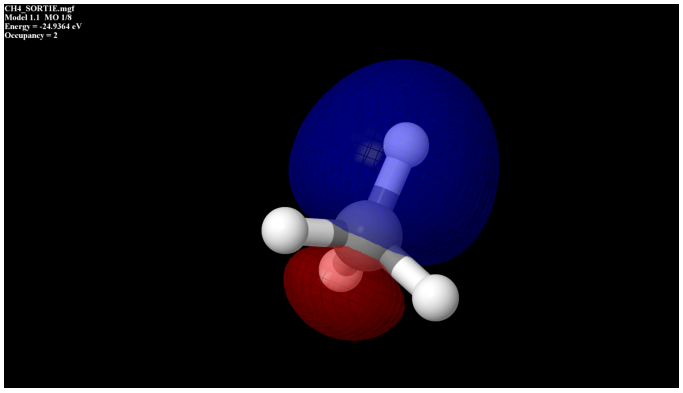
- Prototype :
  ```
  Matrice getCol(Matrice M, int pos);
  ```
- Entrée : une matrice M, un entier (pos) indiquant la colonne a extraire.
- Sortie : un vecteur colonne (res).
- Rôle : Permet de récupérer la colonne d'une matrice sous la forme d'un nouveau vecteur colonne à partir de son indice dans cette matrice.

4 Résultats & Analyse

A la fin de l'exécution du fichier "main.c", le programme écrit un fichier "CH4_output.mgf" (dans le dossier "output") en reprenant exactement le même formalisme que dans le fichier d'entrée "CH4.mgf". Finalement, seules les valeurs des coefficients des OA pour les OM concernées changent. C'est-à-dire que pour le cas de CH_4, seules les 4 premières OM occupées vont avoir leurs coefficients modifiés (puisqu'il y a 8 électrons de valence mis en jeu). On visualise donc via le logiciel Jmol les 4 nouvelles OM ainsi que les anciennes afin de les comparer. Afin de mieux visualiser les lobes,



(a) OM 1

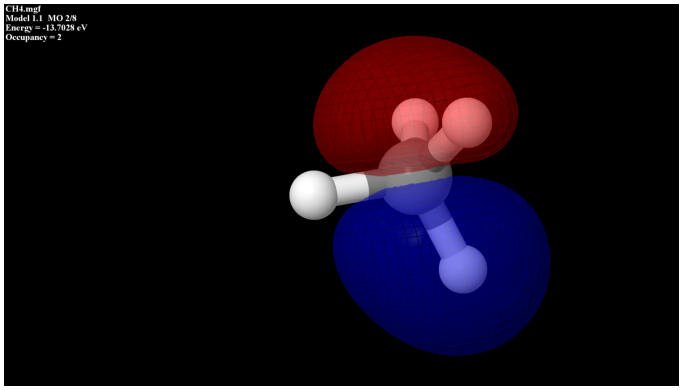


(b) OM_output 1

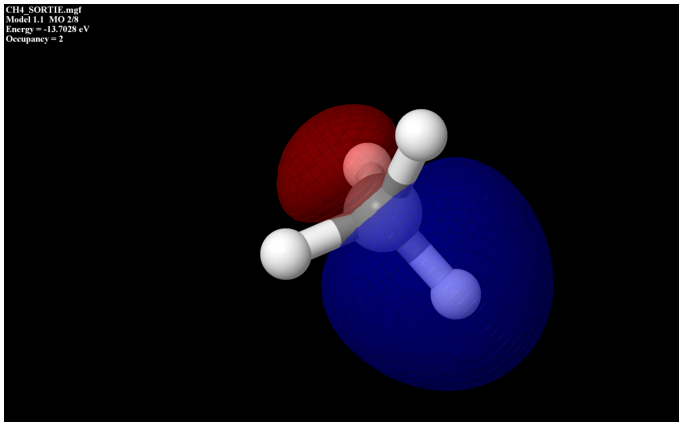
FIGURE 6 – Visualisation & Comparaison des OM n°1

Ici l'OM 1 ressemblait énormément à la 2s du carbone, et cela se confirmait via le fichier mgf avec un coefficient majoritaire sur la $2s_C$ parmi les 4 OA du carbone (+ de 99%). Cependant lorsque l'on visualise l'OM 1 en sortie de programme, celle-ci est localisée sur la liaison C-H_1, et l'on voit une belle

hybridation sp^3. Cela prouve que l'on ne peut pas uniquement se fier aux représentations surfaciques des orbitales pour étudier la localisation des électrons. Puisque malgré que la représentation de l'OM 1 soit véridique, les électrons se trouvent uniquement dans les volumes délimités par l'OM 1 sortante (notre programme restreint la délimitation).

