Hartree-Fock 程序讲解

李睿

August 30, 2019

Contents

1 引言

> 如何让一个学生对计算化学充满恐惧？让他写 Hartree-Fock 程序。高斯积分手写的那种。
>
> ——————李睿

> 建设一流大学从 HF 做起。
>
> ——————王林军教授

Hartree-Fock 方法作为计算化学的开山鼻祖，对于一个想要从事计算化学相关工作的学生而言是一个必须要过去的坎。当然，原则上它作为一种工具你并不需要对它的所有细节都要彻底地把握，但如果并不深入了解，也可以说是十分令人叹息的事——Hartree-Fock 写出来确实算是小程序，但它仍然能称得上是人类智慧的结晶。

本文将着重探讨 Hartree-Fock 的各处细节如何在程序中得到体现，以及它们背后的理论基础。

网址是 https://github.com/Walter-Feng/Hartree-Fock. 不点赞说不过去呀！

2 程序流程概述

虽然 Hartree-Fock 程序相对而言较小，但由于需要照顾许多细节所以总体上仍旧会显得相对庞大，直接从程序源代码入手还并不容易窥见其全貌，因此在这里用文字做一个整体的流程介绍[1]。

流程概括地来讲分为：

1. 读取指定基组；
2. 从文件读取所有剩余的参数，包括所有原子的信息；
3. 将所有信息打包成为程序喜欢的数据结构；
4. 进行 Hartree-Fock 方法计算；
5. 将结果进行输出。

Hartree-Fock 方法采用的流程为：

1. 计算原子核排斥能；
2. 计算重叠矩阵（即 Overlap matrix，用 S 进行表示）；

[1] ~~在 LaTeX 里做流程图很累 de~~

3. 计算动能矩阵（即 Kinetic energy matrix, 用 T 进行表示）；

4. 计算原子核-电子相互作用势能矩阵（即 Nuclear attraction energy matrix, 用 Z 进行表示）；

5. 计算电子排斥能张量（为四阶张量，用 v 进行表示）；

6. 从单电子哈密顿矩阵（$h = T + Z$）得到初始系数矩阵（即 Coefficient matrix, 用 C 进行表示）；

7. 进行 RHF 迭代：

   (a) 从 C 以及电子数目得到电子密度矩阵（即 density matrix, 用 D 表示）；

   (b) 结合 h，D 与 v 得到 Fock 矩阵（用 F 进行表示）；

   (c) 结合 F, S 解本征值问题，得到对应特征值（分子轨道能量）及此时的 C；

   (d) 结合 h，D 与 v 得到对应体系能量；

   (e) 将得到的新的 C 与旧的 C 进行混合（mixing）并代入新一轮的迭代。

8. 判断每次循环的能量，最后进行输出。

3 数据结构讲解

本人认为，在对程序进行解析之前我们必须要了解其所采用的数据结构，因此在该章节我们先着重探讨本程序采用的所有数据结构。

让我们先了解本程序的核心数据结构 `orbital`，其代表了一个原子轨道，定义于 `basis.h` 中：

```
typedef struct orbital
{
    //Angular quantum number;
    int L;
    //magnetic quantum number;
    int m;
    //main quantum number;
    int n;
    //the name of the orbital;
    char label[20];

    //the angular momentum exponents combined with their coefficients (concerning
        the normalization for a linear combination of terms with different angular
        momentum exponents)
    struct angcoef{
        int a[3];
        double coef;
```

```
    }A[4];

    //This will tell the functions how long is the angcoef
    int length;

    //the total number of the terms of coefficients and exponents
    int total;

    //The list of exponents
    double * exponents;
    //The corresponding coefficient list, which will be normalized during the
        process;
    double * coefficients;

    //The cartesian coordinate of the center of the orbital;
    double cartesian[3];

    //pointer storing the next orbital
    orbital* NEXT;
}orbital;
```

其采用了链表的数据结构，用 NEXT 来储存下一个 orbital 的指针，从而实现任意长度的原子轨道数量的储存与调用[2]。在整个程序中磁量子数是通过各坐标的幂函数来体现的，如

$$d_{z^2} \to (x-x_0)^0(y-y_0)^0(z-z_0)^2.$$

其中 (x_0, y_0, z_0) 为原子坐标，而其指数 $(0, 0, 2)$ 便代表了磁量子数的信息，使用 (a_x, a_y, a_z) 来进行表示，而这是为了能够解析得到高斯积分结果的必要操作。考虑到如 $d_{x^2-y^2}$ 包含了两个轨道的线性叠加，我们使用了 angcoef 来描述这样的轨道，并使用 length 来进行长度的指定，方便读取。

由于在实际基组中一个电子轨道往往表示为多个高斯函数的叠加：

$$\psi(\boldsymbol{r}; \boldsymbol{r}_0, \boldsymbol{a}) \sim \sum_i c_i (x-x_0)^{a_x}(y-y_0)^{a_y}(z-z_0)^{a_z} e^{-\alpha_i(\boldsymbol{r}-\boldsymbol{r}_0)^2}. \tag{3.1}$$

我们对 $\{\alpha_i, c_i\}$ 分别通过 * exponents 和 * coefficients 进行储存，并使用 new/delete 来实现可调长度的高斯函数的储存。

当然如果把所有原子的电子轨道都打进这个链表的话查询起来会很麻烦，所以为了将不同原子的电子轨道分开，我们引入新的数据结构 atomic_orbital，同样定义于 basis.h 中：

```
typedef struct atomic_orbital
{
    // The atomic number of the atom;
    int N;
```

[2]不用 C++ 的 vector 类型的必然结果

```
    // The name of the atom;
    char name[5];

    //The cartesian coordinate of the atom;
    double cartesian[3];

    //The HEAD of the orbital
    orbital * orbital_HEAD;

    //Next atom
    atomic_orbital * NEXT;
}atomic_orbital;
```

其同样采用链表的形式，以适应基组含有不同数量的原子的现状。由于链表用起来不如 vector 来得好使，所以有一个函数 `orbital * orbital_enquiry(orbital * HEAD, int index)` 来帮你调用这个链表中第几个位置对应的指针，从而可以将其作为类似于向量的方式使用，但又能够保留自由增添的性质。[3]

而具体到每一步高斯积分的计算是针对每一个高斯函数的，如果直接使用 `orbital` 代入计算将非常让人头疼[4]，因此需要一个过渡的数据结构，将一个电子轨道拆散成高斯函数的线性组合，同时在这个转换的过程中完成高斯函数的归一化工作[5]，从而实现计算部分与信息存储部分的连接。而完成这一大任的是名字很俗的 `gaussian_chain`，定义于 `integral.h` 中：

```
typedef struct gaussian_chain{
    double R[3];
    int a[3];

    double exponent;

    double coefficient;

    gaussian_chain * NEXT;

}gaussian_chain;
```

~~因此如果你不懂链表你根本看不懂这个程序在干什么。~~

最后是储存电子排斥的四阶张量，这是由于在计算电子排斥的时候是双电子积分（four-index integral）。当然它具有着很多对称性，能够优化储存及计算速度，但为了能够更加直观地展现 Hartree-Fock 方法实现的原理，我们使用最原始的“方”四阶张量，以 `gsl_quad_tensor` 的形式储存，并定义于 `gslextra.h` 中：

```
typedef struct gsl_quad_tensor
```

[3]由于这个功能是比较偏后才引入的，所以仍旧会有相当多的部分仍旧使用原始的链表引用方式，看起来较为臃肿，请见谅。

[4]在这一点上和 `libint` 库不谋而合

[5]神奇吧，市面上的原子轨道基组是没有做归一化工作的。

```
{
    int i;
    int j;
    int k;
    int l;

    gsl_matrix *** element;
}gsl_quad_tensor;
```

其借用了 gsl (GNU Scientific Library) 中的矩阵储存形式 gsl_matrix 来简化[6]，并使用 i, j, k, l 指明各个边的长度，每个元素代表了一个双电子积分

$$v_{ijkl} = [ij|kl] = \int d\boldsymbol{r}_1 \int d\boldsymbol{r}_2 \, \phi_i(\boldsymbol{r}_1)\phi_j(\boldsymbol{r}_1)\frac{1}{r_{12}}\phi_k(\boldsymbol{r}_2)\phi_l(\boldsymbol{r}_2). \tag{3.2}$$

当然原则上由于 i 与 j，k 与 l，以及双方整体对调都是会得到一样的值，所以可以利用这样的对称性来节省计算时间，节省内存~~（但哪有时间写下来而且还增加阅读成本呢是吧）~~。

