ELECTRONIC STRUCTURE - PRACTICAL SESSION 4

Self Consistent Field Calculations:
Restricted Hartree-Fock method for two-electron diatomic molecules (Part II)

Gisela Martí Guerrero

January 2023

1 Problem 1

Write the general SCF program for two-electron diatomic molecules described above and use it afterwards to calculate the potential energy curves for the H_2 and HHe^+ molecules in their ground states. What is the equilibrium geometry for the two molecules? To which products does the energy in the dissociation limit correspond?

First, the integrals are calculated using $\zeta = 1.24$ for hydrogen atoms, and $\zeta = 2.0925$ for helium atoms. The intranuclear distances considered are $R = 1.4632$ bohr for HHe^+ molecules and $R = 1.4$ bohr for H_2

```
import numpy as np
import scipy.special as sp

R = float(input("Enter the intranuclear distance between the two atoms: ")) #1.4632 (HHe+), 1.4 (H2)
Z_a = int(input("Enter the atomic number of atom 1: ")) #2 (He), 1 (H)
Z_b = int(input("Enter the atomic number of atom 2: "))
zeta_a = float(input("Enter the zeta constant of atom 1: ")) #2.0925 (He), 1.24 (H)
zeta_b = float(input("Enter the zeta constant of atom 2: "))

N = 3 #number of Gaussian primitives
T = np.zeros([2,2])
V = np.zeros([2,2])
S = np.zeros([2,2])
if Z_a == Z_b:
    E = np.zeros([4,5])
else:
    E = np.zeros([6,5])
Rab2 = R**2
Nelec = 2
dim = 2 #dim is the number of basis functions

def S_int(A,B,Rab2): #A and B are the exponents of the atoms
    """
    Calculates the overlap between two gaussian functions
    """
    return (np.pi/(A+B))**1.5*np.exp(-A*B*Rab2/(A+B))

def T_int(A,B,Rab2):
    """
    Calculates the kinetic energy integrals for un-normalised primitives
    """
    return A*B/(A+B)*(3.0-2.0*A*B*Rab2/(A+B))*(np.pi/(A+B))**1.5*np.exp(-A*B*Rab2/(A+B))

def V_int(A,B,Rab2,Rcp2,Zc):
    """
    Calculates the un-normalised nuclear attraction integrals
    """
    V = (2.0*np.pi/(A+B))*F0((A+B)*Rcp2)*np.exp(-A*B*Rab2/(A+B))
    return -V*Zc
```

```
# Mathematical functions
def F0(t):
    """
    F function for 1s orbital
    """
    if (t<1e-6):
        return 1.0-t/3.0
    else:
        return 0.5*(np.pi/t)**0.5*sp.erf(t**0.5)

def erf(t):
    """
    Approximation for the error function
    """
    P = 0.3275911
    A = [0.254829592,-0.284496736,1.421413741,-1.453152027,1.061405429]
    T = 1.0/(1+P*t)
    Tn=T
    Poly = A[0]*Tn
    for i in range(1,5):
        Tn=Tn*T
        Poly=Poly*A[i]*Tn
    return 1.0-Poly*np.exp(-t*t)

coeff = np.array([[1.00000,0.0000000,0.000000],  #STO-1G
                  [0.678914,0.430129,0.000000],  #STO-2G
