2.15 The SHARK Integral Package and Task Driver

2.15.1 Preface

Starting with ORCA 5.0 very large changes have taken place in the way that the program handles integrals and integral related tasks like building Fock matrices. SHARK is a powerful and efficient infrastructure that greatly facilitates the handling of these tasks. This allows developers to write highly streamlined code with optimal performance and a high degree of reliability. Compared to the way ORCA handled integrals before ORCA 5.0, tens of thousands of lines of codes, often duplicated or nearly duplicated from closely related parts of the program could be eliminated. From the perspective of the user, the visible changes to the input and output of the program compared to ORCA 4.2.1 and earlier are relatively limited. However, under the hood, the changes are vast and massive and will ensure that ORCA's infrastructure is modern and very well suited for the future of scientific computing.

The benefits of SHARK for the users of ORCA are:

1. Improved code efficiency that is consistent through all program tasks. In particular, complicated two-electron integrals, for example in the context of GIAOs, two-electron spin-orbit coupling and two-electron spin-spin coupling integrals are handled with vastly improved efficiency. Also, integral digestion has been vastly improved with very large benefits for calculations that build many Fock matrices at a time, for example in CIS/TD-DFT, analytic Hessians or response property calculations.
2. Improved code reliability, since all integrals now run through a well debugged, common interface
3. Shorter development times. The new infrastructure is so user friendly to programmers that writing new code that makes use of SHARK is much faster than in the past.
4. SHARK handles basis sets much better than the old infrastructure. Whether the basis sets used follow a segmented contraction, general contraction or partial general contraction is immaterial since the algorithms have been optimized carefully for each kind of basis throughout.

2.15.2 The SHARK integral algorithm

One cornerstone of SHARK is a new integral algorithm that allows for highly efficient evaluation of molecular integrals. The algorithm is based on the beautiful McMurchie-Davidson algorithm which leads to the following equation for a given two-electron integral:

$$(\mu_A \nu_B|\kappa_C \tau_D) = C \sum_{tuv} E_t^{\mu\nu;x} E_u^{\mu\nu;y} E_v^{\mu\nu;z} \sum_{t'u'v'} E_{t'}^{\kappa\tau;x} E_{u'}^{\kappa\tau;y} E_{v'}^{\kappa\tau;z} (-1)^{t'+u'+v'} R_{t+t',u+u',v+v'}$$

Here

$$C = 8\pi^{5/2} = 139.947346620998902770103$$

and the primitive Cartesian Gaussian basis functions $\{\mu_A\}$ where A is the atomic center, where basis function μ is centered at position $\mathbf{R}_A$. In order to catch a glimpse of what the McMurchie-Davidson algorithm is about, consider two unnormalized, primitive Gaussians centered at atoms A and B, respectively:

$$G_A = x_A^i y_A^j z_A^k \exp(-\alpha R_A^2)$$
$$G_B = x_B^{i'} y_B^{j'} z_B^{k'} \exp(-\beta R_B^2)$$

By means of the Gaussian product theorem, the two exponentials are straightforwardly rewritten as:

$$\exp(-\alpha R_A^2)\exp(-\beta R_B^2) = K_{AB} \exp\Big(-(\alpha+\beta) r_P^2\Big)$$

With

$$K_{AB} = \exp\left(-\tfrac{\alpha\beta}{\alpha+\beta}|\mathbf{R}_A - \mathbf{R}_B|^2\right)$$

$r_P^2 = |\mathbf{r} - \mathbf{R}_P|^2$ is the electronic position relative to the point

$$\mathbf{R}_P = \tfrac{\alpha}{\alpha+\beta}\mathbf{R}_A + \tfrac{\beta}{\alpha+\beta}\mathbf{R}_B$$

at which the new Gaussian is centered. The ingenious invention of McMurchie and Davidson was to realize that the complicated polynomial that arises from multiplying the two primitive Cartesian Gaussians can be nicely written in terms of Hermite polynomials $\{\Lambda\}$. In one dimension:

$$x_A^i x_B^{i'} = \sum_{t=0}^{i+i'} E_t$$

And hence:

$$G_A G_B = K_{AB} \sum_{t=0}^{i+i'} E_t^{AB} \sum_{u=0}^{j+j'} E_u^{AB} \sum_{v=0}^{k+k'} E_v^{AB} \Lambda_{tuv}^{AB}$$

With

$$\Lambda_{tuv}^{AB} = \left(\frac{\partial}{\partial X_P}\right)^t \left(\frac{\partial}{\partial Y_P}\right)^u \left(\frac{\partial}{\partial Z_P}\right)^v \exp\Big(-(\alpha+\beta)R_P^2\Big)$$

This means that the original four center integral is reduced to a sum of two-center integrals over Hermite Gaussian functions. These integrals are denoted as

$$R_{t+t',u+u',v+v'} = \int\int \Lambda_{tuv}^{AB}(\mathbf{r}_1;\mathbf{R}_P)\Lambda_{t'u'v'}^{CD}(\mathbf{r}_2;\mathbf{R}_Q) r_{12}^{-1} d\mathbf{r}_1 d\mathbf{r}_2$$