4 程序流程讲解

于是我们就开始剖析整个程序所经历的流程了。我会直接在该文档中复制源代码进行讲解，并附带背后的可能的数学原理。

4.1 读取基组

程序使用的基组套装是从 Basis Set Exchange [5](https://www.basissetexchange.org) 里面扒下来的，它会告诉你这个 shell 它会有多少个高斯函数，每个高斯函数的指数和系数分别是多少。但最反人类的是，Pople 基组（也就是 3-21G，6-31G 类型）的记录方式和非 Pople 基组的形式完全不一样！比如 3-21G 的碳的记录方式长这样：

```
BASIS "ao basis" PRINT
#BASIS SET: (6s,3p) -> [3s,2p]
C S
   0.1722560000E+03 0.6176690738E-01
   0.2591090000E+02 0.3587940429E+00
   0.5533350000E+01 0.7007130837E+00
C SP
   0.3664980000E+01 -0.3958951621E+00 0.2364599466E+00
   0.7705450000E+00 0.1215834356E+01 0.8606188057E+00
C SP
   0.1958570000E+00 0.1000000000E+01 0.1000000000E+01
END
```

[6]不然得五重指针呢你说是吧

但 cc-pVDZ 长这样：

```
C S
   6.665000E+03 6.920000E-04 -1.460000E-04 0.000000E+00
   1.000000E+03 5.329000E-03 -1.154000E-03 0.000000E+00
   2.280000E+02 2.707700E-02 -5.725000E-03 0.000000E+00
   6.471000E+01 1.017180E-01 -2.331200E-02 0.000000E+00
   2.106000E+01 2.747400E-01 -6.395500E-02 0.000000E+00
   7.495000E+00 4.485640E-01 -1.499810E-01 0.000000E+00
   2.797000E+00 2.850740E-01 -1.272620E-01 0.000000E+00
   5.215000E-01 1.520400E-02 5.445290E-01 0.000000E+00
   1.596000E-01 -3.191000E-03 5.804960E-01 1.000000E+00
C P
   9.439000E+00 3.810900E-02 0.000000E+00
   2.002000E+00 2.094800E-01 0.000000E+00
   5.456000E-01 5.085570E-01 0.000000E+00
   1.517000E-01 4.688420E-01 1.000000E+00
C D
   5.500000E-01 1.0000000
```

也就是 Pople 基组会直接合并同一层的 S 轨道和 P 轨道，并且是按照每一层来记录，但非 Pople 基组是按照角动量来分类。~~这谁顶得住啊！~~所以在本程序中使用 *Mathematica* 调整为规整的样子，方便读取，比如：

```
N: 2
Name: He

L= 0
total= 4
Exponents:
38.3600000 5.7700000 1.2400000 0.2976000
Coefficients:
0.0238090 0.1548910 0.4699870 0.5130270

L= 0
total= 4
Exponents:
38.3600000 5.7700000 1.2400000 0.2976000
Coefficients:
0.0000000 0.0000000 0.0000000 1.0000000

L= 1
total= 1
Exponents:
1.2750000
Coefficients:
1.0000000
```

按照角动量分类，并且每一个 shell 都会有关键词给你注释，这样既提供可读性也可以在程序中提供 flag，两全其美！但还是要提醒一句，**这个基组并不是归一的，更不是正交的**。

读取该部分基组的函数是 `basis_fscanf(FILE * basis,atomic_orbital * HEAD)`：

由于源代码可能太长，在这里做简单剖析:

```
if(strcmp(str,"N:")==0)
{
    // Check whether this is a HEAD, namely that this is the initial part
    if(HEADFLAG == 0)
    {
        temp1 = HEAD;
        HEADFLAG = 1;
    }

    // if it is not, create new knot and have this atom linked with the previous
        atom, and seal the former atom
    else
    {
        temp2 = temp1;
        temp1 = atomic_orbital_calloc();
        temp2->NEXT = temp1;
    }

    //initialize
    n_counter = 0;
    l_temp = -1;
    l_ref = -1;
    ORBITHEADFLAG = 0;
    ALL_ORBITHEADFLAG = 0;

    //scan the atomic number
    fscanf(basis,"%d",&temp1->N);
}
```

这一部分是判断是不是现在要读一个新的原子。由于有可能这是基组的第一个原子，这个时候你并不需要将前一个原子对应的 `atomic_orbital` 的指针中的 `NEXT` 与新原子的指针对应起来，所以需要用一个 `HEADFLAG` 来进行区分，同时每一个新原子都要进行初始化，并读取原子序数。

```
            if(strcmp(str,"Name:")==0)
            {
                fscanf(basis,"%s",str);
                strcpy(temp1->name,str);
            }
```

Easy. 过。

然后是读取每一个电子轨道了，它们都由 L= ? 来发起，所以用这个做关键词：

```
//Read L: when there is a new L, there is a new electron shell that needs to be
    interpreted, with a set of exponents and coefficients
if(strcmp(str,"L=")==0)
{
    fscanf(basis,"%d",&l_temp);

    //Check if all_orbit has been assigned (or whether all_orbit will be the head of
        a new atom -> which means it will certainly be the tail of the linked list)
       , and ensure that all_orbit is the tail of the list
    if(ALL_ORBITHEADFLAG == 1)
       while(all_orbit->NEXT != NULL)
          all_orbit = all_orbit->NEXT;


    //check whether it has a change compared with the former l_temp, so to restart
        counting the main magnetic number n
    if(l_temp == l_ref) n_counter ++;
    else n_counter = l_temp + 1;

    //set l_ref to the current value of l_temp for next loop
    l_ref = l_temp;

    //Read total
    fscanf(basis,"%s",str);
    fscanf(basis,"%d",&tot_temp);

    //Read exponents
    fscanf(basis,"%s",str);

    exponents_temp = new double[tot_temp];

    for(i=0;i<tot_temp;i++)
       fscanf(basis,"%lf",exponents_temp + i);

    //Read coefficients
    fscanf(basis,"%s",str);

    coefficients_temp = new double[tot_temp];

    for(i=0;i<tot_temp;i++)
       fscanf(basis,"%lf",coefficients_temp + i);

    ...
```

可以很容易地看出它在一层一层地提取信息。不过对于一个原子的电子轨道也会存在是不是 HEAD 的问题，所以需要 all_orbit 来规避。

但需要明白的是，这里面只有角量子数的信息，但轨道是要细化到磁量子数的，所以需要将其在磁量子数上进行循环然后添加轨道：

```
for(m_temp=-l_temp;m_temp<=l_temp;m_temp++)
{
    orbit_temp1 = orbital_calloc(tot_temp);
    //Check if orbit_temp1 should be the head of the linked list of orbitals in the
        same atom. If not, link it with the previous orbital whose pointer is stored
         in orbit_temp2
    if(ORBITHEADFLAG == 1) //It's not the head
    {
        orbit_temp2->NEXT = orbit_temp1;
    }
    else //It's the head, and thus orbit_temp2 is not assigned
    {
        ORBITHEADFLAG = 1;

        //It might be that all_orbit, containing all the orbitals of the atom the
            program is currently interpreting, has not assigned, in which case
            all_orbit will be the head of the linked list of orbitals
        // P.S.: It is only possible that all_orbit can be the head when orbit_temp1
            is the head
        if(ALL_ORBITHEADFLAG == 0) //It's the head
        {
            all_orbit = orbit_temp1;
            ALL_ORBITHEADFLAG = 1;
            temp1->orbital_HEAD = all_orbit;
        }
        else
        {
            all_orbit->NEXT = orbit_temp1;
        }
    }

    //Copying the information
    orbit_temp1->L = l_temp;
    orbit_temp1->total = tot_temp;
    orbit_temp1->n = n_counter;
    orbit_temp1->m = m_temp;

    orbital_label(orbit_temp1->label,n_counter,l_temp,m_temp);
    orbital_angcoef_set(orbit_temp1);

    for(i=0;i<tot_temp;i++)
    {
        *(orbit_temp1->exponents + i) = *(exponents_temp + i);
        *(orbit_temp1->coefficients + i) = *(coefficients_temp + i);
    }
```

```
    orbit_temp2 = orbit_temp1;
}
```

然后顺带将主量子数、角量子数等信息拷贝进去。值得注意的是这里面引用了两个函数 `orbital_label(orbit_temp1->label,n_counter,l_temp,m_temp)` 和 `orbital_angcoef_set(orbit_temp1)`，前一个是针对主量子数和角量子数 n, l, m 写出对应名字（如 $2p_z$），而后一个是需要将对应系数填入 `angcoef` 里（比如 $3d_{x^2-y^2}$ 的情形）。但它们只能做成分支的情形，所以在这里就不进行讲述了。