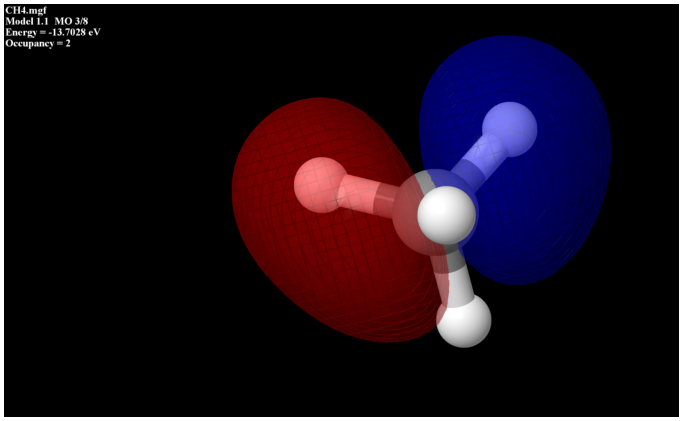


(a) OM 2

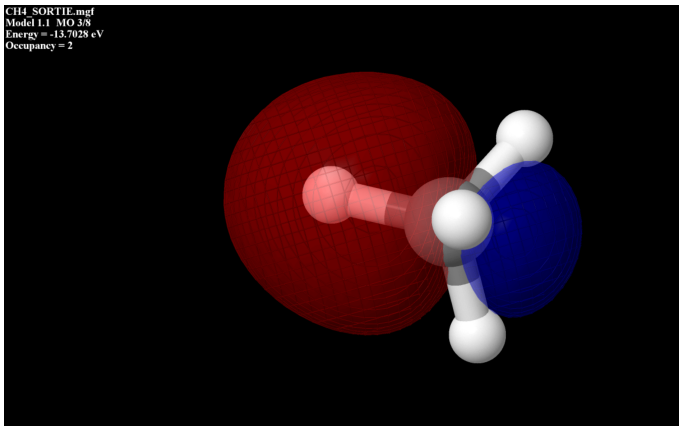


(b) OM_output 2

FIGURE 7 – Visualisation & Comparaison des OM n°2



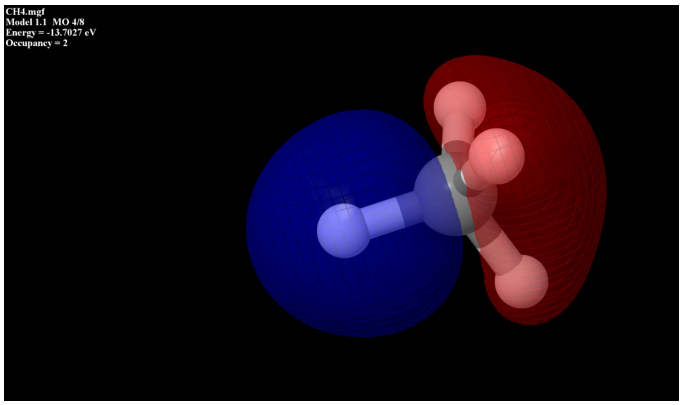
(a) OM 3



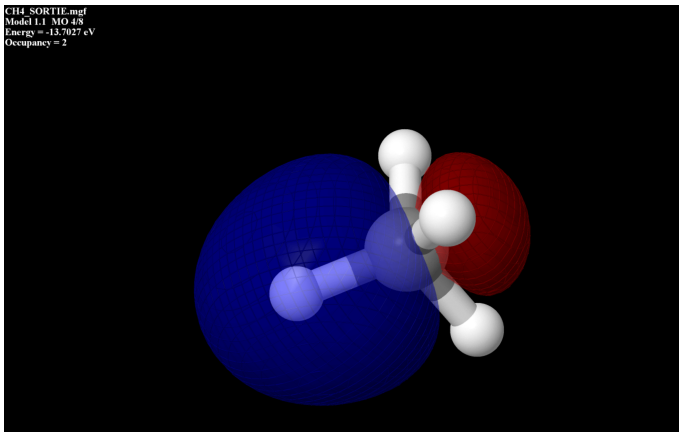
(b) OM_output 3

FIGURE 8 – Visualisation & Comparaison des OM n°3

A première vue, on peut considérer que la visualisation de l'OM 3 localisée n'est pas cohérente avec son OM initiale et avec les trois autres OM localisées. Cependant, on ne s'intéresse pas au signe des lobes et en ce qui concerne la couverture de l'orbitale rouge, celle-ci est parfaitement localisée sur la liaison C-H. On observe donc une OM résultant du recouvrement d'orbitales de type sp^3.



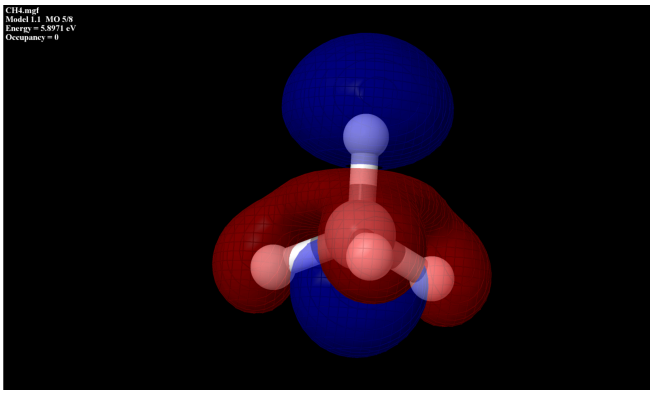
(a) OM 4



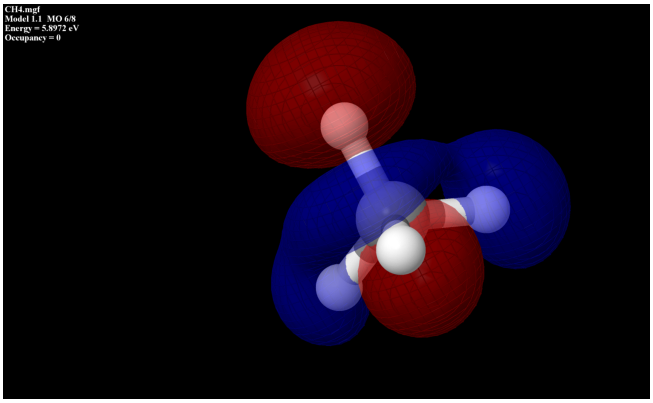
(b) OM_output 4

FIGURE 9 – Visualisation & Comparaison des OM n°4

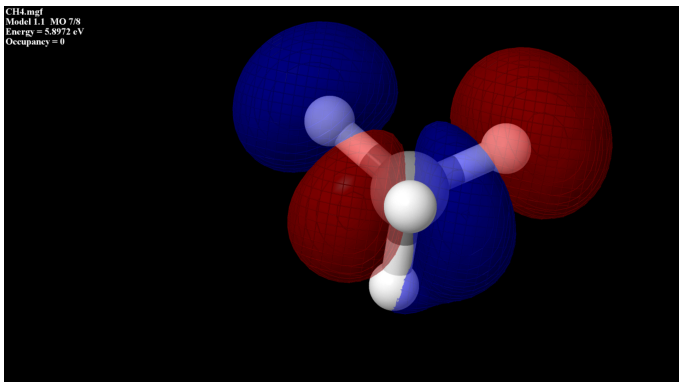
Pour les OM 2 et 4, la visualisation de celles-ci en sortie est **cohérente** avec leur OM non localisée puisque les lobes bleus englobant déjà la liaison C-H finissent par la recouvrir en exclusivité. Tandis que les lobes rouges suivent la forme de l'orbitale de type sp^3.



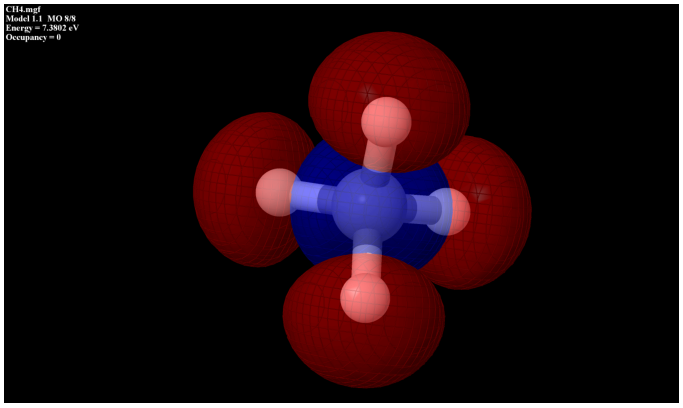
(a) OM 5



(b) OM 6



(c) OM 7



(d) OM 8

FIGURE 10 – Visualisation des quatres OM non occupées (donc non modifiées)

Voici les quatre autres OM de CH_4 non occupées par des électrons dans l'état fondamental, d'où le fait qu'elles soient restées intactes après utilisation du programme. Cela nous conforte également dans l'idée

que le programme fonctionne correctement (en tout cas pour le méthane).