With these definitions one understands the McMurchie Davidson algorithm as consisting of three steps:

1. Transformation of the Bra function product into the Hermite Gaussian Basis
2. Transformation of the Ket function product into the Hermite Gaussian Basis
3. Calculation of the Hermite Gaussian electron repulsion integral

SHARK is the realization that these three steps can be efficiently executed by a triple matrix product:

$$(\mu_A \nu_B|\kappa_C \tau_D) = \left(\mathbf{E}^{\text{bra}}\mathbf{R}\mathbf{E}^{\text{ket}}\right)_{\mu\nu,\kappa\tau}$$

Here $\mathbf{E}^{\text{bra}}$ and $\mathbf{E}^{\text{ket}}$ collect the E coefficients for all members of the shell product on the bra and ket side ($E_{\mu\nu,tuv}^{\text{bra}}$ and $E_{\kappa\tau,tuv}^{\text{ket}}$), respectively, and **R** collects the integrals over Hermite Gaussian functions ($R_{tuv,t'u'v'}$).

There are many benefits to this formulation:

1. The integral is factorized allowing steps to be performed independent of each other. For example, the E matrices can be calculated at the beginning of the calculation and reused whenever needed. Their storage is unproblematic
2. Matrix multiplications lead to extremely efficient formation of the target integrals and drive the hardware at peak performance
3. Steps like contraction of primitive integrals and transformation from the Cartesian to the spherical Harmonics basis can be folded into the definition of the E matrices thus leading to extremely efficient code with next to no overhead creates by short loops.
4. Programming integrals becomes very easy and efficient. Other types of integrals as well as derivative integrals are readily approached in the same way. Also, two- and three-index repulsion integrals, as needed for the RI approximation are also readily formulated in this way.
5. One-electron integrals are equally readily done with this approach.

There is a very large number of technicalities that we will not describe in this manual which is only intended to provide the gist of the algorithm.

2.15.3 SHARK and libint

Up to ORCA 4.2.1, ORCA has almost entirely relied on the libint2 integral library which is known to be very efficient and powerful. Starting from ORCA 5.0, both SHARK and libint are used for integral evaluations and libint is fully integrated into the SHARK programming environment. Integrals that are only available in one of the packages are done with this package (e. g. GIAO, SOC and Spin-Spin integrals in SHARK; F12 or second derivative integrals in libint). For the integrals available in both packages, the program makes a judicious choice about the most efficient route. The reason for this hybrid approach is the following:

The SHARK integral algorithm is at its best for higher angular momentum functions ($l > 2$; d-functions) which is where the efficiency of the matrix multiplications leads to very large computational benefits. Integrals over, say, four f- or g-functions perform much faster (up to a factor of five) than with traditional integral algorithms. However, for low angular momenta, there is overhead created by the matrix multiplications and also by the fact that the McMurchie Davidson algorithm is known to not be the most FLOP count efficient algorithm. To some extent, this is take care of by using highly streamlined routines for low angular momenta that perform extremely well. However, there are penalties for intermediate angular momenta, where the efficiency of the matrix multiplications has not set in and the integrals are too complicated for hand coding. These integrals perform best with libint and consequently, the program will, by default, select libint to perform such integral batches.

2.15.4 Basis set types

One significant aspect of molecular integral evaluation is the type of contraction that is present in a Gaussian basis set. The most general type of basis set is met in the "general contraction" scheme. Here all primitive Gaussian basis functions of a given angular momentum are collected in a vector $\{\phi\}$. In general, all primitives will contribute to all basis functions $\{\varphi\}$ of this same angular momentum. Hence, we can write:

$$\begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \vdots \\ \varphi_{N_l} \end{pmatrix} = \begin{pmatrix} d_{11} & d_{11} & \cdots & d_{1M_l} \\ d_{21} & d_{21} & \cdots & d_{2M_l} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N_l1} & d_{N_l2} & \cdots & d_{N_lM_l} \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{M_l} \end{pmatrix}$$

Where N_l and M_l are the number of actual basis functions and primitives respectively. Typically, the number of primitives is much larger than the number of basis functions. The matrix **d** collects the contraction coefficients for each angular momentum. Typical basis sets that follow this contraction pattern are atomic natural orbital (ANO) basis sets. They are typically based on large primitive sets of Gaussians. Such basis sets put very demands on the integral package since there are many integrals over primitive Gaussian basis functions that need to be generated. If the integral package does not take advantage of the general contraction, then this integral evaluation will be highly redundant since identical integrals will be calculated N_l times (and hence, integrals over four generally contracted shells will be redundantly generated N_l^4 times). SHARK takes full advantage of general contraction for all one- and two-electron integrals that it can generate. Here, the unique advantages of the integral factorization come to full benefit since all integral quadruples of a given atom quadruple/angular momentum quadruple can be efficiently generated by just two large matrix multiplications.