通过这个模块基组便以 `atomic_orbital` 的形式储存了起来方便调用。而对于一个具体的体系，我们需要从里面提取出相应原子，该部分体现在 `main.cpp` 读取 input 文件的时候：

```
if(strcmp(reader,"&COORD")==0)
{
    // allocate -> set the head of the linked list
    atoms = atomic_orbital_calloc();

    // saving the location of the head
    atoms_bk = atoms;

    // read atoms
    while(fscanf(inputfile,"%s",reader)!=EOF)
    {
        if(strcmp(reader,"&END_COORD")==0)
        {
            atomic_orbital_free(atoms);
            atoms = atoms_bk;
            atoms_temp->NEXT = NULL;
            break;
        }

        else
        {
            // scanner for basis
            basis_scanner = basis_HEAD;

            // scan the basis
            while(basis_scanner->NEXT!=NULL)
            {
                // whether the name matches
                if(strcmp(basis_scanner->name,reader)==0)
                {
                    atomic_orbital_single_cpy(atoms,basis_scanner);

                    break;
                }
```

```
            basis_scanner = basis_scanner->NEXT;
        }

        if(strcmp(basis_scanner->name,reader)==0)
        {
            // copy information from the basis
            atomic_orbital_single_cpy(atoms,basis_scanner);
            // add electrons
            el_num += basis_scanner->N;
            // copy coordinates & convert unit
            for(i=0;i<3;i++)
            {
                fscanf(inputfile,"%lf",&coord_temp);
                atoms->cartesian[i] = coord_temp / 0.529177210903;
            }

            // copy the coordinates of the atom to all of its orbitals
            atomic_orbital_sync_coord(atoms);
            // print the name of the atom to all its orbitals
            atomic_orbital_name_print(atoms);

            atoms->NEXT = atomic_orbital_calloc();

            atoms_temp = atoms;

            atoms = atoms->NEXT;
        }

        // no such an atom in the basis -> throw error
        else
        {
            printf("HF_ERROR: atom %s is not defined in the basis.\n",reader);

            return 3;
        }


    }
  }
}
```

可以看到在读取 `&COORD` 和 `&END_COORD` 之间的每一行原子信息都是通过从基组查询名字是不是存在，然后把信息复制出来，把坐标提取出来并转换成原子单位制，并形成一个完整的链表。

当然，对于 Hartree-Fock 而言它还需要一个电子轨道的链表，所以在 `main.cpp`

中把所有涉及的电子轨道重新拿出来形成一个链表：

```
// setting the head for the orbitals
orbitals = orbital_calloc((atoms->orbital_HEAD)->total);

// save the location of the head
orbital_temp = orbitals;
// set the scanner of the atoms' list to its head
atoms_temp = atoms;

// copy all the orbitals from all atoms into `orbitals`
while(atoms_temp->NEXT != NULL)
{
    orbital_cpy(orbital_temp,atoms_temp->orbital_HEAD);
    while(orbital_temp->NEXT != NULL)
        orbital_temp = orbital_temp->NEXT;

    atoms_temp = atoms_temp->NEXT;
    orbital_temp->NEXT = orbital_calloc((atoms_temp->orbital_HEAD)->total);

    orbital_temp = orbital_temp->NEXT;
}
// DO the copying for the last one atom
orbital_cpy(orbital_temp,atoms_temp->orbital_HEAD);

// count the orbitals
length = orbital_count(orbitals);
```

然后通过 `orbital_count` 数有多少电子轨道，方便后续对各类矩阵的操作（包括调用内存，存取信息等）。

4.2 高斯积分

最让人头疼的来了！

我们为什么要用基组？因为在量子力学里所有的物理量都是通过算符来表示，但计算机不知道什么叫算符，所以我们需要通过基组来将其转换成一个个矩阵，将问题转换为线性代数问题。DVR(Discrete Variable Representation)[7]如此，电子结构问题亦如此。但每一个矩阵元意味着一个内积，而内积并不是简单的笛卡尔内积，而是一个全空间积分！这也就是最让人头疼的地方，也是高斯基组最美妙的地方。

我们需要解决的是这么一个线性代数（特征值）问题，

$$FC = SCE \tag{4.1}$$

其中 F 和 S 是我们需要实现求出来的。我们从最简单的 S 开始。

[7]同样可查看源代码于 https://github.com/Walter-Feng/SincDVR.git.

4.2.1 重叠积分

定义是这个：

$$S_{ij} = \langle \phi_i | \phi_j \rangle = \int \phi_i(\boldsymbol{r}) \phi_j(\boldsymbol{r}) d\boldsymbol{r}. \tag{4.2}$$

所幸 ϕ_i 和 ϕ_j 都是高斯函数（或者乘上一个多项式），而这样的形式是能够给出精确解的，比如下面这个公式：

$$\int_{\mathrm{R^n}} dX\, f(X)\, e^{-\frac{1}{2}X^T A X + B^T X} = \sqrt{\frac{(2\pi)^n}{\det A}} e^{\frac{1}{2}B^T A^{-1} B} \exp\left(\frac{1}{2}(A^{-1})_{ij} \frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}\right) f(X)\Big|_{X=B} \tag{4.3}$$

它表达了对于包括任意数量的自变量 X 以及一个函数（最好是多项式），给出高斯函数的二次项矩阵 A 和一次项矢量 B 的情况下得到的精确解。其中 exp 部分应理解为

$$\exp\{\hat{g}\} = \sum_n \frac{\hat{g}^n}{n!} \tag{4.4}$$

当然这么写程序会非常难（我也只能在擅长符号计算的 *Mathematica* 里实现它）。所幸电子轨道的样子是非常有规律的，所以我们可以通过 Obara & Saika 提示的迭代关系来实现。

我们先展现所使用的符号。对于一个高斯函数，

$$|a) \equiv g(\vec{r}, \alpha, \boldsymbol{a}, \vec{A}) = (x - A_x)^{a_x} (y - A_y)^{a_y} (z - A_z)^{a_z} \exp\left(-\alpha |\vec{r} - \vec{A}|^2\right) \tag{4.5}$$

有一个 Gaussian Product Theorem, 即 [3]：

Theorem 4.1. 两个高斯函数的乘积可转换为一系列高斯函数的和，比如：

$$g(\vec{r}, \alpha, \mathbf{0}, \vec{A}) g(\vec{r}, \beta, \mathbf{0}, \vec{B}) = \exp\left(-\xi |\overline{AB}|^2\right) g(\vec{r}, \zeta, \mathbf{0}, \vec{P}), \tag{4.6}$$

其中 $\zeta = \alpha + \beta$, $\xi = \frac{\alpha\beta}{\zeta}$, $\vec{P} = \frac{\alpha\vec{A} + \beta\vec{B}}{\zeta}$, $\overline{AB} = \vec{A} - \vec{B}$. 对于含有多项式的高斯函数。可以利用

$$(x - A_x)^{a_x} = [(x - P_x) + \overline{PA}_x]^{a_x} = \sum_{i=0}^{a_x} C_{a_x}^i (x - P_x)^i P A_x^{a_x - i} \tag{4.7}$$

从而可以展开两个高斯函数的乘积为

$$g_1 g_2 = \exp\left(-\xi |AB|^2\right) \exp\left(-\zeta |\vec{r} - \vec{P}|^2\right) \times \prod_{k=x,y,z} \sum_{i=0}^{i=a_k+b_k} (k - P_k)^i f_i(a_k, b_k, \overline{PA}_k, \overline{PB}_k), \tag{4.8}$$

其中

$$f_i(a_k, b_k, \overline{PA}_k, \overline{PB}_k) = \sum_{n=0}^{n=i} C_{a_k}^n C_{b_k}^{i-n} \overline{PA}_k^{a_x - n} \overline{PB}_k^{b_x - n + k} \tag{4.9}$$

但这不迭代啊！

高斯函数还有一个特殊的性质，我们可以通过对它进行求导来得到一个多项式的组合：

$$\partial_{A_i} g(\vec{r},\alpha,\boldsymbol{a},\vec{A}) = 2\alpha g(\vec{r},\alpha,\boldsymbol{a}+\mathbf{1}_i,\vec{A}) - a_i g(\vec{r},\alpha,\boldsymbol{a}-\mathbf{1}_i,\vec{A}) \tag{4.10}$$

其中 $\mathbf{1}_i = (\delta_{ix},\delta_{iy},\delta_{iz})$, 即沿着某一方向的单位矢量。从而可以有

$$(\boldsymbol{a}+\mathbf{1}_i|\boldsymbol{b}) = \frac{1}{2\alpha}\partial_{A_i}(\boldsymbol{a}|\boldsymbol{b}) + \frac{a_i}{2\alpha}(\boldsymbol{a}-\mathbf{1}_i|\boldsymbol{b}) \tag{4.11}$$

而 ∂_{A_i} 可以通过前面所描述的 Gaussian Product Theorem 展开后观察得到：

$$(\boldsymbol{a}+\mathbf{1}_i|\boldsymbol{b}) = (P_i - A_i)(\boldsymbol{a}|\boldsymbol{b}) + \frac{b_i}{2\zeta}(\boldsymbol{a}|\boldsymbol{b}-\mathbf{1}_i) + \frac{a_i}{2\alpha}(\boldsymbol{a}-\mathbf{1}_i|\boldsymbol{b}) \tag{4.12}$$

而起始条件可以由高斯函数在全空间的积分中非常方便地得到：

$$(\mathbf{0}_A|\mathbf{0}_B) = \left(\frac{\pi}{\zeta}\right)^{3/2} \exp\left(-\xi|\overline{AB}|^2\right), \tag{4.13}$$

从而建立起了完整的迭代关系。该部分的迭代的核心部分由 `SIntegral` 实现，并通过 `double gaussian_chain_SIntegral(gaussian_chain * a, gaussian_chain * b)` 允许将 `gaussian_chain` 作为执行对象：

```
double gaussian_chain_SIntegral(gaussian_chain * a, gaussian_chain
    * b)
{
    return a->coefficient * b->coefficient *SIntegral(a->R,b->R,a->
        a[0],a->a[1],a->a[2],b->a[0],b->a[1],b->a[2],a->exponent,b
        ->exponent);
}
```