Critère de réussite de la localisation d'orbitales du programme :

Finalement, il est intéressant de pouvoir vérifier sans même avoir besoin de visualiser les orbitales unes par unes (ce qui peut être long pour des grosses molécules et lorsque la quantité augmente) que la localisation des orbitales s'est bien faite sur chaque liaison de la molécule. Cette vérification est donc bien possible mais plusieurs conditions sont nécessaires, par ordre de priorité.

Il faut donc vérifier que l'on a UNIQUEMENT des contributions d'orbitales atomiques provenant des **deux atomes concernés** dans la liaison parmi toutes les orbitales mises en jeu dans l'OM (les autres doivent être négligeables, au minimum de l'ordre de 10^{-2}). Prenons le cas de CH_4, il faut donc une ou plusieurs OA de C (2s, $2p_x$ et $2p_y$ par exemple) et une OA d'un des H, tous les autres coefficients doivent être négligeables, il ne faut surtout pas de contribution d'OA d'autres hydrogènes sinon l'OM sera considérée comme non localisée.
De plus, il est nécessaire de **vérifier la normalisation au sein de chaque OM**. Donc pour le cas de CH_4, que pour chaque OM $\sum_{i=1}^{8} C_i^2 = 1$, ce que le programme fait automatiquement mais il n'interrompt pas le calcul si elle n'est pas vérifiée.
En effet, lors de notre court test avec NH_3, la normalisation n'était pas vérifiée sur la $4^{\text{ème}}$ OM (tous les coefs étaient nuls ...), ce qui induisait que malgré le fait que les trois autres OM occupées respectaient les deux règles, la localisation n'avait sûrement pas été fructueuse. Finalement, après visualisation les lobes n'étaient effectivement pas bien localisés sur les liaisons.

5 Conclusion & Perspectives

Globalement, l'application de notre programme sur le méthane est plutôt fructueuse puisque les lobes des OM pleines localisées sont cohérents et bien placés sur les liaisons. On retrouve bien des OM résultant d'un recouvrement d'orbitales hybrides sp^3. Les quatre autres OM non occupées n'ont pas été touchées, ce qui est normal mais heureux.
Cependant, nous avons été contraints de changer légèrement la méthode d'approche du programme préconisé par la publication puisque celle-ci considérait une minimisation des valeurs propres. Mais il s'est avéré plus pratique de partir sur une maximisation, et donc de considérer une Matrice R avec les données des deux atomes formant la liaison et non l'inverse.

Finalement, le sujet étant assez complexe, la lecture de la publication et la compréhension des objectifs du programme nous a pris plus de temps que prévu. Malgré cela, nous avons testé le programme sur la molécule NH_3 mais à cause du doublet non liant de N, un problème de dégénérescence des valeurs propres s'est posé. Par manque de temps, nous n'avons pas plus considéré le cas de NH_3, ce qui aurait été intéressant puisque les auteurs de la publication avaient étudié son cas et également expliqué le problème de dégénérescence.

Références

[1] WS Verwoerd. « The use of population localised orbitals to interpret molecular orbital calculations ». In : Chemical Physics 44.2 (1979), p. 151-162.

6 Annexe

Voici le code du programme, codé en C : Main.c :

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include <limits.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>


#include "../includes/matrice.h"
#include "../includes/luMethod.h"
#include "../includes/gestionfichier.h"

int main(int argc, const char * argv[]) {

    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof cwd)) printf("CWD = %s\n", cwd);

    Matrice E;

    int* aoParAtome;
    int natoms;
    if(lirefichier("data/CH4.mgf", &E, &aoParAtome, &natoms)!=1){
        printf("Erreur dans la lecture du fichier");
        exit(EXIT_FAILURE);
    }
    printf("Voici la matrice E comme elle est toute belle : \n");
    afficheMatrice(E);
    printf("On a : %d atomes ! \n", natoms);

    int nAO = E.n;

    //On stockera nos différents E_prime = EXij ici :
    int nTE_prime = natoms*(natoms-1)/2;
    Matrice TE_prime[nTE_prime];


    for (int i = 0; i < natoms; i++) {
        printf("Atome %d     %d AO\n", i+1, aoParAtome[i]);
    }
    int compteur = 0;
    //C'est parti pour traiter chaque liaison une par une
    for (int i = 0; i < natoms; i++) {
        for (int j = i + 1; j < natoms; j++) { //Liaison entre i et j
            printf("Itération %d\n", compteur);
            compteur++;
            //Choisissons les lignes à éliminer
            int *tokeep = malloc(nAO * sizeof(int));

            //Puis on assigne les lignes de la matrice E idoine dans R :
            int compte = 0;
            for(int k = 0; k<natoms; k++)
            {
                for(int l = k; l - k <aoParAtome[k]; l++)
                {
                    //Si c'est un atome concerné par la liaison : CA DEGAGE
                    if(k== i || k == j) tokeep[compte] = 0;
                    else tokeep[compte] = 1;
                    compte++;
                    //printf("l = %d k = %d AoPA = %d l+k= %d  :: \n", l, k,
                        aoParAtome[k], l+k);
```

```
                }

            }
            //for(int w = 0; w<nAO; w++){printf("%d  ", tokeep[w]);}
            //printf("\nON CHANGE DE LIAISON \n");

            //Maintenant qu'on a cette p- sacré liste, on crée notre variable R à partir de E.
            //Calculons la taille de R..
            int nRRow = nAO - aoParAtome[i] - aoParAtome[j];
            Matrice R = createMatrice(nRRow, nAO);

            //Pour chaque ligne, si l'AO contribue à la liason : ligne de 0, sinon : ligne de R.
            int compteR = 0;
            for(int i=0; i<nAO; i++){
                if(tokeep[i]){
                    for(int j =0; j<nAO; j++){
                        R.data[compteR][j] = E.data[i][j];
                    }
                    compteR++;
                }
        }

        //On calcule ensuite TRR (R^T*R)
        Matrice TR = transpose(R);
        Matrice TRR = produit(TR, R);
        //Ensuite on diagonalise la matrice
        Matrice VectPR = createMatrice(nAO, nAO);
        vpJacobi(TRR, VectPR);

        //Matrice colonne des VALEURS propres de R
        Matrice VPdeR = vpDiag(TRR);
        int position;
        double vp = minVP(VPdeR, &position);
        printf("Valeur propre : %g\n", vp);
        //On récupère le vecteur propre X correspondant à cette position
        Matrice X = getCol(VectPR, position);
        //On multiple E par ce vecteur colonne pour obtenir Li
        Matrice Li = produit(E, X);
        TE_prime[compteur] = copieMatrice(Li);
    }
}