The opposite of general contraction is met with segmented contraction. Here each basis function involves a number of primitives:

$$\varphi_\mu = \sum_k d_{k\mu}\phi_k$$

Quite typically, none of the ϕ_k that occur in the contraction of one basis functions occurs in any other basis function. Typical basis sets of this form are the "def2" basis sets of the Karlsruhe group. They are readily handled by most integral packages and both SHARK and libint are efficient in this case.

The third class of basis sets is met, when general contraction is combined with segmented contraction. Basis sets of this type are, for example, the correlation consistent (cc) basis sets. We call such basis sets "partially generally contracted". In such basis sets, part of the basis functions are generally contracted (for example, the s- and p-functions in main group elements), while other basis functions (e. g. polarization functions, diffuse functions, core correlation functions) are not generally contracted. It is difficult to take full advantage of such basis sets given their complicated structure. In ORCA 5, special code has been provided that transforms the basis set into an intermediate basis set that does not contain any redundancies and hence drives SHARK or libint at peak performance.

In assessing the efficiency vs the accuracy of different integral algorithms, it is clear that segmented basis sets lead to the highest possible efficiency if they are well constructed. For such basis set the pre-screening that is an essential step of any integral direct algorithm performs best. The highest possible accuracy (per basis functions) is met with generally contracted basis sets. However, here the pre-screening becomes rather inefficient since it can only be performed at the level of atom/angular momentum combinations rather than individual shell quadruples. Thus, as soon as a given atom/angular momentum combination leads to any non-negligible integral, all integrals for this combination need to be calculated. This created a sizeable overhead. Consequently, SCF calculations can never be as efficient as with segmented basis sets. If this is immaterial, for example, because a subsequent coupled cluster or MRCI calculation is dominating the calculation time, general contraction is very worthwhile to be explored. For partial general contraction, our algorithm performs very nearly as efficiently as for segmented contraction in SCF calculations. However, since the intermediate basis set is larger than the original orbital basis, certain limited performance penalties can arise in some job types.

2.15.5 Task drivers

In traditional algorithms, quantum chemical programs frequently contain many instances of nested loops over basis function shells, the integral package is called and the integrals are “digested” for a given task. While these steps are inevitable, programming them repeatedly is laborious and error prone. In addition, improvements, say in the handling of contractions or symmetry, need to implemented in many different places. In the SHARK infrastructure all of this is unnecessary since it is programmed in an object-oriented fashion, where the programmer does not need to take care of any detail. Hence, developers only need to write short code sections that distribute the generated integrals into whatever data structure they need, while the SHARK interface takes care of all technical aspects and triggers the sophisticated and efficient machinery that underlies it.

Given this situation, the future of ORCA will involve SHARK taking care of nearly of the compute intensive, laborious tasks, while ORCA will organize and trigger all of these tasks. ORCA and SHARK communicate via a lean and well-defined interface to exchange the necessary data. In this way, a modern, efficient, easy to use and readily maintainable development environment is created.

2.15.6 SHARK User Interface

While SHARK is a large and complicated machinery, we have deliberately kept the interface as straightforward and simple as possible.

In the simple input line there are keywords to turn SHARK on (default) or off:

```
! UseShark
! NoUseShark
```

Deprecated since version 6.0.0: The legacy (non-SHARK code) was removed from the program, so SHARK can no longer be disabled in calculations, which require integrals (as opposed to MM calculations, for example).

Further, there are a few flags that can be set in the `%shark` block:

```
%shark
 UseGeneralContraction false  # turns general contraction algorithm on or
                              # off. There normally is no need to set this
                              # flag since the program will find the
                              # contraction case automatically
 Printlevel 1                 # Amount of output generated. Choose 0 to
                              # suppress output and 2 for more output.
                              # Everything else is debug level printing and
                              # will fill your harddrive very quickly with
                              # unusable information
 PartialGCFlag -1             # Let the program decide whether to use PGC
                0             # do not use it
                1             # Enforce PGC (even for ANO bases)
 FockFlag SHARK_libint_hybrid # default: best of both worlds
          force_shark         # Force Shark where possible
```

(continues on next page)

(continued from previous page)

```
         force_libint        # Force libint where possible
 RIJFlag  RIJ_Auto            # default: program decides the best way
          Split_rij           # new SHARK Split-RI-J algorithm
          Split_rij_2003      # Highly efficient re-implementation of the
                              # Original 2003 algorithm. Mostly used!
          rij_regular         # Use traditional 3 center integrals
                              # (not recommended)
end
```

2.16 SCF Stability Analysis

The SCF stability will give an indication whether the SCF solution is at a local minimum or a saddle point.[173, 174] It is available for RHF/RKS and UHF/UKS. In the latter case, the SCF is restarted by default using new unrestricted start orbitals if an instability was detected. For a demonstration, consider the following input:

```
! BHLYP def2-SVP NORI

%scf
 guess hcore
 HFTyp UHF
 STABPerform true
end

* xyz 0 1
h 0.0 0.0 0.0
h 0.0 0.0 1.4
*
```

The HCORE guess leads to a symmetric/restricted guess, which does not yield the unrestricted solution. The same is often true for other guess options. For more details on the stability analysis, see Section *SCF Stability Analysis*.