而 `void single_electron_transform(gaussian_chain * HEAD, orbital * a)` 可以将 `orbital` 数据结构转换为一系列的 `gaussian_chain` 并完成归一化：

```
//transform the struct orbital to struct gaussian_chain
void single_electron_transform(gaussian_chain * HEAD, orbital * a)
{
    gaussian_chain * temp, * bk;

    temp = HEAD;

    int i,j,q;

    for(i=0;i<a->length;i++)
    {
        for(j=0;j<a->total;j++)
        {
            temp->coefficient = a->A[i].coef * *(a->coefficients + j) * normalize(*(a
                ->exponents + j),a->A[i].a[0],a->A[i].a[1],a->A[i].a[2]);
```

```
            temp->exponent = *(a->exponents + j);
            for(q=0;q<3;q++)
            {
                temp->R[q] = a->cartesian[q];
                temp->a[q] = a->A[i].a[q];
            }
            temp->NEXT = gaussian_chain_calloc();
            bk = temp;
            temp = temp->NEXT;
        }
    }
    delete temp;
    bk->NEXT = NULL;
}
```

其中归一化所用的函数为

```
double normalize(double alpha, int ax, int ay, int az)
{
    return pow(2 * alpha / M_PI, 0.75) * pow(4 * alpha, (double) (ax + ay + az) /
        2.0) / sqrt(double_factorial(2 * ax - 1) * double_factorial(2 * ay - 1) *
        double_factorial(2 * az - 1));
}
```

即 [4]

$$N(\alpha, a_x, a_y, a_z) = \left(\frac{2\alpha}{\pi}\right)^{3/4} \frac{(4\alpha)^{(a_x+a_y+a_z)/2}}{[(2a_x-1)!!(2a_y-1)!!(2a_z-1)!!]^{1/2}}. \tag{4.14}$$

4.2.2 动能积分

其实前面已经大概说明了，对于高斯函数而言求导并不算什么大问题——它很快就能拆开成两个不同的高斯函数的加和，那么我们只要在求导后对每一个高斯函数进行一次重叠积分，我们就能够知道动能的期望值。比如对于 x 方向的求导，

```
// obtain the derivative of a gaussian function chain and store as another gaussian
    function chain over certain axis.
// key: 0->x 1->y 2->z
void gaussian_chain_derivative(gaussian_chain * dest, gaussian_chain * src, int key)
{
    gaussian_chain * temp1, * temp2;

    int i;

    temp1 = dest;
    temp2 = src;

    while(temp2->NEXT != NULL)
```

```
{
    if(temp2->a[key] == 0)
    {
        for(i=0;i<3;i++)
        {
            temp1->a[i] = temp2->a[i];
            temp1->R[i] = temp2->R[i];
        }
        temp1->a[key] = temp2->a[key] + 1;
        temp1->coefficient = - temp2->coefficient * 2.0 * temp2->exponent;
        temp1->exponent = temp2->exponent;
        temp1->NEXT = gaussian_chain_calloc();
        temp1 = temp1->NEXT;
        temp2 = temp2->NEXT;
    }
    else
    {
        for(i=0;i<3;i++)
        {
            temp1->a[i] = temp2->a[i];
            temp1->R[i] = temp2->R[i];
        }
        temp1->a[key] = temp2->a[key] + 1;
        temp1->coefficient = - temp2->coefficient * 2.0 * temp2->exponent;
        temp1->exponent = temp2->exponent;
        temp1->NEXT = gaussian_chain_calloc();
        temp1 = temp1->NEXT;
        for(i=0;i<3;i++)
        {
            temp1->a[i] = temp2->a[i];
            temp1->R[i] = temp2->R[i];
        }
        temp1->coefficient = (double) temp2->a[key] * temp2->coefficient;
        temp1->exponent = temp2->exponent;
        temp1->a[key] = temp2->a[key] - 1;
        temp1->NEXT = gaussian_chain_calloc();
        temp1 = temp1->NEXT;
        temp2 = temp2->NEXT;
    }

}
...
```

在这里面我对它在 x 方向上是不是没有角动量（即 $a_x = 0$）进行了判断，然后按照各自的情况进行展开，这并没有什么难度。对于动能是要求各方向的二次求导，也不难：

```
// obtain the second derivative of a gaussian function chain and store as another
    gaussian function chain over certain axis.
void gaussian_chain_second_derivative(gaussian_chain * dest, gaussian_chain * src,
    int key)
{
    gaussian_chain * temp1, * temp2, * temp3;

    temp1 = dest;
    temp2 = src;
    temp3 = gaussian_chain_calloc();

    gaussian_chain_derivative(temp3,temp2,key);
    gaussian_chain_derivative(temp1,temp3,key);

    gaussian_chain_free(temp3);
}

void gaussian_chain_laplacian(gaussian_chain * dest, gaussian_chain * src)
{
    int i;
    gaussian_chain * temp1, * temp2;

    temp1 = dest;
    for(i=0;i<3;i++)
    {
        gaussian_chain_second_derivative(temp1,src,i);
        while(temp1->NEXT != NULL)
            temp1 = temp1->NEXT;
        temp1->NEXT = gaussian_chain_calloc();
        temp2 = temp1;
        temp1 = temp1->NEXT;
    }
    delete temp1;
    temp2->NEXT = NULL;
}
// obtain the expectation value of kinetic energy operator, T = p^2/ 2m = - \nabla^2
     /2, for gaussian_chain struct
double gaussian_chain_kinetic_energy(gaussian_chain * a_HEAD, gaussian_chain *
    b_HEAD)
{
    double result;
    gaussian_chain * laplacian_temp;

    laplacian_temp = gaussian_chain_calloc();

    gaussian_chain_laplacian(laplacian_temp, b_HEAD);
```

```
    result = - 0.5 * gaussian_chain_full_SIntegral(a_HEAD,laplacian_temp);

    gaussian_chain_free(laplacian_temp);

    return result;
}
```

也就是先抓过来求一次导，然后换个地方再求一次导，就得到了一个方向的二阶导；把每一个方向的串联起来[8]，然后和原来的高斯函数系列做个重叠积分，乘上系数 $-\frac{1}{2}$ 便能得到最后的动能期望值。

4.2.3 核吸引积分

啊对就是 Nuclear attraction energy integral[9].

定义是

$$Z_{ij} = \langle \phi_i | \frac{1}{|\boldsymbol{r} - \boldsymbol{r}_l|} | \phi_j \rangle = \int \frac{\phi_i(\boldsymbol{r}) \phi_j(\boldsymbol{r})}{|\boldsymbol{r} - \boldsymbol{r}_l|} d\boldsymbol{r} \tag{4.15}$$

其中 $\boldsymbol{r}_l$ 为原子核所在的坐标。

解决这个问题最强的地方在于——把 $\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}$ 转换为高斯函数！

$$|\boldsymbol{r}_1 - \boldsymbol{r}_2|^{-1} = \frac{2}{\pi^{1/2}} \int_0^\infty du \exp\left[-\left(\boldsymbol{r}_1 - \boldsymbol{r}_2\right)^2 u^2 \right]. \tag{4.16}$$

而

$$\exp\left[-\left(\boldsymbol{r} - \boldsymbol{r}_l\right)^2 u^2 \right] = g(\boldsymbol{r}; u^2, \boldsymbol{0}, \boldsymbol{r}_l) \tag{4.17}$$

于是就变成了三中心重叠积分：

$$\langle \phi_i | \frac{1}{|\boldsymbol{r} - \boldsymbol{r}_l|} | \phi_j \rangle = \frac{2}{\pi^{1/2}} \int_0^\infty du \left(\phi_i | \boldsymbol{0}_{\boldsymbol{r}_2}^{u^2} | \phi_j \right) \tag{4.18}$$

类似于二电子重叠积分，我们对三中心重叠积分也有迭代关系：

$$\begin{aligned} (\boldsymbol{a} + \boldsymbol{1}_i | \boldsymbol{c} | \boldsymbol{b}) = & \left(G_i - A_i\right) (\boldsymbol{a} | \boldsymbol{c} | \boldsymbol{b}) + \frac{a_i}{2\left(\zeta + \zeta_c\right)} \\ & \times (\boldsymbol{a} - \boldsymbol{1}_i | \boldsymbol{c} | \boldsymbol{b}) + \frac{b_i}{2\left(\zeta + \zeta_c\right)} (\boldsymbol{a} | \boldsymbol{c} | \boldsymbol{b} - \boldsymbol{1}_i) \\ & + \frac{c_i}{2\left(\zeta + \zeta_c\right)} (\boldsymbol{a} \left| \boldsymbol{c} - \boldsymbol{1}_i \right| \boldsymbol{b}) \end{aligned} \tag{4.19}$$

