UNIVERSITÀ
DI TRENTO

Quantum molecular dynamics of the hydrogen molecule: supporting informations

Giuseppe Gambini

Contents

1 Run the code

In this supporting information file, we briefly give an overview on the procedures written in the code. Since the code is quite extended, we only focus on the most important functions implemented. We also describe possible usage scenarios. The code can perform different jobs depending on the input passed by the user.

The user has to compile the Makefile by prompting `make` on a terminal window in the same folder of the Makefile. Then the program can be executed with `./H2 OPTION_NAME`.

If the user wants to remove all the binary `.o` files from the folder in which they have been created, it is sufficient to type `make clean` on a terminal window.
The possible choices for `OPTION_NAME` are the following:

- Programs in which Hartree-Fock energy is computed: `SC_HF, BOMD_HF, CPMD_HF, CGMD_HF`.
- Programs in which DFT energy is computed: `SC_DFT, BOMD_DFT, CPMD_DFT, CGMD_DFT`.
- Car-Parrinello evolution of orbital energy: `Evolve_coefficients`.
- Scalar product between sets of coefficients: `Scal_prod`.
- Print the integrals of the exchange and correlation matrix: `EX_CORR`

The options of the type `AAMD_BBB` (where `AA` can be `BO`, `CP` or `CG` and `BBB` can be either `HF` or `DFT`) represent the core of the whole program, providing the main observables and trajectories.

1.1 Main outputs of the `AAMD_BBB` options

The code returns one coefficients file with the name `AAMD_BBB_coeff.txt` and a file with statistics `AAMD_BBB_X_energies.txt`.

The columns of the `X_energies.txt` files are organised as follows: `X` (internuclear distance), `E_0` (HF or DFT energy), `T_N` (nuclear kinetic energy), `E_ee` (electron-electron energy), `E_ob` (energy associated with one-body operators), `E_xc` (exchange and correlation energy), `T_f` fictitious kinetic energy, `F` (force given by the gradient of the energy).

X	E_0	T_N	E_ee	E_ob	E_xc	T_f	F
1.43462	-1.17521	0.0218615	-0.635305	-2.47389	-0.0985913	1.55495e-06	0.259554
1.45561	-1.17467	0.0213696	-0.631927	-2.45949	-0.0983689	1.48501e-06	0.264643
1.47635	-1.17401	0.0206266	-0.628632	-2.44545	-0.0981512	1.40099e-06	0.269236
1.49673	-1.17324	0.0196661	-0.625436	-2.43185	-0.0979398	1.3063e-06	0.273357
1.51663	-1.17237	0.0185238	-0.622361	-2.41874	-0.0977344	1.20406e-06	0.277013

The fictitious kinetic energy `T_f` column is provided only if `CPMD_HF` and `CPMD_DFT` are specified. Moreover, the column containing the exchange and correlation energy `E_xc` is reported only in `DFT` runs.

The coefficients file contain eight columns each containing the time evolution of a component of $\mathbf{C}$.

Due to the evident symmetry of H_2, the first four components are equal to the other four.

At the end of each simulation, the code will prompt the parameters γ_N, M_N, h_N and h, m used in the simulation as well as the name of the functional employed in case of `DFT`.

The execution time required for the `DFT` cases is much longer than the time required for `HF`. For example, approximately four seconds are required for single nuclear step in a computer with 1.8 GHz dual core. For this reason, we implemented the possibility to restart the simulation provided that the `coeff.txt` and `X_energies.txt` files are in the current folder.

! To restart a DFT simulation the user can execute `./H2 AAMD_DFT` inside the folder of the previously obtained `coeff.txt` and `X_energies.txt` files.

2 Structure of the code

All the functions are listed together in the header file `definitions.h`.

! The following `C++` libraries must be installed on the local machine: `iostream`, `fstream`, `sstream`, `cstdlib`, `string`, `cmath`, `vector`, `gsl/gsl_math.h`, `gsl/gsl_eigen.h`, `gsl/gsl_matrix.h`, `gsl/gsl_blas.h`, `xc.h`

Typically, these files are located on Mac in the folder `/usr/local/Cellar`. For windows user we recommend the following guide to include libraries in the path. The user can modify the list of parameters of the program in the header file `definitions.h`:

```
#define FUNCTIONAL_X XC_LDA_X
#define FUNCTIONAL_C XC_LDA_C_PZ

const int iter = 800; /* iterations used in BOMD or Conjugate Gradient */
const double m = 2.0; /* fictitious mass for electronic problem */
const double gamma_el= 1.0; /* electronic damping */
const double M_N = 1836.5; /* nuclear mass */
const double gamma_N = 15.0; /* nuclear damping */
const double h = 0.1; /* electronic time scale */
const double h_N = 43*h; /* nuclear time scale*/
const int N = 4; /* basis centerd on each atom */
const double a[N] = {13.00773, 1.962079, 0.444529, 0.1219492}; /* exponents of the Gaussians */
const double pi = 3.141592653589793;
const double a_x = 0.0; /* for inclusion of exchange functional */
```