//NORMALEMENT on a toute nos matrice Eprime, faisons en la moyenne !
Matrice EPrime = meanMatrices(TE_prime, nTE_prime);
printf("La sainte Matrice E' : \n");
afficheMatrice(EPrime);
int res = ecrire_orbital_replaced("data/CH4.mgf", "output/CH4.mgf", &EPrime, nAO);
printf("Conclusion du code : %d \n", res);
return EXIT_SUCCESS;
```

matrice.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "matrice.h"

#define TOL 1e-12

void jacobi(double **a, double **b, int n);

//M[i] = Pointeur de la ligne
Matrice createMatrice(int n, int m) //LiCo //Crée une matrice de taille m*n sans valeur particulière
{
    Matrice matrice;
```

```
    matrice.n = n; //ligne
    matrice.m = m; //colonne
    //Allocation du tableau de pointeur
    matrice.data = malloc(n * sizeof(double));
    if(!matrice.data)
    {
        printf("Echec de l'allocation mémoire lors de la création \n");
        exit(EXIT_FAILURE);
    }

    //Allocation des lignes
    for( int i = 0; i<n ; i++)
    {
        matrice.data[i] = malloc(m * sizeof(double));
        if(!matrice.data[i])
        {
            printf("Echec de l'allocation mémoire lors de la création \n");
            exit(EXIT_FAILURE);
        }
        for(int j=0; j<m; j++)
        {
            matrice.data[i][j] = 0;
        }
    }

    return matrice;
}

void freeMatrice(Matrice *M) {
    if (!M || !M->data) return;

    // libérer chaque ligne
    for (int i = 0; i < M->n; i++) {
        free(M->data[i]);
    }

    // libérer le tableau de pointeurs
    free(M->data);

    // mettre les champs à zéro pour éviter les "dangling pointers"
    M->data = NULL;
    M->n = 0;
    M->m = 0;
}

Matrice copieMatrice(Matrice M){
    Matrice res = createMatrice(M.n, M.m);
    for (int i=0; i<M.n; i++) {
        for (int j=0; j<M.m; j++) {
            res.data[i][j] = M.data[i][j];
        }
    }
    return res;
}

void afficheMatrice(Matrice mat) //Affiche la matrice
{
    if(mat.n < 0 || mat.m < 0){
        printf("Affichage matrice impossible\n");
        exit(EXIT_FAILURE);
    }
    for(int i = 0; i<mat.n; i++)
    {
        for(int j=0; j<mat.m;j++)
        {
            printf("%10.2g", mat.data[i][j]);
        }
```

```
81          printf("\n");
82      }
83  }
84
85  //Génére une matrice de test où chaque coeff = id_coeff
86  void testMatrice(Matrice M)
87  {
88      int k = 0;
89      for(int i=0; i<M.n; i++)
90      {
91          for(int j=0; j<M.m; j++)
92          {
93              M.data[i][j] = k;
94              k++;
95          }
96      }
97  }
98
99  //Crée la matrice transposé
100 Matrice transpose(Matrice M)
101 {
102     double a = M.n, b = M.m;
103     //Dans M : 0<i<a ; 0<j<b
104     //Dans MT : 0<i<b; 0<j<a
105     Matrice MT = createMatrice(b, a);
106     for(int i=0; i<a; i++)
107     {
108         for(int j=0; j<b; j++)
109         {
110             MT.data[j][i] = M.data[i][j];
111         }
112     }
113
114     return MT;
115 }
116
117 //Crée le produit de deux matrices
118 Matrice produit(Matrice M, Matrice N)
119 {
120     if(M.m != N.n)
121     {
122         printf("Produit Matriciel invalide !! \n");
123         exit(EXIT_FAILURE);
124     }
125     int n_somme = M.m;
126     Matrice res = createMatrice(M.n, N.m);
127     for(int i=0; i<res.n; i++)
128     {
129         for(int j=0; j<res.m; j++)
130         {
131             double coeff = 0;
132             for(int k = 0; k<n_somme; k++)
133             {
134                 coeff += M.data[i][k]*N.data[k][j];
135             }
136             res.data[i][j] = coeff;
137         }
138     }
139     return res;
140 }
141
142 void pScaMat(Matrice M, double a){
143     for(int i=0; i<M.n; i++)
144     {
145         for(int j=0; j<M.m; j++)
146         {
147             M.data[i][j] = a*M.data[i][j];
```

```
        }
    }
}

double pScalaire(Matrice X)
{
    if(X.m != 1)
    {
        printf("Pas un vecteur colonne !!! \n");
        exit(EXIT_FAILURE);
    }
    Matrice TX = transpose(X);
    return produit(X, TX).data[0][0];

}
//Addition M + aN
Matrice addMat(Matrice M, Matrice N, double a)
{
    if(M.n != N.n || M.m != N.m)
    {
        printf("Addition entre deux matrices de tailles différentes !! \n");
        exit(EXIT_FAILURE);
    }
    int n = M.n, m = M.m;
    Matrice res = createMatrice(n, m);

    for(int i=0; i<n;i++)
    {
        for(int j=0; j<m; j++)
        {
            res.data[i][j] = M.data[i][j] + a*N.data[i][j];
        }
    }
    return res;
}

double norme(Matrice M)
{
    return sqrt(pScalaire(M));
}

//Génère une matrice identité de taille n
Matrice Identite(int n)
{
    Matrice I = createMatrice(n, n);
    for(int i=0;i<n;i++)
    {
        for(int j=0; j<n; j++)
        {
            I.data[i][j] = i == j ? 1 : 0;
        }
    }

    return I;
}

//Donne un vecteur colonne des valeurs propre d'une matrice diagonale
Matrice vpDiag(Matrice D)
{
    if(D.n != D.m){
        printf("Matrice diagonale non carrée");
        exit(EXIT_FAILURE);
    }
    Matrice res = createMatrice(D.n, 1);
    for(int i=0; i<D.n; i++){
        res.data[i][0] = D.data[i][i];
    }
```

```
    return res;
}


double minVP(Matrice colVP, int *pos_out){
    if(colVP.m != 1){
        printf("Pas une matrice colonne");
        exit(EXIT_FAILURE);
    }
    double min = colVP.data[0][0];
    double pos = 0;
    for(int i=0; i<colVP.n; i++){
        if(fabs(colVP.data[i][0]) < fabs(min)) {
            min = colVP.data[i][0];
            pos = i;
        }
    }

    if(pos_out) *pos_out = pos;
    return min;
}

Matrice getCol(Matrice M, int pos){
    if(pos>= M.m){
        printf("Postion mauvaise");
        exit(EXIT_FAILURE);
    }

    Matrice res = createMatrice(M.m, 1);

    for (int i=0; i<M.n; i++) {
        res.data[i][0] = M.data[i][0];
    }
    return res;
}

//Compatibilité Structure Matrice <-> Fonction Jacobi des TP 1A
void vpJacobi(Matrice a, Matrice b)
{
    if(a.n != a.m || b.n != b.m || a.n != b.n)
    {
        printf("Entrée invalide pour Jacobi");
        exit(EXIT_FAILURE);
    }
    jacobi(a.data, b.data, a.n);
}

//Moyenne arithmétique d'une liste de matrice
Matrice meanMatrices(const Matrice *list, int count) {
    if (!list || count <= 0) {
        // signal d'erreur : matrice vide (pas d'allocation)
        Matrice err = {0, 0, NULL};
        return err;
    }

    const int n = list[0].n;
    const int m = list[0].m;

    // Vérification des dimensions
    for (int k = 0; k < count; k++) {
        Matrice justpourvoir = list[k];
        if (list[k].n != n || list[k].m != m || list[k].data == NULL) {
            Matrice err = {0, 0, NULL};
            return err;
        }
    }