其中 $\boldsymbol{G} = \frac{\zeta \boldsymbol{P} + \zeta_c \boldsymbol{C}}{\zeta + \zeta_c}$. 然后由于在这里 c 没有角动量，所以最后一项舍去，$\zeta_c = u^2$，从而有

$$\begin{aligned} \left(\boldsymbol{a} + \boldsymbol{1}_i | \boldsymbol{0}_{\boldsymbol{r}_l}^{u^2} | \boldsymbol{b} \right) = & \left(G_i - A_i\right) (\boldsymbol{a} | \boldsymbol{0}_{\boldsymbol{r}_l}^{u^2} | \boldsymbol{b}) + \frac{a_i}{2\left(\zeta + u^2\right)} \\ & \times \left(\boldsymbol{a} - \boldsymbol{1}_i | \boldsymbol{0}_{\boldsymbol{r}_l}^{u^2} | \boldsymbol{b} \right) + \frac{b_i}{2\left(\zeta + u^2\right)} \left(\boldsymbol{a} | \boldsymbol{0}_{\boldsymbol{r}_l}^{u^2} | \boldsymbol{b} - \boldsymbol{1}_i \right) \end{aligned} \tag{4.20}$$

[8] 体现了为数不多的链表的优越性所在

[9] ~~我只看过英文的资料我怎么知道中文叫什么嘛~~

$G_i - A_i$ 是

$$G_i - A_i = \frac{\beta(B_i - A_i) + u^2(C_i - A_i)}{\zeta + u^2}, \quad \boldsymbol{C} = \boldsymbol{r}_l \tag{4.21}$$

也就是说最后在积分的时候 $\frac{u^2}{\zeta+u^2}$ 和 $\frac{u^2}{\zeta+u^2}$ 会影响最后的积分。这怎么玩嘛!

请见数值积分方法，本章节完（不是）

有变换

$$\frac{1}{\zeta + u^2} = \frac{1}{\zeta}\left(1 - \frac{u^2}{\zeta + u^2}\right) \tag{4.22}$$

这样至少能把 $\frac{u^2}{\zeta+u^2}$ 都转换成 $\frac{u^2}{\zeta+u^2}$. 这也就意味着最后的结果最多也就乘上 $(\frac{u^2}{\zeta+u^2})^m$ 这一项。我们不妨定义

$$(\boldsymbol{a}|\frac{1}{|\boldsymbol{r} - \boldsymbol{r}_l|}|\boldsymbol{b})^{(m)} = \frac{2}{\pi^{1/2}}\int_0^\infty \left(\frac{u^2}{\zeta + u^2}\right)^m du\,(\boldsymbol{a}|\mathbf{0}_{\boldsymbol{r}_2}^{u^2}|\boldsymbol{b}) \tag{4.23}$$

那么 $m = 0$ 就是我们本来想要的。$(\boldsymbol{a}|\mathbf{0}_{\boldsymbol{r}_2}^{u^2}|\boldsymbol{b})$ 这个东西可以通过迭代关系最终化简到 $(\mathbf{0}_A|\mathbf{0}_C|\mathbf{0}_B)$ 类型, 然后利用高斯积分计算可以得到

$$(\mathbf{0}_A\,|\mathbf{0}_C|\,\mathbf{0}_B) = \left(\frac{\pi}{\zeta_a + \zeta_b + \zeta_c}\right)^{3/2} \kappa_{abc}, \tag{4.24}$$

其中

$$\kappa_{abc} = \exp\left[-\xi(\boldsymbol{A} - \boldsymbol{B})^2\right] \exp\left[-\frac{\zeta\zeta_c}{\zeta + \zeta_c}(\boldsymbol{P} - \boldsymbol{C})^2\right]. \tag{4.25}$$

对 $\left(\frac{\pi}{\zeta_a+\zeta_b+\zeta_c}\right)^{3/2}$ 我们可以使用刚才的方法继续转换成 $\frac{1}{\zeta+u^2}$ 类型。这样我们就能够将积分转换成类似于

$$\int_0^\infty \left(\frac{u^2}{\zeta + u^2}\right)^{(m+\frac{3}{2})} \exp\left[-\frac{u^2}{\zeta + u^2}\zeta(\boldsymbol{P} - \boldsymbol{r}_2)^2\right] du.$$

看来只能用换元法呀！令

$$t^2 = \frac{u^2}{\zeta + u^2}. \tag{4.26}$$

那么

$$\frac{1}{t^2} = 1 + \frac{\zeta}{u^2}. \tag{4.27}$$

也就是

$$u = \sqrt{\frac{\zeta t^2}{1 - t^2}}. \tag{4.28}$$

省去求微分环节，就可以把上面的积分式转换得到

$$\int_0^1 t^{2m} \exp\left[-\zeta(\boldsymbol{P} - \boldsymbol{r}_2)^2 t^2\right] dt.$$

这怎么积啊!

请见数值积分方法，本章节完（不是）

谷歌一查有惊喜，这个就是 Boys function. 也可以直接放进 *Mathematica*，得到

$$\int_0^1 t^{2m}\exp\left(-Tt^2\right)dt=\frac{1}{2}T^{-\frac{2m+1}{2}}\left[\Gamma\left(\frac{2m+1}{2}\right)-\Gamma\left(\frac{2m+1}{2},T\right)\right], \tag{4.29}$$

其中 $\Gamma(n,x)$ 是 incomplete Gamma function(定义详见 wiki).

将所有结果进行汇总，就能够得到

$$\begin{aligned}(\boldsymbol{a}+\mathbf{1}_i|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b})^{(m)}=&\frac{\beta}{\zeta}\overline{BA}_i(\boldsymbol{a}|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b})^{(m)}\\&+\left(C_i-A_i-\frac{\beta}{\zeta}\overline{BA}_i\right)(\boldsymbol{a}|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b})^{(m+1)}\\&+\frac{a_i}{2\zeta}\left[(\boldsymbol{a}-\mathbf{1}_i|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b})^{(m)}-(\boldsymbol{a}-\mathbf{1}_i|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b})^{(m+1)}\right]\\&+\frac{b_i}{2\zeta}\left[(\boldsymbol{a}|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b}-\mathbf{1}_i)^{(m)}-(\boldsymbol{a}|\frac{1}{|\boldsymbol{r}-\boldsymbol{r}_l|}|\boldsymbol{b}-\mathbf{1}_i)^{(m+1)}\right]\end{aligned} \tag{4.30}$$

以及初始条件

$$\frac{2\pi}{\zeta}\exp\left(-\xi\overline{AB}^2\right)F_m(T), \tag{4.31}$$

其中

$$T=\zeta\overline{PC}^2, \tag{4.32}$$

以及

$$F_m(T)=\int_0^1 t^{2m}\exp\left\{-Tt^2\right\}dt. \tag{4.33}$$

在程序中体现为一系列的 `ZIntegral` 函数。

4.2.4 双电子积分/库仑积分

定义为

$$J_{ij}=\langle\phi_i|\frac{1}{|\boldsymbol{r}_1-\boldsymbol{r}_2|}|\phi_j\rangle=\int\frac{\phi_i(\boldsymbol{r})\phi_j(\boldsymbol{r})}{|\boldsymbol{r}_1-\boldsymbol{r}_2|}d\boldsymbol{r}. \tag{4.34}$$

所以其实方法和 Nuclear attraction energy integral 类似，把中间的距离的反比转换成高斯函数，然后变成迭代关系。

$$\begin{aligned}(\boldsymbol{a}+\mathbf{1}_i|\frac{1}{r_{12}}|\boldsymbol{b})=&\overline{PA}_i(\boldsymbol{a}|\frac{1}{r_{12}}|\boldsymbol{b})^{(m+1)}\\&+\frac{a_i}{2\alpha}\left\{(\boldsymbol{a}-\mathbf{1}_i|\frac{1}{r_{12}}|\boldsymbol{b})^{(m)}-\frac{\beta}{\zeta}(\boldsymbol{a}-\mathbf{1}_i|\frac{1}{r_{12}}|\boldsymbol{b})^{(m+1)}\right\}\\&+\frac{b_i}{2\zeta}(\boldsymbol{a}|\frac{1}{r_{12}}|\boldsymbol{b}-\mathbf{1}_i)^{(m+1)}\end{aligned} \tag{4.35}$$

$$(\mathbf{0}_A|\frac{1}{r_{12}}|\mathbf{0}_B)^{(m)}=\frac{2\pi^{\frac{5}{2}}}{\alpha\beta\zeta^{\frac{1}{2}}}F_m(T), \tag{4.36}$$

其中

$$T = \xi |\overline{AB}|^2, \tag{4.37}$$

以及

$$F_m(T) = \int_0^1 t^{2m} \exp\left\{-Tt^2\right\} \, dt. \tag{4.38}$$

在程序中体现为一系列的 `JIntegral` 函数。

值得一提的是在体系中应该是以双电子积分的形式存在，不过利用 Gaussian Product Theorem 可以把两边的高斯函数乘积转化成为一系列高斯函数之和，然后问题就简单了。在程序中以 `two_electron_transform` 的形式体现。