The SCF stability analysis evaluates the electronic Hessian (with respect to orbital rotations) at the point indicated by the SCF solution to determine the lowest eigenvalues of the Hessian. If one or more negative eigenvalues are found, the SCF solution corresponds to a saddle point and not a true local minimum in the space considered in the analysis. A typical case are stretched bonds of diatomics, where the symmetry of the initial guess leads to a restricted solution instead of the often preferred unrestricted one. Several spaces are theoretically possible[173]. However, ORCA limits itself to the analysis RHF/RKS in the space of UHF/UKS or UHF/UKS in the space of UHF/UKS. As such, it is on the available for the SCF parts of DFT and HF.[174] In the following, HF is used to indicate both HF and KS. Consider the following input (unless indicated otherwise, default values are shown):

```
! BHLYP def2-SVP NORI

%scf
 guess hcore # for illustrative purposes only
 HFTyp UHF # default based on spin multiplicity
 STABPerform true # default false
 STABRestartUHFifUnstable true # restart the UHF-SCF if unstable
 STABNRoots 3                  # number of eigenpairs sought
 STABMaxDim 3                  # Davidson expansion space = MaxDim * NRoots
 STABMaxIter 100               # maximum number of Davidson iterations
 STABNGuess 4096               # size of initial guess matrix: 4096 x 4096
 STABDTol 0.0001               # convergence criterion from iteration to iteration
 STABRTol 0.0001               # convergence criterion max residual norm
 STABlambda +0.5               # mixing parameter
 STABORBWIN -1, -1, -1, -1,  -1, -1, -1, -1 # defines the donor / acceptor spaces
                               # 4 parameters for RHF
                               # 8 paramters for UHF (4 alpha, 4 beta)
                               # orbital window, -1 refers to automatic␣
```

(continues on next page)

(continued from previous page)

```
↪determination
 STABEWIN -5.0, 5.0            # lower and upper cutoff in Eh for automatic␣
↪freezing
 #------------------------------------------------------------------------------
 # alternative specification using a sub-block:
 stab
  NRoots 3
  MaxDim 3 # etc.
 end
end

* xyz 0 1
h 0.0 0.0 0.0
h 0.0 0.0 1.4
*
```

The determination of the electronic Hessian is structurally comparable to the TDHF/CIS/TDDFT procedure. Thus, many options are very similar and the user is encouraged to read the section on TDDFT (Section *Excited States via RPA, CIS, TD-DFT and SF-TDA*) to clarify some of the options given here. Since one is usually only interested in the qualitative determination "stable or not?", three roots should be sufficient to find the lowest eigenvalue. By the same philosophy, `StabMaxDim`, `StabMaxIter`, `StabNGuess` and the convergence criteria were chosen. The parameter `StabLambda` refers to the λ of equation 37 of reference [173], which determines the mixing of the original SCF solution and the new orbitals to yield a new guess. Choosing this value is not trivial, since positive and negative values can lead to different new solutions (at least in principle). The convergence of the ensuing SCF depends on it, as well, since all SCF procedures require a sufficiently good guess to converge in a decent number of iterations (or even at all).

The orbital window and the energy window can be specified. Note that the `StabEWIN` will be overridden by the appropriate `StabORBWIN` values. The automatic determination is also influenced by the `%method FrozenCore` settings. Tests have shown that significant curtailing of the actual orbital window can drastically influence the results to the point of qualitative failure.

Current limitations on the method are:

- Only single-point-like calculations are supported. For geometry optimizations etc., one must use the guess MORead feature *Choice of Initial Guess and Restart of SCF Calculations* to employ the guess obtained here. Likewise, one must extract a geometry and run a separate calculation if one is interested in the SCF stability.
- As for TDDFT, NORI, RIJONX, and RIJCOSX are supported. RI-JK is not supported.
- Other, more advanced features like finite-temperature calculations and relativistic calculations (beside ECPs) are not possible at this time.

Overall, the user is cautioned against using the stability analysis blindly without critically evaluating the result in terms of energy difference and by investigating the orbitals (by the printout or by plotting). Its usefulness cannot be denied, but it is certainly not black-box.

An SCF stability analysis with default settings can be requested via STABILITY, SCFSTABILITY, SCFSTAB or STAB on the simple input line.