```

```
    // Somme : S = list[0] + list[1] + ... + list[count-1]
    Matrice S = copieMatrice(list[0]);       // S = list[0]
    for (int k = 1; k < count; k++) {
        Matrice tmp = addMat(S, list[k], 1.0); // tmp = S + 1.0 * list[k]
        freeMatrice(&S);
        S = tmp;
    }

    // Moyenne : S *= 1/count (in place)
    pScaMat(S, 1.0 / (double)count);
    return S; //    libérer par l'appelant avec freeMatrice(&S)
}

//Donne la matrice des vecteurs propres
//OUTPUT : a : matrice diagonalisé b : matrice des vecteurs propres
//a = matrice carré; b = vecteur; n = taille
void jacobi(double **a, double **b, int n)
{
    int i,j,k,l;
    double M,alpha,t,h,c,s,r,u,v;
    int iter=0;

    for (i=0;i<n;i++)
    { for (j=0;j<n;j++) b[i][j]=0;
        b[i][i]=1;
    }

    do {
        M=0;
        for (i=0;i<n-1;i++)
            for (j=i+1;j<n;j++)
                if (fabs(a[i][j])>M) {
                    k=i;
                    l=j;
                    M=fabs(a[i][j]);
                }

        if (M<TOL) break;
        iter++;
        alpha=0.5*(a[l][l]-a[k][k])/a[k][l];
        t=sqrt(1+alpha*alpha);
        if (alpha<0)
            t=-t;
        t-=alpha;
        h=t*a[k][l];
        c=1/sqrt(1+t*t);
        s=t*c;
        r=s/(1+c);
        for (j=0;j<n;j++) {
            u=b[j][k];
            v=b[j][l];
            b[j][k]-=s*(v+r*u);
            b[j][l]+=s*(u-r*v);
            if (j!=k && j!=l) {
                u=a[k][j];
                v=a[l][j];
                a[k][j]-=s*(v+r*u);
                a[l][j]+=s*(u-r*v);
                a[j][k]=a[k][j];
                a[j][l]=a[l][j];
            }
        }
        a[k][k]-=h;
        a[l][l]+=h;
        a[k][l]=a[l][k]=0;
    } while (M>TOL && iter<1000);
    }
```

matrice.h :

```h
#ifndef MATRICE_H
#define MATRICE_H

//Définition d'une Matrice
typedef struct {
    int n;
    int m;
    double** data;
} Matrice;

Matrice createMatrice(int n, int m);
void freeMatrice(Matrice *M);
Matrice copieMatrice(Matrice M);
void pScaMat(Matrice M, double a);
double pScalaire(Matrice X);
Matrice addMat(Matrice M, Matrice N, double a);
double norme(Matrice M);
Matrice Identite(int n);
void afficheMatrice(Matrice mat);
void testMatrice(Matrice M);
Matrice transpose(Matrice M);
Matrice produit(Matrice M, Matrice N);
Matrice vpDiag(Matrice D);
double minVP(Matrice colVP, int *pos_out);
Matrice getCol(Matrice M, int pos);
Matrice meanMatrices(const Matrice *list, int count);
void vpJacobi(Matrice a, Matrice b);

#endif
```

luMethod.c :

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "matrice.h"
#include "luMethod.h"

/* INPUT: A - array of pointers to rows of a square matrix having dimension N
 *        Tol - small tolerance number to detect failure when the matrix is near degenerate
 * OUTPUT: Matrix A is changed, it contains a copy of both matrices L-E and U as A=(L-E)+U such that P*A=L*U.
 *        The permutation matrix is not stored as a matrix, but in an integer vector P of size N+1
 *        containing column indexes where the permutation matrix has "1". The last element P[N]=S+N,
 *        where S is the number of row exchanges needed for determinant computation, det(P)=(-1)^S
 */
int LUPDecompose(Matrice A, double Tol, int *P) {
    if(A.n != A.m)
    {
        printf("Décomposition LU sur une matrice non carré !!!\n");
        exit(EXIT_FAILURE);
    }

    int i, j, k, imax;
    int N = A.n;
    double maxA, *ptr, absA;

    for (i = 0; i <= N; i++)
        P[i] = i; //Unit permutation matrix, P[N] initialized with N

    for (i = 0; i < N; i++) {
```

```
30          maxA = 0.0;
31          imax = i;
32
33          for (k = i; k < N; k++)
34              if ((absA = fabs(A.data[k][i])) > maxA) {
35                  maxA = absA;
36                  imax = k;
37              }
38
39          if (maxA < Tol) return 0; //failure, matrix is degenerate
40
41          if (imax != i) {
42              //pivoting P
43              j = P[i];
44              P[i] = P[imax];
45              P[imax] = j;
46
47              //pivoting rows of A
48              ptr = A.data[i];
49              A.data[i] = A.data[imax];
50              A.data[imax] = ptr;
51
52              //counting pivots starting from N (for determinant)
53              P[N]++;
54          }
55
56          for (j = i + 1; j < N; j++) {
57              A.data[j][i] /= A.data[i][i];
58
59              for (k = i + 1; k < N; k++)
60                  A.data[j][k] -= A.data[j][i] * A.data[i][k];
61          }
62      }
63
64      return 1;  //decomposition done
65  }
66
67
68  /* INPUT: A,P filled in LUPDecompose; N - dimension
69   * OUTPUT: IA is the inverse of the initial matrix
70   */
71  Matrice LUPInvert(Matrice A, int *P) {
72
73      Matrice IA = copieMatrice(A);
74
75      if(A.n != A.m)
76      {
77          printf("Inversion LU sur une matrice non carré !!!\n");
78          exit(EXIT_FAILURE);
79      }
80      int N = A.n;
81
82      for (int j = 0; j < N; j++) {
83          for (int i = 0; i < N; i++) {
84              IA.data[i][j] = P[i] == j ? 1.0 : 0.0;
85
86              for (int k = 0; k < i; k++)
87                  IA.data[i][j] -= A.data[i][k] * IA.data[k][j];
88          }
89
90          for (int i = N - 1; i >= 0; i--) {
91              for (int k = i + 1; k < N; k++)
92                  IA.data[i][j] -= A.data[i][k] * IA.data[k][j];
93
94              IA.data[i][j] /= A.data[i][i];
95          }
96      }
```