The code supports only the local density approximation (LDA) exchange and correlation functionals. The possible macros labelling the functionals are listed on the libxc site. The parameter `a_x` can be varied by the user to tune the functional. The values of γ_N, M_N, h_N and h, m (needed only for CP dynamics) are suggested in the book by Thijssen.

3 Principal functions

3.1 Filling the matrices and their derivatives

To avoid the allocation of a large number of matrices, we passed the same matrices to different functions, taking advantage of the fact that at all the Verlet steps these matrices have to be recomputed. To better explain this concept we show a piece of code in which first the matrices `S`, `H` and the tensor `Q` are computed to solve the electronic problem and then refilled with their derivatives with respect to X:

```
/**** Fill S, H and Q for electronic problem ****/
create_S(S, R_A, R_B);
one_body_H(H, R_A, R_B);
build_Q(Q, R_A, R_B);

...

/**** Fill H, Q and F for nuclear problem ****/
create_dS_dX(S, R_A, R_B);
one_body_dH_dX(H, R_A, R_B, X[n]);
build_dQ_dX(Q, R_A, R_B, X[n]);
```

The functions filling the matrices use the following functions defined in `matrix_elements.cpp` and `der_matrix_elements.cpp`.

$\mathcal{S}$	`overlap`	$\partial\mathcal{S}/\partial X$	`doverlap_dX`
$\mathcal{H}$	`laplacian` and `el_nucl`	$\partial\mathcal{H}/\partial X$	`dlaplacian_dX` and `del_nucl_dX`
$\mathcal{Q}$	`direct_term`	$\partial\mathcal{Q}/\partial X$	`ddirect_term_dX`

3.2 Adaptive Simpson integration

The functions of this section are written inside `Adaptive.cpp`

`Integrand`: this function computes the integrands $\rho\chi_p(\rho,z)\chi_q(\rho,z), v_{xc}(n(\rho,z))$. The extra ρ comes from the Jacobian associated to the cylindrical coordinates. Below, we show how this functions collects the values of the exchange and correlation potential v_{xc} from the values of the density `n`:

```
/**** Compute the density ****/
double n = density(rho, z, c, X);

/**** Compute the exchange part ****/
double v_x = 0.0, v_c = 0.0;
xc_func_type functional_x;
xc_func_init(&functional_x, FUNCTIONAL_X, XC_UNPOLARIZED);
xc_lda_vxc(&functional_x, 1, &n, &v_x);
xc_func_end(&functional_x);

/**** Compute the correlation part ****/
xc_func_type functional_c;
xc_func_init(&functional_c, FUNCTIONAL_C, XC_UNPOLARIZED);
xc_lda_vxc(&functional_c, 1, &n, &v_c);
xc_func_end(&functional_c);
```

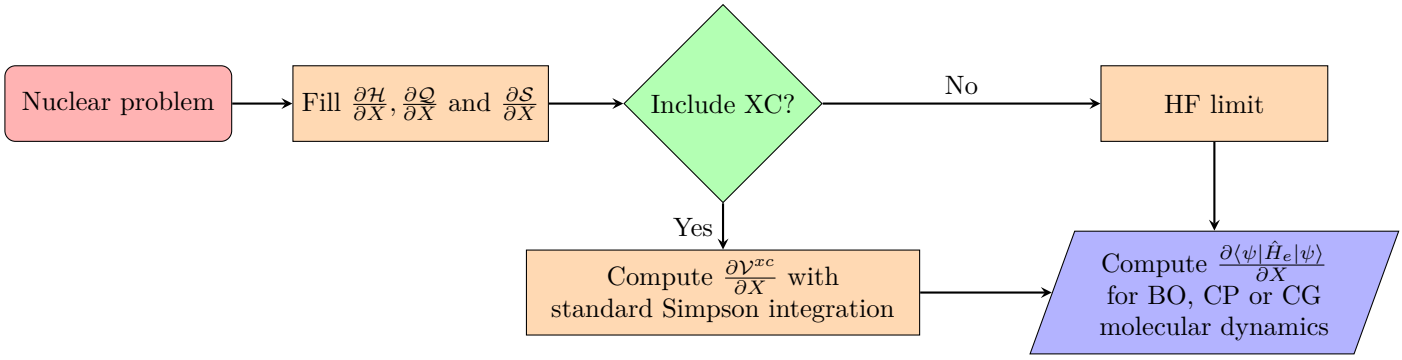
`Adaptive_Simpson_rho`: given two extremes of integrations in input, it computes the integral over `rho` of the function returned by `Integrand` with an adaptive Simpson integration. The coordinate `z` is treated as a parameter. The core of the function is the recursive part

```
/**** Lyness 1969 + Richardson extrapolation, see article ****/
if(fabs(delta) <= 15*eps){
    return left + right + delta/15;
}else{
    double new_left = Adaptive_Simpsons_rho(a, m, z, eps/2.0, left, alpha, beta, R_A, R_B, c, X, s);
    double new_right = Adaptive_Simpsons_rho(m, b, z, eps/2.0, right, alpha, beta, R_A, R_B, c, X, s);
    return new_left + new_right;
}
```