4.3 Hartree-Fock 迭代

终于到了最为核心的 Hartree-Fock 部分！在解决了所有电子积分相关问题以后后面就是纯粹的线性代数问题了。

4.3.1 非正交基组下的特征解问题——Löwdin 方法

如果你看过我和我的学长——张超群学长和顾锴学长——合著的一本指南——《浙江大学王林军课题组本科生入门指南》(https://github.com/Warlocat/A-Quick-Tour-Guide-for-LinJun-Group.git)，你应该还记得我们为什么要讨论这个问题。这是由于我们一直都使用的是原子轨道基组，而原子轨道基组并不是正交的[10]，所以我们需要研究非正交基组如何求得特征解。

~~等一下我去那边拷贝一下……~~

~~来子来子~~

我们先定义原子轨道基组 $\{\phi_i\}$，他们之间不满足正交归一性，而是存在一个重叠积分矩阵

$$S_{ij} \equiv \langle \phi_i | \phi_j \rangle = \int \mathrm{d}\boldsymbol{x} \phi_i^*(\boldsymbol{x}) \phi_j(\boldsymbol{x}) \tag{4.39}$$

Fock 算符的本征态可以在这个基下展开 (我们先忽略自旋的问题，或者认为上述原子轨道其实是自旋轨道)，

$$|\psi_i\rangle = \sum_j C_{ji} |\phi_i\rangle \tag{4.40}$$

这样，系数矩阵 C_{ji} 的第 i 列表示第 i 个分子轨道对原子轨道的展开系数，这与多数软件中的设定是一致的。原本的 Fock 算符的本征值问题可以写作

$$\hat{f} |\psi_j\rangle = \varepsilon_j |\psi_j\rangle \tag{4.41}$$

[10] 当然如果您量子力学学得很好您应该记得比如一个氢原子的所有轨道都是正交归一的——但我们研究的是多原子体系，所以如果一个地方的氢原子的 $1s$ 轨道和另一个地方的氢原子放在那里……那自然就不正交啦

左乘一个 $\langle\psi_i|$ 然后将原子轨道展开代入

$$\sum_k \sum_l C_{ki}^* C_{lj} \langle\phi_k|\hat{f}|\phi_l\rangle = \sum_k \sum_l C_{ki}^* C_{lj} \langle\phi_k|\phi_l\rangle \varepsilon_j \tag{4.42}$$

我们定义原子轨道基组下的 Fock 算符的矩阵为 $F_{kl} \equiv \phi_k|\hat{f}|\phi_l$，在很多文献中，这个矩阵称为 Fock 矩阵。改写上式为

$$\sum_k \sum_l C_{ik}^\dagger F_{kl} C_{lj} = \sum_k \sum_l C_{ik}^\dagger S_{kl} C_{lj} \varepsilon_j \tag{4.43}$$

即

$$\left(C^\dagger F C\right)_{ij} = \left(C^\dagger S C \varepsilon\right)_{ij} \tag{4.44}$$

其中 ε 为对角线为原本本征能量的对角阵，将上式的矩阵形式左乘 $C^{\dagger -1}$ 就得到了 Hartree-Fock 近似下最后求解的方程

$$FC = SC\varepsilon \tag{4.45}$$

上式经常用于在已知 Hartree-Fock 分子轨道能量的情况下反推 Fock 矩阵 (比如你有 Gaussian 计算结果，想知道 Fock 矩阵的时候)，即使用 $F = SC\varepsilon C^{-1}$。另外，上式其实是一个矩阵的形式，并不是常见的本征值方程，如果单独拿出某一列，才会化成一般的本征值问题，即

$$Fc = ESc \tag{4.46}$$

~~欢迎回来~~

所以我们虽然能够通过上面的积分方法得到 F，但我们不能直接把它投进任何一种可以矩阵对角化的库，因为它们得到的结果都是在基组互相正交归一的前提下进行的[11]线性代数曾教过我们如何解决这样的非正交情况下变成正交基组的操作——没错就是~~（翻阅谷歌一分钟）~~Gram–Schmidt 方法。在这里不过多讲述它，但它在做这个问题的时候的最大的毛病是它采用了迭代的方法来得到所有的正交基组，而我们现在用的所有数值方法都是存在误差的（比如 `double` 类型的精度问题），而在迭代得到每一个新的向量的过程中误差会不断地积累，最终爆炸——比如最后一个和第一个并不正交。所以我们需要一个整体进行正交化然后求解的方法——而这就是我们即将学习的 Löwdin 方法。

初始想法很简单——我们做这么一个变化：

$$FC = SCE \Rightarrow FS^{-\frac{1}{2}} S^{\frac{1}{2}} C = S^{\frac{1}{2}} S^{\frac{1}{2}} CE \Rightarrow S^{-\frac{1}{2}} F S^{-\frac{1}{2}} S^{\frac{1}{2}} C = S^{\frac{1}{2}} CE,$$

我们就能够看到对基组进行一个 $S^{1/2}$ 的变换，对 F 进行变换 $F' = S^{-\frac{1}{2}} F S^{-\frac{1}{2}}$，我们就能够完美抵消 S 的存在，最终就能够转换成为

$$F'C' = C'E. \tag{4.47}$$

[11]这句话当然有毛病——它们只帮你解决线性代数问题，而线性代数下的矢量内积是直接默认用点乘的。

于是问题转化成为如何求得 $S^{-\frac{1}{2}}$ 及 $S^{\frac{1}{2}}$。注意到 S 为实对称矩阵（而且是正定矩阵），

$$SU = sU \Leftrightarrow S = UsU^T \tag{4.48}$$

其中 s 为 S 对角化后的矩阵，U 为对应的特征向量组。然后约定 $s^{1/2}$ 表示每个特征值取根号值，那么

$$Us^{1/2}U^TUs^{1/2}U^T = UsU^T = S \tag{4.49}$$

因此我们可以判定 $Us^{1/2}U^T$ 为我们想要的 $S^{\frac{1}{2}}$. $S^{-\frac{1}{2}}$ 同理，从而有

$$S^{\frac{1}{2}} = Us^{1/2}U^T, \quad S^{-\frac{1}{2}} = Us^{-1/2}U^T. \tag{4.50}$$

有了这个我们就能愉快地求解非正交基组下的特征解问题了。

需要注意的是，进行变换后得到的特征向量组虽然库会帮你正交归一，但在实际使用的时候需要把它还原到原来的原子轨道基组中，也就是要乘上 $S^{-\frac{1}{2}}$.

代码也很明晰——对角化,做变换,组合后输出。函数为 `void gsl_eigen_Lowdin_diag (gsl_matrix * m, gsl_matrix * S, gsl_vector * eigen, gsl_matrix * eigenvec, int length )`, 定义于 `gslextra.cpp`:

```
void gsl_eigen_Lowdin_diag(gsl_matrix * m, gsl_matrix * S, gsl_vector * eigen,
    gsl_matrix * eigenvec, int length)
{
    int i;

    gsl_matrix * M, * S_minus_half, * U, * S_cpy, * temp;

    gsl_vector * S_eigen;

    M = gsl_matrix_calloc(length,length);
    // inverse square root of S
    S_minus_half = gsl_matrix_calloc(length,length);
    // copy of S
    S_cpy = gsl_matrix_calloc(length,length);
    // transformation matrix (storing all eigenvectors)
    U = gsl_matrix_calloc(length,length);
    // temporary matrix helping tranforming
    temp = gsl_matrix_calloc(length,length);

    // copy S (gsl_eigen_symmv will distroy the source matrix)
    gsl_matrix_memcpy(S_cpy,S);

    // vector storing eigenvalues
    S_eigen = gsl_vector_calloc(length);

    gsl_eigen_symmv_workspace * w = gsl_eigen_symmv_alloc(length);

    gsl_eigen_symmv(S_cpy,S_eigen,U,w);
```

```
    for(i=0;i<length;i++)
        gsl_matrix_set(S_minus_half,i,i,1.0/sqrt(gsl_vector_get(S_eigen,i)));

    gsl_matrix_mul(U,S_minus_half,temp,length,length,length);

    gsl_matrix_transpose(U);

    gsl_matrix_mul(temp,U,S_minus_half,length,length,length);

    gsl_matrix_mul(S_minus_half,m,temp,length,length,length);
    gsl_matrix_mul(temp,S_minus_half,M,length,length,length);

    gsl_eigen_symmv(M,eigen,temp,w);

    gsl_eigen_symmv_sort(eigen,temp,GSL_EIGEN_SORT_VAL_ASC);

    gsl_matrix_mul(S_minus_half,temp,eigenvec,length,length,length);

    gsl_matrix_free(M);
    gsl_matrix_free(S_minus_half);
    gsl_matrix_free(U);
    gsl_matrix_free(S_cpy);
    gsl_matrix_free(temp);

    gsl_vector_free(S_eigen);

    gsl_eigen_symmv_free(w);
}
```

顺便 `void gsl_matrix_mul(gsl_matrix * A, gsl_matrix *B, gsl_matrix * Result,int Acolumn ,int Arow,int Bcolumn)` 这个函数作为矩阵相乘并不是 gsl 库自带的（摊手），不过也不难写，就不展示了。