```
97
98      return IA;
99  }
100
101 /* INPUT: A,P filled in LUPDecompose; N - dimension.
102  * OUTPUT: Function returns the determinant of the initial matrix
103  */
104 double LUPDeterminant(Matrice A, int *P) {
105
106     if(A.n != A.m)
107     {
108         printf("Calcul de determinant LU sur une matrice non carré !!!\n");
109         exit(EXIT_FAILURE);
110     }
111     int N = A.n;
112
113     double det = A.data[0][0];
114
115     for (int i = 1; i < N; i++)
116         det *= A.data[i][i];
117
118     return (P[N] - N) % 2 == 0 ? det : -det;
119 }
```

luMethod.h :

```
1  #ifndef LUMETHOD_H
2  #define LUMETHOD_H
3
4  #include "matrice.h"
5
6  int LUPDecompose(Matrice A, double Tol, int *P);
7  Matrice LUPInvert(Matrice A, int *P);
8  double LUPDeterminant(Matrice A, int *P);
9
10 #endif
```

gestionfichier.c :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <string.h>
5  #include <ctype.h>
6
7  #include "gestionfichier.h"
8  #include "matrice.h"
9
10 void sauteLigne(FILE *f);
11 int ao_count_for_Z(int Z);
12
13 int lirefichier(const char *path, Matrice *E,int **aoParAtome, int *natoms_out)
14 {
15     FILE *f;
16
17     f=fopen(path,"r");
18     if (f==0) {printf("Problème d'ouverture du fichier !\n"); exit(EXIT_FAILURE);}
19     //On veut le nombre d'atomes
20     int natoms = 0;
21     if (fscanf(f, " %d", &natoms) != 1) {   // " %d" saute les espaces initiaux
22         fprintf(stderr, "Impossible de lire le nombre d'atomes en tête de fichier.\
           n");
23         fclose(f);
24         return 2;
25     }
26     printf("Nombre d'atomes = %d\n", natoms);
27     //Sautons une ligne
28     sauteLigne(f);
```

```
29
30      //C'est l'occasion d'allouer le tableau :)
31      if (aoParAtome) {
32          *aoParAtome = malloc(natoms * sizeof(int));
33          if (!*aoParAtome) { fclose(f); return -100; }
34      }
35
36
37      //On en déduit le nombre d'AO du système
38      //Pour cela on prend trouve les éléments chimiques de chacun et on additionne
            selon la période.
39      int nAO = 0;
40      for(int i=0; i<natoms;i++)
41      {
42          int Z;
43          //On récupère son numéro atomique
44          if(fscanf(f, " %d %*lf %*lf %*lf %*lf", &Z) != 1){
45              fprintf(stderr, "Ligne d'atome %d invalide.\n", i+1);
46              fclose(f);
47              exit(EXIT_FAILURE);
48          }
49          if(Z<3)//Période 1 -> 1 AO
50          {
51              nAO += 1;
52              if (aoParAtome && *aoParAtome) (*aoParAtome)[i] = 1;
53          }
54          else if(Z==6 || Z==7 || Z==8 || Z==9)//Période 2 -> 4 AO
55          {
56              nAO += 4;
57              if (aoParAtome && *aoParAtome) (*aoParAtome)[i] = 4;
58          }
59          else
60          {
61              printf("Numéro atomique trop lourd !");
62              exit(EXIT_FAILURE);
63          }
64      }
65
66      printf("Nombre Orbitale Atomique : %d \n", nAO);
67
68      //Maitenant on prépare notre matrice E : E[OA][OM] (je suis trop hype)
69      *E = createMatrice(nAO, nAO);
70
71      //C'est parti pour chercher les nAO dans le fichier
72      char line[4096];
73      int count = 0;
74
75      int i=0;
76
77      while (fgets(line, sizeof line, f) && i<nAO) {
78          // Tolère des espaces avant "ORBITAL"
79          const char *p0 = line;
80          while (*p0==' ' || *p0=='\t' || *p0=='\r') p0++;
81          if (strncmp(line, " ORBITAL", 7) == 0) {
82              // Récupérer l'énergie (eV) écrite après "ORBITAL"
83              // Exemple de ligne : " ORBITAL 2" ou "ORBITAL 0"
84              double ev = 0.0;
85              const char *p = p0 + 7;              // après "ORBITAL"
86              while (*p==' ' || *p=='\t') p++;     // espaces
87              if (*p) {
88                  char *endp = NULL;
89                  // strtod accepte 2, 2.0, +2.3e-1, etc.
90                  double tmp = strtod(p, &endp);
91                  if (endp != p) ev = tmp;
92              }
93              if (ev_out) {
94                  ev_out->data[i][0] = ev; // stocke l'énergie de la MO i (en eV)
```

```
            }
            int got = 0;  // nb de doubles collectés pour cette orbitale

            // Lire des lignes suivantes et accumuler des doubles jusqu'à en avoir nAO
            while (got < nAO && fgets(line, sizeof line, f)) {
                char *p = line;
                while (*p && got < nAO) {
                    // sauter espaces
                    while (*p == ' ' || *p == '\t' || *p == '\r' || *p == '\n') p++;
                    if (!*p) break;

                    char *endp;
                    double v = strtod(p, &endp);
                    if (endp == p) break; // rien de lisible sur ce reste de ligne

                    E->data[got][i] = v;   // j = got
                    got++;
                    p = endp;
                }
            }

            if (got != nAO) {
                fprintf(stderr, "Erreur: orbitale %d incomplète (%d/%d coeffs lus).\n",
                        i+1, got, nAO);
                fclose(f);
                return 10;
            }
            i++;        // prochaine orbitale
            count++;    // compteur d'ORBITAL vues
        }
    }

    printf("Total ORBITAL: %d\n", count);
    *natoms_out = natoms;
    fclose(f);
    return 1;
}

// saute le reste de la ligne courante (jusqu'à et y compris '\n')
void sauteLigne(FILE *f) {
    int ch;
    while ((ch = fgetc(f)) != EOF && ch != '\n') {
        // on boucle tant qu'on n'a pas rencontré un retour ligne
    }
}

int ao_count_for_Z(int Z) {
    if (Z < 3) return 1;                          // H (He non utilisé ici)
    if (Z==6 || Z==7 || Z==8 || Z==9) return 4; // C, N, O, F
    return 0; // non géré ici
}

/* saute espaces initiaux (ne modifie pas la chaîne d'origine) */
static const char* lskip(const char *s) {
    while (*s == ' ' || *s == '\t' || *s == '\r') ++s;
    return s;
}

/* compte le nb de doubles lisibles dans 'line' (tolère D/d     E) ;
   renvoie aussi si la ligne contenait un 'D' */
static int count_doubles_in_line(const char *line, int *saw_D) {
    char tmp[4096];
    size_t L = strlen(line);
    if (L >= sizeof tmp) L = sizeof tmp - 1;
```

```
159     memcpy(tmp, line, L); tmp[L] = '\0';
160 
161     int saw = 0;
162     for (size_t i = 0; i < L; ++i) {
163         if (tmp[i] == 'D' || tmp[i] == 'd') { tmp[i] = 'E'; saw = 1; }
164     }
165     if (saw_D) *saw_D |= saw;
166 
167     int count = 0;
168     char *p = tmp;
169     for (;;) {
170         while (isspace((unsigned char)*p)) ++p;
171         if (!*p) break;
172         char *endp;
173         (void)strtod(p, &endp);
174         if (endp == p) break;
175         ++count;
176         p = endp;
177     }
178     return count;
179 }
180 
181 /* imprime v dans buf, fmt %.12E, puis remplace  E D  si use_D */
182 static void format_double_ED(char *buf, size_t bufsz, double v, int use_D) {
183     snprintf(buf, bufsz, "%.12E", v);
184     if (use_D) {
185         for (char *p = buf; *p; ++p) if (*p == 'E') *p = 'D';
186     }
187 }
188 
189 /* --- fonction principale --- */
190 
191 int ecrire_orbital_replaced(const char *in_path, const char *out_path,const Matrice
        *E, int nAO)
192 {
193     if (!in_path || !out_path || !E || nAO <= 0) return -1;
194 
195     FILE *in = fopen(in_path, "r");
196     if (!in) return -2;
197 
198     FILE *out = fopen(out_path, "w");
199     if (!out) { fclose(in); return -3; }
200 
201     char line[4096];
202     int mo = 0; /* index du bloc ORBITAL courant (0..nAO-1) */
203 
204     while (fgets(line, sizeof line, in)) {
205         const char *q = lskip(line);
206 
207         /* Si ce n'est pas un début de bloc ORBITAL, recopier tel quel */
208         if (strncmp(q, "ORBITAL", 7) != 0) {
209             fputs(line, out);
210             continue;
211         }
212 
213         /* Début de bloc ORBITAL : recopier l'en-tête */
214         fputs(line, out);
215 
216         /* On va consommer dans l'entrée les lignes suivantes jusqu'à lire nAO
               nombres.
217            Pour chaque ligne de nombres source (k nombres), on écrit k nombres
218            pris dans E->data[oa][mo], en gardant l'indentation initiale. */
219         if (mo >= nAO) {
220             /* Plus de colonnes disponibles : on passe à travers sans remplacer */
221             continue;
222         }
223 
```

```
        int got = 0;            /* nb de coeffs écrits pour cette ORBITAL */
        int prefer_D = 0;       /* si la première ligne numérique source utilise D,
           on l'imite */

        while (got < nAO && fgets(line, sizeof line, in)) {
            /* détecter indentation de la ligne source */
            size_t indent = 0;
            while (line[indent] == ' ' || line[indent] == '\t') ++indent;

            /* combien de nombres sur cette ligne ? */
            int k = count_doubles_in_line(line, &prefer_D);

            if (k == 0) {
                /* ligne sans nombres (vide/commentaire) : recopier telle quelle */
                fputs(line, out);
                continue;
            }

            /* écrire une ligne avec le même nombre de nombres (k), depuis E */
            if (indent > 0) fwrite(line, 1, indent, out); /* réécrit juste l'
               indentation */

            for (int t = 0; t < k && got < nAO; ++t, ++got) {
                char numbuf[64];

                // Vérifs défensives
                if (!E)             { fprintf(stderr,"E NULL\n"); exit(EXIT_FAILURE)
                   ; }
                if (!E->data)       { fprintf(stderr,"E->data NULL\n"); exit(
                   EXIT_FAILURE); }
                if (got < 0 || got >= nAO) { fprintf(stderr,"got=%d hors [0,%d)\n",
                    got, nAO); exit(EXIT_FAILURE); }
                if (mo  < 0 || mo  >= nAO) { fprintf(stderr,"mo=%d hors [0,%d)\n",
                    mo,  nAO); exit(EXIT_FAILURE); }
                if (!E->data[got]) { fprintf(stderr,"E->data[%d] NULL\n", got);
                   exit(EXIT_FAILURE); }

                format_double_ED(numbuf, sizeof numbuf, E->data[got][mo], prefer_D)
                   ;
                if (t > 0) fputc(' ', out);
                fputs(numbuf, out);
            }
            fputc('\n', out);

            /* Si la ligne source contenait plus de nombres que nécessaire (rare),
               on ignore l'excédent, car on s'arrête quand got==nAO. */
            if (got >= nAO) break;
        }

        ++mo; /* bloc ORBITAL suivant */
    }

    fclose(in);
    fclose(out);
    return 0;
}
```