2.17 Finite Electric Fields

Electric fields can have significant influences on the electronic structure of molecules. In general, when an electric field is applied to a molecule, the electron cloud of the molecule will polarize along the direction of the field. The redistribution of charges across the molecule will then influence the wavefunction of the molecule. Even when polarization effects are not significant, the electric field still exerts a drag on the negatively and positively charged atoms of the molecule in opposite directions, and therefore affect the orientation and structure of the molecule. The combination of electrostatic and polarization effects make electric fields a useful degree of freedom in tuning e.g. reactivities, molecular structures and spectra [175]. Meanwhile, the energy/dipole moment/quadrupole moment changes of the system in the presence of small dipolar or quadrupolar electric fields are useful for calculating many electric properties of the system via numerical differentiation, including the dipole moment, quadrupole moment, dipole-dipole polarizability, quadrupole-quadrupole polarizability, etc. Such finite difference property calculations can be conveniently done using compound scripts in the ORCA Compound Scripts Repository.

An overview of relevant keywords is given in Table 2.59.

2.17.1 Dipolar Electric Fields

A uniform or equivalently speaking, dipolar electric field can be added to a calculation via the `EField` keyword in the `%scf` block:

```
%scf
  EField 0.1, 0.0, 0.0 # x, y, z components (in au) of the electric field
end
```

Although the keyword is in the `%scf` block, it applies the electric field to all other methods (post-HF methods, multireference methods, TDDFT, etc.) as well. Analytic gradient contributions of the electric field are available for all supported methods that already support analytic gradients, but analytic Hessian contributions are not.

Warning

The electric field functionality is not available for *GFN-xTB* and force field methods (as well as any method that involves GFN-xTB or force fields, e.g. *QM/XTB* and *QM/MM*! Combination with these method will result in an abort.

The sign convention of the electric field is chosen in the following way: suppose that the electric field is generated by a positive charge in the negative z direction, and a negative charge in the positive z direction, then the z component of the electric field is positive. This convention is consistent with most but not all other programs [175], so care must be taken when comparing the results of ORCA with other programs.

Another important aspect is the gauge origin of the electric field. The gauge origin of the electric field is the point (or more accurately speaking, one of the points - as there are infinitely many such points) where the electric potential due to the electric field is zero. Different choices of the gauge origin do not affect the geometry and wavefunction of the molecule, as they do not change the electric field felt by the molecule, but they do change the energy of the molecule. The default gauge origin is the (0,0,0) point of the Cartesian coordinate system, but it is possible to choose other gauge origins via the `EFieldOrigin` keyword in the `%scf` block.

```
%scf
 EFieldOrigin CenterOfMass      # use center of mass
              CenterOfNucCharge # use center of nuclear charge
              0.0, 0.0, 0.0     # use given X,Y,Z as origin (default: 0,0,0)
                                # in the units chosen for the coordinates␣
↪(Angstrom/Bohr)
end
```

Note the default gauge origin of the electric field is different from the default gauge origin of the ELPROP module, which is the center of mass. If the user chooses the center of mass/nuclear charge as the gauge origin of the electric field, the gauge origin will move as the molecule translates; this has important consequences. For example, in an MD simulation of a charged molecule in an electric field, the molecule will not accelerate, unlike when `EFieldOrigin` is fixed at a given set of coordinates, where the molecule will accelerate forever. In general, `CenterOfMass` and `CenterOfNucCharge` are mostly suited for the finite difference calculation of electric properties, where one frequently wants to choose the center of mass or nuclear charge as the gauge origin of the resulting multipole moment or polarizability tensor. Instead, a fixed origin is expected to be more useful for simulating the changes of wavefunction, geometry, reactivity, spectra etc. under an externally applied electric field, as experimentally the electric field is usually applied in the lab frame, rather than the comoving frame of the molecule.

2.17.2 Quadrupolar Electric Fields

Similar to dipolar electric fields, quadrupolar fields can be added via the `QField` keyword in the `%scf` block:

```
%scf
  QField 0.1, 0.0, 0.0, 0.05, 0.0, 0.0 # xx, yy, zz, xy, xz, yz components (in au)
                                        # of the quadrupolar field
end
```

The gauge origin of the quadrupolar field is the same as that of the *dipolar electric field*.

2.17.3 Combination of Dipolar and Quadrupolar Electric Fields

Dipolar and *quadrupolar electric fields* can be combined using the respective `EField` and `QField` keywords in the `%scf` block. This allows one to simulate a gradually varying electric field, for example the following input specifies an electric field that has a strength of 0.01 au at the gauge origin ((0,0,0) by default), pointing to the positive z direction, and increases by 0.001 au for every Bohr as one goes in the positive z direction:

```
%scf
  EField 0.0, 0.0, 0.01
  QField 0.0, 0.0, 0.001, 0.0, 0.0, 0.0
end
```

As a second example, one can also simulate an ion trap:

```
%scf
  QField -0.01, -0.01, -0.01, 0.0, 0.0, 0.0
end
```

Under this quadrupolar field setting, a particle will feel an electric field that points towards the gauge origin, whose strength (in au) is 0.01 times the distance to the gauge origin (in Bohr). This will keep cations close to the origin, but pushes anions away from the origin. Unfortunately, there is no analytic gradient available for quadrupolar fields.