`Adaptive_Simpson_z`: it uses the integrals computed in `Adaptive_Simpson_rho` as integrand functions for a new adaptive integration.

3.3 Standard Simpson integration

The functions described in this section can be found in the source file `DFT_derivatives.cpp`.



`Simpson_rho_dX` and `Simpson_z_dX`: as for the functions implementing the adaptive integration, these functions provide integrals along the radial direction ρ and the internuclear direction z. We report an extract of code in which the Simpson integral over z is computed:

```
/**** Simpson integration ****/
double z = 0.0, Simpson_sum = 0.0, f_i = 0.0, f_i_plus_1 = 0.0;
for(i=1; i<=N_mesh-1; i=i+2){

 /**** The z coordinate starts at a finite value ****/
    z = a + i*dz;
    f_i = Simpson_rho_dX(z, alpha, beta, R_A, R_B, c, X);
    f_i_plus_1 = Simpson_rho_dX(z + dz, alpha, beta, R_A, R_B, c, X);
    Simpson_sum = Simpson_sum + (4.*f_i + 2.*f_i_plus_1);
}

Simpson_sum = (Simpson_sum + f_a + f_b)*(dz/3.);
return Simpson_sum;
```

3.4 Solving the Roothan problem (BOMD only)

The following functions are in the `solving_Roothan.cpp` source file.

`create_eval_evec`: it computes the eigenvalues and eigenvectors of the real symmetric matrix `A`. The matrix `B` is filled with the eigenvectors. The diagonal and lower triangular part of `A` are destroyed during the computation.

The eigenvectors are normalised to unit magnitude.

`diag_S`: diagonalises $\mathcal{S}$ and computes the matrix $V = U\mathcal{S}^{-1/2}$, where U is the eigenvector matrix of $\mathcal{S}$. The V matrix is such that $V^\top \mathcal{S} V = \mathbb{I}_d$.
`solve_FC_eSC`: it computes the transformed Fock matrix $\mathcal{F}' = V^\top \mathcal{F} V$ and it diagonalise it $\mathcal{F}'\mathbf{C}' = \varepsilon \mathbf{C}'$. This last problem is equivalent to the Roothan problem $\mathcal{F}V\mathcal{C}' = \varepsilon \mathcal{S} V \mathbf{C}'$, where $\mathbf{C} = V\mathbf{C}'$.

The user can obtain the energy profile either in HF or DFT by using the options `SC_HF` and `SC_DFT` (`SC` stands for self-consistent).

3.5 Evolution of C (CPMD only)

The functions listed in this section are reported in the source file `Cs_evol.cpp`. These functions represent the electronic cycle in the Car-Parrinello part of the code.

We report the scheme of the algorithm:

Algorithm 1 Evolution of $\mathbf{C}$

while $(t <= h_N)$ **do**
  Compute $\mathcal{F}$
  Find $\widetilde{\mathbf{C}}(t+h) = [2(m-h^2\mathcal{F})\mathbf{C}(t) - (m-h\gamma)\mathbf{C}(t-h)]/(m+h\gamma)$
  Define $\widetilde{\lambda} = \frac{2h^2\lambda}{m+h\gamma}$
  Solve $[\widetilde{\mathbf{C}}(t+h) - \widetilde{\lambda}\mathcal{S}\mathbf{C}(t)]^\top \mathcal{S}[\widetilde{\mathbf{C}}(t+h) - \widetilde{\lambda}\mathcal{S}\mathbf{C}(t)] = 1$ in $\widetilde{\lambda}$
  Set $\mathbf{C}(t+h) = \widetilde{\mathbf{C}}(t+h) - \widetilde{\lambda}\mathcal{S}\mathbf{C}(t)$
  Set $t = t+h$
end while