gestionfichier.h :

```
#ifndef GESTIONFICHIER_H
#define GESTIONFICHIER_H

#include "matrice.h"

int lirefichier(const char *path, Matrice *E, int **aoParAtome, int *natoms_out);
int ecrire_orbital_replaced(const char *in_path, const char *out_path,const Matrice
    *E, int nAO);

```

```
#endif
```

```
 5 MOPAC-Graphical data Version 2016.21.174M
  6    0.0000000   0.0000000   0.0000000  -0.6027
  1    1.0890000   0.0000000   0.0000000   0.1507
  1   -0.3629960   1.0267204   0.0000000   0.1507
  1   -0.3629960  -0.5133602  -0.8891660   0.1507
  1   -0.3629960  -0.5133602   0.8891660   0.1507
 1.9422440  1.7087230  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
ORBITAL 2 1a1    -24.9364
0.72345307E+00 0.14590141E-05 0.23727747E-12 0.61411014E-13 0.34518756E+00
0.34518652E+00 0.34518652E+00 0.34518652E+00
ORBITAL 2 1t2    -13.7028
-0.18322931E-11-0.33705004E-06 0.76637410E+00 0.72938922E-01-0.24199805E-06
0.51877682E+00-0.30214749E+00-0.21662908E+00
ORBITAL 2 1t2    -13.7028
0.12343088E-11 0.21443345E-06-0.72938922E-01 0.76637410E+00 0.15396071E-06
-0.49374131E-01-0.42458684E+00 0.47396082E+00
ORBITAL 2 1t2    -13.7027
-0.22420909E-05 0.76983766E+00 0.35585218E-06-0.18153566E-06 0.55273178E+00
-0.18424376E+00-0.18424401E+00-0.18424423E+00
ORBITAL 0 2t2      5.8971
0.22846732E-05-0.63823975E+00 0.75404303E-06-0.39381018E-06 0.66669795E+00
-0.22223475E+00-0.22223398E+00-0.22223331E+00
ORBITAL 0 2t2      5.8972
0.95000346E-11 0.69222934E-06 0.63208898E+00 0.88397820E-01-0.72310189E-06
-0.62251113E+00 0.38665070E+00 0.23586115E+00
ORBITAL 0 2t2      5.8972
-0.61337663E-11-0.49444947E-06-0.88397820E-01 0.63208898E+00 0.51650095E-06
0.87058217E-01 0.49558129E+00-0.58264003E+00
ORBITAL 0 2a1      7.3802
0.69037356E+00 0.30833843E-05-0.89075279E-11 0.38725203E-11-0.36172776E+00
-0.36172613E+00-0.36172613E+00-0.36172613E+00
```