                  [0.444635,0.535328,0.154329]]) #STO-3G

expon = np.array([[0.270950,0.000000,0.000000],  #STO-1G
                  [0.151623,0.851819,0.000000],  #STO-2G
                  [0.109818,0.405771,2.227660]]) #STO-3G

D1 = np.zeros([3])
A1 = np.zeros([3])
D2 = np.zeros([3])
A2 = np.zeros([3])

# This loop constructs the contracted Gaussian functions
for i in range(N):
    A1[i] = expon[N-1,i]*(zeta_a**2)
    D1[i] = coeff[N-1,i]*((2.0*A1[i]/np.pi)**0.75)
    A2[i] = expon[N-1,i]*(zeta_b**2)
    D2[i] = coeff[N-1,i]*((2.0*A2[i]/np.pi)**0.75)

S12 = 0.0
T11 = 0.0
T12 = 0.0
T22 = 0.0
V11A = 0.0
V12A = 0.0
V22A = 0.0
V11B = 0.0
V12B = 0.0
V22B = 0.0

if Z_a == Z_b:
    for i in range(N):
        for j in range(N):
            Rap = A2[j]*R/(A1[i]+A2[j])
            Rap2 = Rap**2 # Rap2 - squared distance between centre A and centre P
            Rbp2 = (R-Rap)**2
            S12 = S12 + S_int(A1[i],A2[j],Rab2)*D1[i]*D2[j]
            T11 = T11 + T_int(A1[i],A1[j],0.0)*D1[i]*D1[j]
            T12 = T12 + T_int(A1[i],A2[j],Rab2)*D1[i]*D2[j]
            T22 = T22 + T_int(A2[i],A2[j],0.0)*D2[i]*D2[j]
            V11A = V11A + V_int(A1[i],A1[j],0.0,0.0,Z_a)*D1[i]*D1[j]
            V12A = V12A + V_int(A1[i],A2[j],Rab2,Rap2,Z_a)*D1[i]*D2[j]
            V22A = V22A + V_int(A2[i],A2[j],0.0,Rab2,Z_a)*D2[i]*D2[j]

    V[0,0] = V11A*2
    V[0,1] = 0.0
    V[1,0] = V12A*2
    V[1,1] = V22A*2
else:
    for i in range(N):
        for j in range(N):
            Rap = A2[j]*R/(A1[i]+A2[j])
            Rap2 = Rap**2
            Rbp2 = (R-Rap)**2
            S12 = S12 + S_int(A1[i],A2[j],Rab2)*D1[i]*D2[j]
            T11 = T11 + T_int(A1[i],A1[j],0.0)*D1[i]*D1[j]
            T12 = T12 + T_int(A1[i],A2[j],Rab2)*D1[i]*D2[j]
            T22 = T22 + T_int(A2[i],A2[j],0.0)*D2[i]*D2[j]
            V11A = V11A + V_int(A1[i],A1[j],0.0,0.0,Z_a)*D1[i]*D1[j]
            V12A = V12A + V_int(A1[i],A2[j],Rab2,Rap2,Z_a)*D1[i]*D2[j]
            V22A = V22A + V_int(A2[i],A2[j],0.0,Rab2,Z_a)*D2[i]*D2[j]
            V11B = V11B + V_int(A1[i],A1[j],0.0,Rab2,Z_b)*D1[i]*D1[j]
            V12B = V12B + V_int(A1[i],A2[j],Rab2,Rbp2,Z_b)*D1[i]*D2[j]
            V22B = V22B + V_int(A2[i],A2[j],0.0,0.0,Z_b)*D2[i]*D2[j]

    V[0,0] = V11A + V11B
    V[0,1] = 0.0
    V[1,0] = V12A + V12B
    V[1,1] = V22A + V22B

S[0,0] = 1.0 #Saa
S[0,1] = 0.0 #Sab
S[1,0] = S12 #Sba
S[1,1] = 1.0 #Sbb

T[0,0] = T11
T[0,1] = 0.0
T[1,0] = T12
T[1,1] = T22

def TwoE(A,B,C,D,Rab2,Rcd2,Rpq2):
    """
    Calculate two electron integrals
    A,B,C,D are the exponents alpha, beta, etc.
    Rab2 equals squared distance between centre A and centre B
    """