4.3.2 单电子哈密顿矩阵

这一部分是不包含电子之间的排斥的能量。我们有了 T 和 Z 后就很好求了，求矩阵元的函数为 `double single_electron_hamiltonian_matrix_element(orbital * a, orbital * b, atomic_orbital * atom_HEAD)`，定义于 `RHF.cpp`:

```
// calculate the single electron hamiltonian matrix (core hamiltonian matrix)
    element
double single_electron_hamiltonian_matrix_element(orbital * a, orbital * b,
    atomic_orbital * atom_HEAD)
{
    double result;
```

```
    atomic_orbital * atom_temp;

    result = orbital_kinetic_energy(a,b);

    atom_temp = atom_HEAD;

    while(atom_temp->NEXT != NULL)
    {
        result -= atom_temp->N * orbital_ZIntegral(a,b,atom_temp->cartesian);
        atom_temp = atom_temp->NEXT;
    }

    result -= atom_temp->N * orbital_ZIntegral(a,b,atom_temp->cartesian);

    return result;
}
```

4.3.3 初猜

初猜——Initial guess[12], 是通过迭代方法求解某类问题时必需的东西。在本程序中通过对单电子哈密顿量求特征解来获得初始的电子轨道构型。其实可以看到，在能量的构成中单电子哈密顿量占相当大的比重，所以通过单电子哈密顿量来获得初猜也不失为一个不错的选择。我们把单电子哈密顿矩阵假装为 Fock 矩阵:

$$hC_0 = SC_0E_0 \tag{4.51}$$

这个 C_0 便成为我们的初猜构型参与战斗。

代码也非常 easy:

```
// perform initial guess of the coefficient matrix by performing diagonalization of
    core hamiltonian matrix
void initial_guess(gsl_matrix * dest, gsl_matrix * core_hamiltonian, gsl_matrix * S,
     int length)
{
    int i;

    i = 0;

    gsl_vector * temp_vector;

    temp_vector = gsl_vector_calloc(length);

    gsl_eigen_Lowdin_diag(core_hamiltonian,S,temp_vector,dest,length);

    gsl_vector_free(temp_vector);
```

[12]这翻译不是我想出来的不关我事

```
}
```

4.3.4 HF 能量

一般 Hartree-Fock 的能量的求法是

$$\langle\Psi|\mathscr{H}|\Psi\rangle = \sum_{m=1}^{N}[m|h|m] + \sum_{m=1}^{N}\sum_{n>m}^{N}([mm|nn] - [mn|nm]) \tag{4.52}$$

但如果从二次量子化的角度出发，会得到一个很有意思的结果 [1]。在二次量子化下体系的哈密顿量表示为

$$\hat{H} = \sum_{\alpha\beta} h_{\alpha\beta}a_\alpha^\dagger a_\beta + \frac{1}{2}\sum_{\alpha\beta\gamma\delta} v_{\alpha\beta\gamma\delta}a_\alpha^\dagger a_\beta^\dagger a_\delta a_\gamma. \tag{4.53}$$

而电子密度算符为

$$\hat{\rho} = \sum_{\alpha=1}^{A}|\alpha\rangle\langle\alpha| = \sum_{\alpha=1}^{A}a_\alpha^\dagger|0\rangle\langle 0|a_\alpha \tag{4.54}$$

即表示所有具有电子占据的轨道对应的投影矩阵相加。而一个 Slater 行列式表示为

$$|\Phi\rangle = \prod_{\alpha=1}^{A}a_\alpha^\dagger|0\rangle \tag{4.55}$$

这就有意思了——相互组合，并注意算符交换导致的符号变化，然后就能够得到

$$E[\rho] = \sum_{ij} h_i j\rho_j i + \frac{1}{2}\sum_{ijkl}\bar{v}_{ijkl}\rho_{ji}\rho_{lk}, \tag{4.56}$$

其中

$$\bar{v}_{ijkl} = v_{ijkl} - v_{ikjl} = [ij|kl] - [ik|jl]. \tag{4.57}$$

所以实质上 Hartree-Fock 也是一种 DFT 泛函哒！

求变分，得到 Fock 矩阵

$$F_{ij} = h_{ij} + \sum_{kl}\bar{v}_{ijkl}\rho_{lk}. \tag{4.58}$$

从而那个四阶张量就可以派上用场啦！求电子密度矩阵很好求，只要注意数到电子数的一半就可以跳出循环。函数为 `void density_matrix(gsl_matrix * dest, gsl_matrix * coef, int el_num, int length)`，

```
// calculate density matrix from coefficient matrix and the number of electrons
void density_matrix(gsl_matrix * dest, gsl_matrix * coef, int el_num, int length)
{
    gsl_matrix * temp1, * temp2;
```

```
    gsl_vector * vector_temp;

    temp1 = gsl_matrix_calloc(length,length);
    temp2 = gsl_matrix_calloc(length,length);

    vector_temp = gsl_vector_calloc(length);

    int i;
    // get coefficient vectors of occupied orbitals and store into temp1, temp2
    for(i=0;i<el_num/2;i++)
    {
        gsl_matrix_get_col(vector_temp,coef,i);
        gsl_matrix_set_col(temp1,i,vector_temp);
        gsl_matrix_set_row(temp2,i,vector_temp);
    }

    // perform \sum n_a | a > < a |
    gsl_matrix_mul(temp1,temp2,dest,length,length,length);

    gsl_matrix_free(temp1);
    gsl_matrix_free(temp2);
}
```

在 RHF（Restricted Hartree-Fock）中我们只需要考虑轨道，而自旋导致的影响只会出现在双电子积分中：库仑积分为交换积分的两倍，即：

$$F = h_{ij} + \sum_{kl}(2v_{ijkl} - v_{ikjl})\rho_{lk}. \tag{4.59}$$

代码也非常明晰，矩阵元函数为 `double fock_matrix_element(gsl_quad_tensor * v, gsl_matrix * density_matrix, gsl_matrix * h_matrix, int i, int j, int length)`:

```
// calculate the fock matrix element (F_{ij} = h_{ji} + \sum \bar{v}_{ijkl} \rho_{lk
    })
double fock_matrix_element(gsl_quad_tensor * v, gsl_matrix * density_matrix,
    gsl_matrix * h_matrix, int i, int j, int length)
{
    double fock_matrix_temp;
    int k,l;

    fock_matrix_temp = gsl_matrix_get(h_matrix,j,i);

    for(k=0;k<length;k++)
    {
        for(l=0;l<length;l++)
        {
            // Coulomb integral
```

```
            fock_matrix_temp += 2 * gsl_quad_tensor_get(v,j,i,k,l) * gsl_matrix_get(
                density_matrix,l,k);
            // Exchange integral
            fock_matrix_temp -= gsl_quad_tensor_get(v,j,k,i,l) * gsl_matrix_get(
                density_matrix,l,k);
        }
    }

    return fock_matrix_temp;
}
```

能量求值也类似，函数为 `double HF_energy(gsl_quad_tensor * v, gsl_matrix * density_matrix, gsl_matrix * h_matrix, int length)`：

```
// calculate the Hartree-Fock energy of the system, E(HF) = \sum h_{ij} \rho_{ji} +
    0.5 \sum \rho_{lk} \bar{v}_{ijkl} \rho{ji}
double HF_energy(gsl_quad_tensor * v, gsl_matrix * density_matrix, gsl_matrix *
    h_matrix, int length)
{
    double energy_temp = 0;

    int i,j,k,l;

    for(i=0;i<length;i++)
    {
        for(j=0;j<length;j++)
        {
            // add core hamiltonian energy
            energy_temp += gsl_matrix_get(h_matrix,i,j) * gsl_matrix_get(
                density_matrix,j,i);
        }
    }

    for(i=0;i<length;i++)
    {
        for(j=0;j<length;j++)
        {
            for(k=0;k<length;k++)
            {
                for(l=0;l<length;l++)
                {
                    // Coulumb integrals
                    energy_temp += gsl_quad_tensor_get(v,i,j,k,l) * gsl_matrix_get(
                        density_matrix,j,i) * gsl_matrix_get(density_matrix,l,k);
                    // Exchange integrals
                    energy_temp -= 0.5 * gsl_quad_tensor_get(v,i,k,j,l) *
                        gsl_matrix_get(density_matrix,j,i) * gsl_matrix_get(
                        density_matrix,l,k);
```

```
            }
        }
    }
}

    return 2.0 * energy_temp;
}
```

有了这些你也可以打造自己的 Hartree-Fock 循环啦!