Notes on Electric Fields

- An au (atomic unit) is a fairly large unit for electric fields: 1 au = 51.4 V/Angstrom. By comparison, charged residues in proteins, as well as scanning tunneling microscope (STM) tips, typically generate electric fields within about 1 V/Angstrom; electrode surfaces usually generate electric fields within 0.1 V/Angstrom under typical electrolysis conditions [175]. If the molecule is not close to the source of the electric field, it is even harder to generate strong electric fields: for example, a 100 V voltage across two metal plates that are 1 mm apart generates an electric field of merely 10^{-5} V/Angstrom. Therefore, if experimentally a certain strength of homogeneous electric field seems to promote a reaction, but no such effect is found in calculation, please consider the possibility that the experimentally observed reactivity is due to a strong local electric field near the electrode surface (that is much higher than the average field strength in the system), or due to other effects such as electrolysis. Conversely, if you predict a certain molecular property change at an electric field strength of, e.g. > 0.1 au, it may be a non-trivial question whether such an electric field can be easily realized experimentally.

- The electric field breaks the rotational symmetry of the molecule, in the sense that rotating the molecule can change its energy. Therefore, geometry optimizations in electric fields cannot be done with internal coordinates. When the user requests geometry optimization, the program automatically switches to Cartesian coordinates if it detects an electric field. While Cartesian coordinates allow the correct treatment of molecular rotation, they generally lead to poor convergence, so a large number of iterations is frequently necessary. Also, since certain geometry optimization tasks (notably transition state optimization) can currently only be done under internal coordinates, electric fields cannot be used in such types of optimization tasks yet.
- Similarly, when the molecule is charged, its energy is not invariant with respect to translations. However, when there is only a dipolar electric field but no other translational symmetry-breaking forces (quadrupolar field, point charges, wall potentials), a charged molecule will accelerate forever in the field, and its position will never converge. Therefore, for geometry optimizations within a purely dipolar electric field and no wall potentials, we do not allow global translations of the molecule, even when translation can reduce its energy. For MD simulations we however do allow the global translations of the molecule by default. If this is not desired, one can fix the center of mass in the MD run using the `CenterCOM` keyword (section *Run*).
- For *frequency calculations* in electric fields, we do not project out the translational and rotational contributions of the Hessian (equivalent to setting `ProjectTR false` in `%freq`; see *Vibrational Frequencies* for details). Therefore, the frequencies of translational and rotational modes can be different from zero, and can mix with the vibrational modes. When the electric field is extremely small but not zero, the "true" translational/rotational symmetry breaking of the Hessian may be smaller than the symmetry breaking due to numerical error; this must be kept in mind when comparing the frequency results under small electric fields versus under zero electric field (in the latter case `ProjectTR` is by default true). Besides, when the translational and rotational frequencies exceed `CutOffFreq` (which is 1 cm^{-1} by default; see section *Vibrational Frequencies*), their thermochemical contributions are calculated as if they are vibrations.
- While the program allows the combination of electric fields with an *implicit solvation model*, the results must be interpreted with caution, because the solvent medium does not feel the electric field. The results may therefore differ substantially from those given by experimental setups where both the solute and the solvent are subjected to the electric field. If the solvent's response to the electric field is important, one should use an explicit solvation model instead. Alternatively, one can also simulate the electric field in the implicit solvent by adding inert ions (e.g. Na$^+$, Cl$^-$) to the system. Similarly, implicit solvation models cannot describe the formation of electrical double layers in the electric field and their influence on solute properties, so in case electrical double layers are important, *MD simulations* with explicit treatment of the ions must be carried out.
- The electric field not only contributes to the core Hamiltonian, but has extra contributions in GIAO calculations, due to the magnetic field derivatives of dipole integrals. In the case of a dipolar electric field, the GIAO contributions have been implemented, making it possible to study e.g. the effect of electric fields on *NMR shieldings*, and as a special case, nucleus independent chemical shieldings (*NICSs*), which are useful tools for analyzing aromaticity. *Quadrupolar fields* cannot be used in GIAO calculations at the moment.

2.17.4 Keywords

Table 2.59: `%scf` block input keywords and options for finite electric fields.

Keyword	Option	Description
`EField`	`<x, y, z>`	Activates *dipolar electric field* with x, y, z components (in au)
`QField`	`<xx, yy, zz, xz, xz, yz>`	Activates *quadrupolar electric field* with xx, yy, zz, xz, xz, yz components (in au)
`EFieldOri`	`CenterOfMass`	Sets origin to center of mass
	`CenterOfNucCharge`	Sets origin to center of nuclear charge
	`<X, Y, Z>`	Sets origin to X,Y,Z coordinates (default: 0,0,0) in the units chosen for the coordinates (Angstrom/Bohr)

2.18 Fragment Specification

Atoms in a calculation can be grouped into specific *fragments*, which serve multiple purposes. Fragment definitions can be used to *assign Basis Sets and ECPs*, organize output in the *population analysis section*, and enable features like *fragment constrain optimization* and *Rigid Body Optimization*. They are also used in *local energy decomposition* and *multi-level calculations*.