This loop is implemented inside the programs `CPMD_HF` as follows:

```
/**** Evolve the c vector of coefficients ****/
double n_times = h_N/h;
for(int k=0; k<n_times; k++){
    two_body_F(Q, c, F);
    gsl_matrix_add(F, H);
    lambda = update_c(F, S, c, c_old);
}
```

Similarly, the same loop is implemented also for `CPMD_DFT`, but the XC part is added:

```
/**** Evolve the c vector of coefficients ****/
double n_times = h_N/h;
for(int k=0; k<n_times; k++){
    two_body_F(Q, c, F);
    gsl_matrix_scale(F, 1. + a_x);

    /**** The XC part has to be recomputed because it strictly depends on the vector c ****/
    Adaptive_Ex_Corr(V_xc, dVxc_dX, R_A, R_B, c, X[n], s);
    gsl_matrix_add(F, H);
    gsl_matrix_add(F, V_xc);
    lambda = update_c(F, S, c, c_old);
}
```

`partial_evolution`: it computes the first line of 1, in which the Lagrange multiplier λ is not considered.

`lowest_positive_root`: it calls the function `solve_eq2degree`, which solves in $\widetilde{\lambda}$ the equation in the third line of 1. It saves the two possible solutions $\widetilde{\lambda}_1$ (in `sol_1`) and $\widetilde{\lambda}_2$ (in `sol_2`) and returns the lowest positive one.

```
double sol_1 = solve_eq2degree(a, b, c, 0);
double sol_2 = solve_eq2degree(a, b, c, 1);
if((sol_1 < 0 & sol_2 > 0) || (sol_1 > 0 && sol_2 > sol_1)){
    sol = sol_1;
}
if((sol_2 < 0 & sol_1 > 0) || (sol_2 > 0 && sol_1 > sol_2)){
    sol = sol_2;
}

if(sol_1 == 0 && sol_2 == 0){
    cout << "Problem in 2nd order equation ! " << endl;
}
return sol;
```

`Get_A`, `Get_B`, `Get_C`: these functions return the coefficients A, B, C of the second degree equation $A\lambda^2 + B\lambda + C = 0$.

`update_c`: it updates $\mathbf{C}$ as in the last line of 1 and it returns λ, which msut be used also in the nuclear equations.

By specifying the option `Evolve_coefficients`, the user can obtain a file named `Energies_C_evolution.txt` which contains the evolution of the energy in the electronic cycle.

3.6 Conjugate gradient routines

These functions can be found in the source file `CG_routine.cpp`.

We report the algorithmic scheme of conjugate gradient:

Algorithm 2 Conjugate Gradient

while $(\mathbf{r}_{k-1}^\top \mathbf{r}_{k-1} > \varepsilon = 10^{-7})$ **do**
  $\alpha_k = \mathbf{r}_{k-1}^\top \mathbf{r}_{k-1} / \mathbf{d}_{k-1}^\top \mathbf{H}\mathbf{d}_{k-1}$
  $\Delta\mathbf{C}_k = \Delta\mathbf{C}_{k-1} + \alpha_k \mathbf{d}_{k-1}$
  $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{H}\mathbf{d}_{k-1}$
  $\beta_k = \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{r}_{k-1}^\top \mathbf{r}_{k-1}$
  $\mathbf{d}_k = \mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$
  $k = k+1$
end while

`Get_Hessian_and_b`: it computes the Hessian matrix $\mathbf{H} = 2\mathcal{F} + 2(\boldsymbol{\nabla}_\mathbf{C}\mathcal{F})\mathbf{C} - 2\lambda\mathcal{S} = 2\mathcal{F} + 2\mathbf{C}^\top \mathcal{Q}\mathbf{C} - 2\lambda\mathcal{S}$ and the vector $\mathbf{b} = -2\mathcal{F}\mathbf{C} + 2\lambda\mathcal{S}\mathbf{C}$.
`Conj_grad`: it contains the algorithm in 2:

```
while(sqrt(norm) > tol){
    gsl_blas_ddot(r, r, &norm);

    /**** Update the Delta_C ****/
    alpha = Get_alpha(Hessian, r, d);
    gsl_vector_scale(d, alpha);
    gsl_vector_add(Delta_c, d);
    gsl_vector_scale(d, 1./alpha);

    /**** Update the remainder r ****/
    gsl_blas_dgemv(CblasNoTrans, 1., Hessian, d, 0., Hd);
    gsl_vector_scale(Hd, alpha);
    gsl_vector_add(r, Hd);
    beta = Get_beta(norm, r);
    gsl_vector_scale(d, beta);
    gsl_vector_sub(d, r);

    iter = iter + 1;
}
```

Note that the Hessian matrix $\mathbf{H}$ (`Hessian` in the code) and the vector $\mathbf{b}$ (`b` in the code) are not affected by the cycle in 2, as the algorithm updates only the increment $\Delta\mathbf{C}$ and not the value of $\mathbf{C}$ itself.