下面是我的循环:

```
for(i=0;i<iteration_max;i++)
{
    fock_matrix(F,v,D,h_matrix,length);
    gsl_eigen_Lowdin_diag(F,S,energy,coef,length);
    density_matrix(D,coef,el_num,length);

    energy_temp = HF_energy(v,D,h_matrix,length);

    if(fabs(energy_temp - energy_bk) < errmax) count++;
    else count = 0;

    if(count >= countmax) break;

    printf("iteration␣=␣%d,␣energy␣=␣%lf\n",i+1,energy_temp);
    if(SCF_FOCK_FLAG==1)
    {
        printf("\nFock␣matrix:\n");
        gsl_matrix_printf(F,length,length,"%10.4f");
    }
    if(SCF_COEF_FLAG==1)
    {
        printf("\nCoefficient␣matrix:\n");
        gsl_matrix_printf(coef,length,length,"%10.4f");
    }



    gsl_matrix_memcpy(diff_temp,diff);

    gsl_matrix_memcpy(diff,coef);
    gsl_matrix_sub(diff,input_coef_temp);



    energy_bk = energy_temp;
```

甚至自信地认为不需要进行额外的注释（骄傲地

4.3.5 Anderson's mixing

一般来说迭代意味着要把上一次获得的结果以某种形式代入下一步进行运算——比如进行 SCF 迭代的时候，最容易的方法就是把上一次得到的系数矩阵直接拿过来算下一步的电子密度矩阵，然后算出 Fock 矩阵，再对角化，如此循环——但这种情况很难保证在一定的循环次数内达到收敛（即相差数值很小），而且也很难说是一定能收敛——从二分法就可以很容易想象。因此有很多 mixing 的方法，把前几次得到的运算结果以某种形式混合起来形成新的结果猜测，从而更快地达到收敛。本程序使用了相对简单的 Anderson's mixing 的方法 [2]。

最简单的 mixing 的方法就是直接按照一定比例混合：

$$|n^{(m+1)}\rangle = (1-\alpha)|n^{(m)}\rangle + \alpha|n_{\text{out}}^{(m)}\rangle = |n^{m}\rangle + \alpha|F^{(m)}\rangle, \tag{4.60}$$

其中 $|n^{(m)}\rangle$ 表示第 m 次输入，$|n_{\text{out}}^{(m)}\rangle$ 表示第 m 次输出，以及

$$|F^{(m)}\rangle \equiv |n_{\text{out}}^{(m)}\rangle - |n^{(m)}\rangle, \tag{4.61}$$

也就是把第 m 次的输入和输出按照一定比例混合然后作为第 $m+1$ 次输入。但这个方法收敛也容易很慢，因为在即将接近结果的时候跳得过快，使得在很多情况下需要让 α 设得非常小才能收敛。

Anderson's mixing 试图通过一个可调的 β 来将前一次的输入/输出和再前一次的输入/输出进行混合，从而起到把两次输入/输出综合考量的结果，并通过上面的方法把输入/输出混合起来形成下一步的猜测：

$$|\bar{n}_{\text{in(out)}}\rangle = (1-\beta)|n_{\text{in(out)}}^{(m)}\rangle + \beta|n_{\text{in(out)}}^{(m-1)}\rangle. \tag{4.62}$$

而 β 在趋近的过程中应该满足

$$\partial D\left[\overline{n}_{\text{in}}, \overline{n}_{\text{out}}\right]/\partial\beta = 0, \tag{4.63}$$

其中

$$\begin{aligned} D\left[n_{\text{out}}, n\right] &\equiv \left(\langle n_{\text{out}} - n|n_{\text{out}} - n\rangle\right)^{1/2} \\ &= \langle F(n)|F(n)\rangle^{1/2} \end{aligned}. \tag{4.64}$$

从而有

$$\beta = \frac{\left\langle F^{(m)}|F^{(m)} - F^{(m-1)}\right\rangle}{D^2\left[F^{(m)}, F^{(m-1)}\right]}, \tag{4.65}$$

而后通过原来的 α 进行混合：

$$\left|n^{(m+1)}\right\rangle = (1-\alpha)\left|\overline{n}_{\text{in}}^{(m)}\right\rangle + \alpha\left|\overline{n}_{\text{out}}^{(m)}\right\rangle. \tag{4.66}$$

代码体现为

```
// Anderson's mixing
if(i>3 && mixing_type==1)
{
    for(j=0;j<length;j++)
    {
        //calculating parameter beta
        gsl_matrix_get_col(vector_temp1,diff,j);
        gsl_matrix_get_col(vector_temp2,diff_temp,j);

        difference_temp = gsl_vector_inner_product(vector_temp1,vector_temp1,length)
            - gsl_vector_inner_product(vector_temp1,vector_temp2,length);

        gsl_vector_sub(vector_temp2,vector_temp1);

        beta = difference_temp/gsl_vector_inner_product(vector_temp2,vector_temp2,
            length);

        // Set the output part |n_{out}>
        gsl_matrix_get_col(vector_temp1,coef,j);
        gsl_vector_scale(vector_temp1,1.0 - beta);
        gsl_matrix_get_col(vector_temp2,output_coef_temp,j);
        gsl_vector_scale(vector_temp2,beta);
        gsl_vector_add(vector_temp1,vector_temp2);

        // Start writing the next input matrix
        gsl_matrix_set_col(mixing_temp,j,vector_temp1);

        // Set the input part |n_{in>}
        gsl_matrix_get_col(vector_temp1,input_coef_temp,j);
        gsl_vector_scale(vector_temp1,1.0-beta);
        gsl_matrix_get_col(vector_temp2,diff_temp,j);
        gsl_vector_scale(vector_temp2,beta);
        gsl_vector_sub(vector_temp1,vector_temp2);
        gsl_matrix_get_col(vector_temp2,output_coef_temp,j);
        gsl_vector_scale(vector_temp2,beta);
        gsl_vector_add(vector_temp1,vector_temp2);

        gsl_vector_scale(vector_temp1,1.0-alpha);
        gsl_matrix_get_col(vector_temp2,mixing_temp,j);
        gsl_vector_scale(vector_temp2,alpha);
        gsl_vector_add(vector_temp1,vector_temp2);
        gsl_matrix_set_col(mixing_temp,j,vector_temp1);
    }

    // overwrite the coef matrix
    gsl_matrix_memcpy(output_coef_temp,coef);
```

```
    gsl_matrix_memcpy(coef,mixing_temp);
}
gsl_matrix_memcpy(input_coef_temp,coef);
```

5 效果展示

```
    $ ./HF -f test/H2O.in
```

使用 3-21G，输出结果为

```
    SCF converged.

    Fock matrix:

     -0.437615 -0.404153 -1.133001 -1.022207 -0.415453 0.393827 0.000000
     -0.404153 -0.437615 -1.133001 -1.022207 0.415453 0.393827 0.000000
     -1.133001 -1.133001-20.146148 -5.145982 -0.000000 0.027780 0.000000
     -1.022207 -1.022207 -5.145982 -2.331408 -0.000000 0.111017 0.000000
     -0.415453 0.415453 -0.000000 -0.000000 -0.227014 0.000000 0.000000
      0.393827 0.393827 0.027780 0.111017 0.000000 -0.275262 0.000000
      0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 -0.296996



    MOs:

    MO_NUM: 0, MO_ENERGY = -20.154493 , occ = 2
    MO_NUM: 1, MO_ENERGY = -1.208528 , occ = 2
    MO_NUM: 2, MO_ENERGY = -0.557811 , occ = 2
    MO_NUM: 3, MO_ENERGY = -0.379794 , occ = 2
    MO_NUM: 4, MO_ENERGY = -0.296996 , occ = 2
    MO_NUM: 5, MO_ENERGY = 0.963347 , occ = 0
    MO_NUM: 6, MO_ENERGY = 1.048381 , occ = 0

    Total Energy: -75.003904

    MO_LABEL:
    [ H1s , H1s , O1s , O2s , O2py , O2pz , O2px ]

    MO_COEFF:


       0.003630 -0.208665 0.422342 -0.230836 0.000000 -0.778788 -0.829289
       0.003630 -0.208665 -0.422342 -0.230836 0.000000 -0.778788 0.829289
      -0.994402 0.222023 0.000000 -0.106497 0.000000 -0.142506 -0.000000
      -0.024274 -0.772323 -0.000000 0.522462 0.000000 0.898434 0.000000
```

```
-0.000000 -0.000000 0.643727 0.000000 0.000000 -0.000000 0.939536
 0.003189 0.128637 -0.000000 0.800802 0.000000 -0.715459 -0.000000
 0.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000
```

References

[1] Trygve Helgaker, Poul Jorgensen, and Jeppe Olsen. *Molecular electronic-structure theory*. John Wiley & Sons, 2014.

[2] Duane D Johnson. Modified broyden' s method for accelerating convergence in self-consistent calculations. *Physical Review B*, 38(18):12807, 1988.

[3] Andrew James May. *Density fitting in explicitly correlated electronic structure theory*. PhD thesis, University of Bristol, 2006.

[4] Shigeru Obara and A Saika. Efficient recursive computation of molecular integrals over cartesian gaussian functions. *The Journal of chemical physics*, 84(7):3963–3974, 1986.

[5] Karen L Schuchardt, Brett T Didier, Todd Elsethagen, Lisong Sun, Vidhya Gurumoorthi, Jared Chase, Jun Li, and Theresa L Windus. Basis set exchange: a community database for computational sciences. *Journal of chemical information and modeling*, 47(3):1045–1052, 2007.