    return 2.0*(np.pi**2.5)/((A+B)*(C+D)*np.sqrt(A+B+C+D))*F0((A+B)*(C+D)*Rpq2/(A+B+C+D))*
    np.exp(-A*B*Rab2/(A+B)-C*D*Rcd2/(C+D))

E1111 = 0.0
E1122 = 0.0
E2111 = 0.0
E2121 = 0.0
E2211 = 0.0
E2221 = 0.0
E2222 = 0.0
```

```
if Z_a == Z_b:
    # Calculate two electron integrals
    for i in range(N):
        for j in range(N):
            for k in range(N):
                for l in range(N):
                    Rap = A2[i]*R/(A2[i]+A1[j])
                    Rbp = R - Rap
                    Raq = A2[k]*R/(A2[k]+A1[l])
                    Rbq = R - Raq
                    Rpq = Rap - Raq
                    Rap2 = Rap*Rap
                    Rbp2 = Rbp*Rbp
                    Raq2 = Raq*Raq
                    Rbq2 = Rbq*Rbq
                    Rpq2 = Rpq*Rpq

                    E1111 = E1111 + TwoE(A1[i],A1[j],A1[k],A1[l],0.0,0.0,0.0)*D1[i]*D1[j]*D1[k]*D1[l]
                    E1122 = E1122 + TwoE(A1[i],A1[j],A2[k],A2[l],0.0,0.0,Rab2)*D1[i]*D1[j]*D2[k]*D2[l]
                    E2111 = E2111 + TwoE(A2[i],A1[j],A1[k],A1[l],Rab2,0.0,Rap2)*D2[i]*D1[j]*D1[k]*D1[l]
                    E2121 = E2121 + TwoE(A2[i],A1[j],A2[k],A1[l],Rab2,Rab2,Rpq2)*D2[i]*D1[j]*D2[k]*D1[l

    #build the two-electron integral's matrix
    E[0,:] = (1,1,1,1,E1111)
    E[1,:] = (1,1,2,2,E1122)
    E[2,:] = (2,1,1,1,E2111)
    E[3,:] = (2,1,2,1,E2121)
else:
    for i in range(N):
        for j in range(N):
            for k in range(N):
                for l in range(N):
                    Rap = A2[i]*R/(A2[i]+A1[j])
                    Rbp = R - Rap
                    Raq = A2[k]*R/(A2[k]+A1[l])
                    Rbq = R - Raq
                    Rpq = Rap - Raq
                    Rap2 = Rap*Rap
                    Rbp2 = Rbp*Rbp
                    Raq2 = Raq*Raq
                    Rbq2 = Rbq*Rbq
                    Rpq2 = Rpq*Rpq
                    E1111 = E1111 + TwoE(A1[i],A1[j],A1[k],A1[l],0.0,0.0,0.0)*D1[i]*D1[j]*D1[k]*D1[l]
                    E2111 = E2111 + TwoE(A2[i],A1[j],A1[k],A1[l],Rab2,0.0,Rap2)*D2[i]*D1[j]*D1[k]*D1[l]
                    E2121 = E2121 + TwoE(A2[i],A1[j],A2[k],A1[l],Rab2,Rab2,Rpq2)*D2[i]*D1[j]*D2[k]*D1[l
                    E2211 = E2211 + TwoE(A2[i],A2[j],A1[k],A1[l],0.0,0.0,Rab2)*D2[i]*D2[j]*D1[k]*D1[l]
                    E2221 = E2221 + TwoE(A2[i],A2[j],A2[k],A1[l],0.0,Rab2,Rbq2)*D2[i]*D2[j]*D2[k]*D1[l]
                    E2222 = E2222 + TwoE(A2[i],A2[j],A2[k],A2[l],0.0,0.0,0.0)*D2[i]*D2[j]*D2[k]*D2[l]

    E[0,:] = (1,1,1,1,E1111)
    E[1,:] = (2,2,2,2,E2222)
    E[2,:] = (2,2,1,1,E2211)
    E[3,:] = (2,1,1,1,E2111)
    E[4,:] = (2,2,2,1,E2221)
    E[5,:] = (2,1,2,1,E2121)
```