This function is called subsequently in the `main_H2.cpp` file in the options `CGMD_HF` and `CGMD_DFT`.

4 Other options

4.1 The `EX_CORR` option

When this option is specified, the program returns a the $\mathcal{V}^{xc}$ matrix computed with both standard and adaptive Simpson integrations. Below, an example of output.

```
Matrix V_xc computed with STANDARD SIMPSON:
-0.00271121   -0.00612233   -0.007163   -0.00742275   -2.29877e-07   -0.000528754   -0.0038674   -0.00624256
-0.00612233   -0.0429401   -0.0861228   -0.10544   -0.000528754   -0.0106887   -0.0519861   -0.0899596
-0.007163   -0.0861228   -0.337474   -0.603313   -0.0038674   -0.0519861   -0.252473   -0.535806
-0.00742275   -0.10544   -0.603313   -1.61919   -0.00624256   -0.0899596   -0.535806   -1.51702
-2.29877e-07   -0.000528754   -0.0038674   -0.00624256   -0.00271121   -0.00612233   -0.007163   -0.00742275
-0.000528754   -0.0106887   -0.0519861   -0.0899596   -0.00612233   -0.0429401   -0.0861228   -0.10544
-0.0038674   -0.0519861   -0.252473   -0.535806   -0.007163   -0.0861228   -0.337474   -0.603313
-0.00624256   -0.0899596   -0.535806   -1.51702   -0.00742275   -0.10544   -0.603313   -1.61919

**************************************************************************

Matrix V_xc computed with ADAPTIVE SIMPSON:
-0.00271118   -0.00612229   -0.00715887   -0.00741789   -5.91488e-09   -0.000528742   -0.00386476   -0.006241
-0.00612229   -0.0429463   -0.0860938   -0.105494   -0.000528742   -0.0106993   -0.0519743   -0.0899581
-0.00715887   -0.0860938   -0.335202   -0.603305   -0.00386476   -0.0519743   -0.252855   -0.535916
-0.00741789   -0.105494   -0.603305   -1.60289   -0.00624194   -0.0899581   -0.535916   -1.52353
-5.91488e-09   -0.000528742   -0.00386476   -0.00624194   -0.00271118   -0.00612229   -0.00715887   -0.007417
-0.000528742   -0.0106993   -0.0519743   -0.0899581   -0.00612229   -0.0429463   -0.0860938   -0.105494
-0.00386476   -0.0519743   -0.252855   -0.535916   -0.00715887   -0.0860938   -0.335202   -0.603305
-0.00624194   -0.0899581   -0.535916   -1.52353   -0.00741789   -0.105494   -0.603305   -1.60289

Density written successfully in a txt file
Density derivative written successfully in a txt file
Density derivative w.r.t z written successfully in a txt file
```

Notice that the matrices respect the symmetry

$$\left|\begin{array}{c|c} AA & AB \\ \hline BA & BB \end{array}\right| \tag{1}$$

As already mentioned in the principal document, the only critical integral is the `[0][0]` element of each submatrix. For the above matrix obtained with adaptive integration we used the following set of thresholds:

```
double eps[4][4] = {
    {1E-9, 1E-7, 1E-6, 1E-6},
    {1E-7, 1E-5, 1E-5, 1E-4},
    {1E-6, 1E-5, 1E-4, 1E-4},
    {1E-6, 1E-4, 1E-4, 1E-3}
};
```

however, since the correction is minimal, we employed 10^-6 for the `[0][0]` and 10^{-6} for the `[0][1]` elements.

5 The `Scal_prod` option

When the `Scal_prod` is called, the user must specify two choices among `BO`, `CG`, `CP`. The program then writes a file in which the scalar product $\mathbf{C}^\top \mathcal{S}\mathbf{C}$ is computed.

he first column is the projection of the second choice on the first choice motion, while the second column is the projection of the first choice on the second choice motion.

For instance, $\mathbf{C}_{BO}^\top \mathcal{S}_{BO}\mathbf{C}_{CP}$ is the reported in the second column of the file when the options `CP` and `BO` are specified.

5.1 Legend

Piece of useful information.

Important observations.