(a) CH_4.mgf

```
 5 MOPAC-Graphical data Version 2016.21.174M
  6    0.0000000   0.0000000   0.0000000  -0.6027
  1    1.0890000   0.0000000   0.0000000   0.1507
  1   -0.3629960   1.0267204   0.0000000   0.1507
  1   -0.3629960  -0.5133602  -0.8891660   0.1507
  1   -0.3629960  -0.5133602   0.8891660   0.1507
 1.9422440  1.7087230  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
ORBITAL 2 1a1    -24.9364
0.34065454E+00 0.67915111E+00-0.24981732E-10 0.10328300E-09 0.65016017E+00
-0.11766868E-08 0.31153784E-08-0.30762169E-08
ORBITAL 2 1t2    -13.7028
0.34065701E+00-0.22638344E+00 0.64030867E+00 0.39892292E-09-0.81353949E-09
0.65016010E+00-0.10943049E-09-0.98279824E-09
ORBITAL 2 1t2    -13.7028
-0.34065701E+00 0.22638344E+00 0.32015433E+00 0.55452357E+00-0.27872291E-08
0.71991273E-09-0.65016009E+00 0.35663792E-09
ORBITAL 2 1t2    -13.7027
0.34065701E+00-0.22638345E+00-0.32015433E+00 0.55452357E+00-0.31384724E-08
-0.12740856E-08 0.48652019E-10 0.65016010E+00
ORBITAL 0 2t2      5.8971
0.22846732E-05-0.63823975E+00 0.75404303E-06-0.39381018E-06 0.66669795E+00
-0.22223475E+00-0.22223398E+00-0.22223331E+00
ORBITAL 0 2t2      5.8972
0.95000346E-11 0.69222934E-06 0.63208898E+00 0.88397820E-01-0.72310189E-06
-0.62251113E+00 0.38665070E+00 0.23586115E+00
ORBITAL 0 2t2      5.8972
-0.61337663E-11-0.49444947E-06-0.88397820E-01 0.63208898E+00 0.51650095E-06
0.87058217E-01 0.49558129E+00-0.58264003E+00
ORBITAL 0 2a1      7.3802
0.69037356E+00 0.30833843E-05-0.89075279E-11 0.38725203E-11-0.36172776E+00
-0.36172613E+00-0.36172613E+00-0.36172613E+00
```

(b) CH_4_out.mgf

FIGURE 11 – Fichiers Entrée et Sortie MGF pour CH_4

```
 4 MOPAC-Graphical data Version 2016.21.174M
  7    0.0000000   0.0000000   0.0000000  -0.7104
  1    1.0000000   0.0000000   0.0000000   0.2310
  1   -0.2923717   0.9563048   0.0000000   0.2397
  1   -0.2923717  -0.4781524  -0.8281842   0.2397
 2.3543440  2.0282880  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
ORBITAL 2 1a'    -35.7296
0.84744187E+00 0.52539769E-01 0.62009036E-01-0.10740280E+00 0.29892021E+00
0.29525250E+00 0.29525250E+00
ORBITAL 2 1a"    -16.3236
-0.21770356E-11-0.42227656E-10 0.66654444E+00 0.38482961E+00-0.31137553E-10
0.45145580E+00-0.45145580E+00
ORBITAL 2 2a'    -15.9764
-0.13511489E-01 0.72617566E+00-0.13355668E+00 0.23132696E+00 0.52217129E+00
-0.25345013E+00-0.25345013E+00
ORBITAL 2 3a'    -10.2755
0.30777417E+00-0.31213361E+00-0.42880646E+00 0.74271458E+00-0.14994457E+00
-0.15789930E+00-0.15789930E+00
ORBITAL 0 4a'      4.5759
-0.43229086E+00-0.13130205E+00-0.18147217E+00 0.31431902E+00 0.45147483E+00
0.47975320E+00 0.47975320E+00
ORBITAL 0 5a'      6.7590
-0.77066074E-02-0.59602086E+00 0.10728635E+00-0.18582542E+00 0.64161589E+00
-0.30576735E+00-0.30576735E+00
ORBITAL 0 2a"      6.9443
-0.50250671E-11 0.69599173E-10 0.55291817E+00 0.31922745E+00-0.71362607E-10
-0.54423126E+00 0.54423126E+00
```

(a) NH_3.mgf

```
 4 MOPAC-Graphical data Version 2016.21.174M
  7    0.0000000   0.0000000   0.0000000  -0.7104
  1    1.0000000   0.0000000   0.0000000   0.2310
  1   -0.2923717   0.9563048   0.0000000   0.2397
  1   -0.2923717  -0.4781524  -0.8281842   0.2397
 2.3543440  2.0282880  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
 1.2602370  0.0000000  0.0000000
ORBITAL 2 1a'    -35.7296
0.37895505E+00 0.68533610E+00 0.34707579E-01-0.60115287E-01 0.61797171E+00
0.72421137E-08 0.72421137E-08
ORBITAL 2 1a"    -16.3236
0.48615110E+00-0.20987671E+00 0.59475149E+00 0.10808685E-02 0.62524713E-08
0.60487881E+00 0.13024423E-08
ORBITAL 2 2a'    -15.9764
0.54027113E+00-0.23060890E+00-0.32580591E+00-0.44516079E+00 0.37413730E-08
0.13927107E-08 0.59212247E+00
ORBITAL 2 3a'    -10.2755
0.00000000E+00 0.00000000E+00 0.00000000E+00 0.00000000E+00 0.00000000E+00
0.00000000E+00 0.00000000E+00
ORBITAL 0 4a'      4.5759
-0.43229086E+00-0.13130205E+00-0.18147217E+00 0.31431902E+00 0.45147483E+00
0.47975320E+00 0.47975320E+00
ORBITAL 0 5a'      6.7590
-0.77066074E-02-0.59602086E+00 0.10728635E+00-0.18582542E+00 0.64161589E+00
-0.30576735E+00-0.30576735E+00
ORBITAL 0 2a"      6.9443
-0.50250671E-11 0.69599173E-10 0.55291817E+00 0.31922745E+00-0.71362607E-10
-0.54423126E+00 0.54423126E+00
```

(b) NH_3_out.mgf

FIGURE 12 – Fichiers Entrée et Sortie MGF pour NH_3