2.18.1 Fragments defined on Input File

There are three ways to assign atoms to fragments using the input file. The first method is to assign a specific atom to a specific fragment by placing `(n)` directly after the atomic symbol in the coordinates section.

```
*xyz -2 2
 Cu(1)  0.00  0.00  0.00
 Cl(2)  2.25  0.00  0.00
 Cl(2) -2.25  0.00  0.00
 Cl(2)  0.00  2.25  0.00
 Cl(2)  0.00 -2.25  0.00
*
```

In this example the fragment feature is used to divide the molecule into a "metal" and a "ligand" fragment and consequently the program will print the metal and ligand charges and populations.

```
----------------------------------------------
CARTESIAN COORDINATES OF FRAGMENTS (ANGSTROEM)
----------------------------------------------

 FRAGMENT 1
  Cu    0.000000    0.000000    0.000000

 FRAGMENT 2
  Cl    2.250000    0.000000    0.000000
  Cl   -2.250000    0.000000    0.000000
  Cl    0.000000    2.250000    0.000000
  Cl    0.000000   -2.250000    0.000000

...

----------------------------------------------
MULLIKEN FRAGMENT CHARGES AND SPIN POPULATIONS
----------------------------------------------
 Fragment   0 :   0.752589     0.842580
 Fragment   1 :  -2.752589     0.157420
Sum of fragment charges         :   -2.0000000
Sum of fragment spin populations:    1.0000000

...

--------------------------------------------
LOEWDIN FRAGMENT CHARGES AND SPIN POPULATONS
--------------------------------------------
 Fragment   0 :   0.222028     0.851552
 Fragment   1 :  -2.222028     0.148448
```

Alternatively, the `%coords` block can be used for fragment definitions in the same way—by placing `(n)` directly after the atomic symbol.

```
%coords
 CTyp   xyz  # the type of coordinates xyz or internal
 Charge -2   # the total charge of the molecule
```

(continues on next page)

(continued from previous page)

```
 Mult   2    # the multiplicity = 2S+1
 coords
    Cu(1)  0.00  0.00  0.00
    Cl(2)  2.25  0.00  0.00
    Cl(2) -2.25  0.00  0.00
    Cl(2)  0.00  2.25  0.00
    Cl(2)  0.00 -2.25  0.00
 end
end
```

Important

- In cases where all atoms are explicitly assigned to fragments, the fragment numbering must start at 1 and use consecutive integers. Non-consecutive or incorrect numbering may lead to errors.
- If any atom is left unassigned (or explicitly assigned to fragment 0), it will be automatically assigned to a fragment using the fragmentation procedure described in *Automatic Fragmentation* section. In such cases—where only a subset of atoms is manually assigned to fragments—it is not necessary to use consecutive fragment numbers. However, the highest fragment number specified in the input must be less than the total number of fragments generated by the combination of manual and automatic procedures. If this condition is not met, ORCA will automatically reorder all fragment numbers in ascending order, starting from 1.

Finally, a third way to define fragments consists of using a `Definition` inside the `%frag` block. In this scheme, the fragment number comes first, followed by a list of atoms (enumerated starting from 0) enclosed in curly brackets `{}` and finishing with `end`. Consecutive atoms can also be specified using the notation `initial_atom:final_atom`.

```
*xyz -2 2
 Cu   0.00  0.00  0.00
 Cl   2.25  0.00  0.00
 Cl  -2.25  0.00  0.00
 Cl   0.00  2.25  0.00
 Cl   0.00 -2.25  0.00
*

%frag
 Definition
  1 {0} end     # atom 0 for fragment 1
  2 {1:4} end   # atoms 1 to 4 for fragment 2
 end
end
```

```
*xyz -2 2
 Cl   2.25  0.00  0.00
 Cl  -2.25  0.00  0.00
 Cu   0.00  0.00  0.00
 Cl   0.00  2.25  0.00
 Cl   0.00 -2.25  0.00
*

%frag
 Definition
  1 {2} end         # atom 2 for fragment 1
  2 {0:1 3:4} end   # atoms 0, 1, 3, and 4 for fragment 2
 end
end
```

Note

- With the last option (`Definition`) the `%frag` block has to be written after the coordinate section.
- `%frag Definition` also works with coordinates that are defined via an external file.

2.18.2 Automatic Fragmentation

Starting with ORCA 6.1, a set of automatic fragmentation algorithms has been introduced to recognize and group atoms into fragments automatically.

The automatic fragmentation procedure is triggered by including a `%frag` block in the input file or when only a subset of atoms has been manually assigned to fragments.