Once we have all the integrals calculated and put into matrices, we follow the same SCF code as practice 3.

```
# Step 3: Diagonalize S and use its eigenvalues to obtain a transformation matrix X
def symmetrise(matrix):
    """
    Function to symmetrize a matrix given a triangular one
    """
    return matrix + matrix.T - np.diag(matrix.diagonal())

# Flip the triangular matrix in the diagonal
S = symmetrise(S)
V = symmetrise(V)
T = symmetrise(T)
# Form core Hamiltonian matrix as sum of the T and V matrices
Hcore = T + V

# Diagonalize overlap matrix. S_val are the eigenvalues and S_vec the eigenvectors
S_val, S_vec = np.linalg.eigh(S)
# Find inverse square root of eigenvalues
s_half = (np.diag(S_val**(-0.5)))
# Form the transformation matrix X. The unitary matrix are the eigenvectors
X_matrix = -np.dot(S_vec, np.dot(s_half, np.transpose(S_vec)))

# Step 4: Construct a guess denisty matrix P(0) (null)
P = np.zeros((dim, dim))

# Step 5: Calculate the bielectronic term G(0) using the guess denisty matrix and the two
  electron integrals
def eint(a,b,c,d):
    if a > b:
        ab = a*(a+1)/2 + b
    else:
        ab = b*(b+1)/2 + a
    if c > d:
        cd = c*(c+1)/2 + d
    else:
        cd = d*(d+1)/2 + c
    if ab > cd:
        abcd = ab*(ab+1)/2 + cd
    else:
        abcd = cd*(cd+1)/2 + ab
    return abcd

# two-electron integrals are stored in a dictionary
twoe = {eint(row[0], row[1], row[2], row[3]) : row[4] for row in E}

def two_elec_int(a, b, c, d): # Return value of two electron integral
    """
    Return value of two electron integral
    """
    return twoe.get(eint(a, b, c, d), 0)

# Step 6: Add G(0) to the one-electron term h to get a first guess of the Fock matrix F(0) = h + G(0).
def fock_matrix(Hcore, P, dim):
    """
    Function to build the Fock Matrix
    """
    F = np.zeros((dim, dim)) # zero array
    for i in range(0, dim):
        for j in range(0, dim):
            F[i,j] = Hcore[i,j] # initial Fock matrix
            for k in range(0, dim):
                for l in range(0, dim):
                    # Form the Fock matrix using the product of the density matrix and G matrix
                    F[i,j] = F[i,j] + P[k,l]*(two_elec_int(i+1,j+1,k+1,l+1)-0.5*
                              two_elec_int(i+1,k+1,j+1,l+1))
    return F

# Step 7: Transform the Fock matrix
def f_transform(X, F):
    """
    Transform Fock matrix with the transformation matrix X
    """
    return np.dot(np.transpose(X), np.dot(F, X))

# Step 10: Form a new density matrix P(1) using C(1)
def density_matrix(C, D, dim, Nelec): # Make density matrix and store old one to test for convergence
    """
    Make new density matrix and store old one to test for convergence

    Returns:
        D: new density matrix
        Dold: old denisty matrix
    """
    Dold = np.zeros((dim, dim)) # Initiate zero array
    for mu in range(0, dim):
        for nu in range(0, dim):
            Dold[mu,nu] = D[mu, nu] # Set old density matrix to the density matrix, D, input
                                      into the function
            D[mu,nu] = 0
            for m in range(0, int(Nelec/2)):
                # Form new density matrix
                D[mu,nu] = D[mu,nu] + 2*C[mu,m]*C[nu,m]
    return D, Dold

# Step 11: Determine if the process has converged by comparing P(1) with P(0).
def threshold(D, Dold):
    """
    Calculate change in density matrix using Root Mean Square Deviation (RMSD)
    """
    DELTA = 0.0
    for i in range(0, dim):
        for j in range(0, dim):
            DELTA = DELTA + ((D[i,j] - Dold[i,j])**2)

    return (DELTA/4.0)**(0.5)

# Step 12: If the process has converged, use the resultant solution, represented by C(k),
  P(k), and F(k)
def energy_iteration(D, Hcore, F, dim):
    """
    Function that calculates the energy at each iteration
    """
    EN = 0
    for mu in range(0, dim):
        for nu in range(0, dim):
            EN += 0.5*D[mu,nu]*(Hcore[mu,nu] + F[mu,nu])
    return EN

# Finally we make the iteration loop
DELTA = 1
count = 0 # cycles counter
nuclear_repulsion = (Z_a*Z_b)/R
while DELTA > 1e-4:
    count += 1
    F = fock_matrix(Hcore, P, dim) # Calculate Fock matrix (step 6)
    Fprime = f_transform(X_matrix, F) # Calculate transformed Fock matrix (step 7)
    E, Cprime = np.linalg.eigh(Fprime) # Diagonalize F' matrix (step 8)
    C = np.dot(X_matrix, Cprime) # transform the coefficients into original basis using transformation
    P, OLDP = density_matrix(C, P, dim, Nelec) # Make density matrix (step 10)
    DELTA = threshold(P, OLDP) # Test for convergence (step 11)
    print("E = {:.6f}, N(SCF) = {}".format(energy_iteration(P, Hcore, F, dim), count))

print("SCF procedure complete, TOTAL E(SCF) = {} a.u.".format(energy_iteration(P, Hcore, F, dim)))
print("------------------------")
print("The total energy is: ", energy_iteration(P, Hcore, F, dim) + nuclear_repulsion, "a.u.")
print("------------------------")
print("The expansion coefficients matrix is: ","\n", C)
print("------------------------")
print("The orbital energies matrix is: ","\n", np.diag(E))
print("------------------------")
print("The final delta value is: ",DELTA)
```