Automatic fragmentation is performed using a set of procedures that can be selected by the user within the `%frag` block. Each procedure attempts to identify new fragments among atoms that have not yet been assigned. For example, consider a case below where the first 11 atoms belong to a propane molecule and the last three belong to a water molecule. The `Water` procedure in `FragProc` identifies the last three atoms as a water molecule and assigns them to the first fragment, while the `FunctionalGroups` procedure detects and assigns the CH3 and CH2 groups of propane as fragments 2 to 4.

```
%frag
   FragProc Water, FunctionalGroups
end

* xyz 0 1
 C     0.000000     1.270000    -0.260000
 C     0.000000    -0.000000     0.580000
 C     0.000000    -1.270000    -0.260000
 H    -0.890000     1.320000    -0.910000
 H     0.890000     1.320000    -0.910000
 H     0.000000     2.180000     0.370000
 H     0.880000    -0.000000     1.250000
 H    -0.880000    -0.000000     1.250000
 H     0.890000    -1.320000    -0.910000
 H     0.000000    -2.180000     0.370000
 H    -0.880000    -1.300000    -0.910000
 H    -0.920000     0.850000    -2.430000
 O    -1.690000     0.830000    -3.000000
 H    -1.640000     1.630000    -3.510000
*
```

```
  ----------------------------------------------
  CARTESIAN COORDINATES OF FRAGMENTS (ANGSTROEM)
  ----------------------------------------------

   FRAGMENT 1
    H    -0.920000     0.850000    -2.430000
    O    -1.690000     0.830000    -3.000000
    H    -1.640000     1.630000    -3.510000

   FRAGMENT 2
    C     0.000000     1.270000    -0.260000
    H    -0.890000     1.320000    -0.910000
    H     0.890000     1.320000    -0.910000
    H     0.000000     2.180000     0.370000

   FRAGMENT 3
    C     0.000000    -1.270000    -0.260000
    H     0.890000    -1.320000    -0.910000
```

(continues on next page)

(continued from previous page)

```
    H      0.000000   -2.180000    0.370000
    H     -0.880000   -1.300000   -0.910000

   FRAGMENT  4
    C      0.000000   -0.000000    0.580000
    H      0.880000   -0.000000    1.250000
    H     -0.880000   -0.000000    1.250000
```

This automatic fragmentation yields the same result as the manual fragment definition shown below, without the need to inspect the geometry and assign fragments manually.

```
%frag
 Definition
 1 { 11:13} end  # water
 2 { 0 3:5} end  # CH3
 3 { 2 8:10} end # CH3
 4 { 1 6:7} end  # CH2
 end
end
```

Note

- Any fragment defined in the *input file* take precedence over automatic assignments.
- ORCA supports up to 10 procedures in `FragProc`, with the full list provided in Table 2.60 and Table 2.61.
- *Constrained Fragments* do not enable automatically the Automatic Fragmentation when there are atoms unassigned to fragments. However, automatic fragmentation can be activated by including a `%frag` block in the input file.

ORCA provides over 30 `FragProc` methods, which can be combined in a list to generate fragments for various purposes. Below are explanations and examples of the different procedures.

Automatic Fragmentation: Connectivity

`FragProc Connectivity` groups atoms that are connected by chemical bonds, estimated based on atomic radii. In the example below, the first nine atoms belong to a dimethyl ether molecule, which is automatically detected and assigned to one fragment, while the last three atoms—belonging to a water molecule—are assigned to a second fragment.

```
%frag
  PrintLevel 3
  FragProc Connectivity
end

*xyz 0 1
 O     0.000000     0.000000     0.000000
 C     0.000000     0.000000     1.380000
 C     1.300000     0.000000    -0.460000
 H    -0.500000     0.870000     1.740000
 H    -0.500000    -0.870000     1.740000
 H     1.000000     0.000000     1.740000
 H     1.300000     0.000000    -1.530000
 H     1.800000     0.870000    -0.100000
 H     1.800000    -0.870000    -0.100000
 H    -1.840000     0.000000    -0.650000
 O    -2.440000     0.000000     0.080000
```

(continues on next page)

(continued from previous page)

```
 H    -3.300000     0.000000    -0.300000
*
```

In this case, the output of the automatic fragmentation tool indicates that two fragments were assigned by the `Connectivity` procedure.

```
------------------------------------------
 Assigned Fragment: 1
------------------------------------------
  Name: Connectivity:0 Method: Connectivity
  Natoms: 9 Charge: 0 Mult: 1

 0 O     0.000000    0.000000    0.000000
 1 C     0.000000    0.000000    1.380000
 2 C     1.300000    0.000000   -0.460000
 3 H    -0.500000    0.870000    1.740000
 4 H    -0.500000   -0.870000    1.740000
 5 H     1.000000    0.000000    1.740000
 6 H     1.300000    0.000000   -1.530000
 7 H     1.800000    0.870000   -0.100000
 8 H     1.800000   -0.870000   -0.100000
------------------------------------------


------------------------------------------
 Assigned Fragment: 2
------------------------------------------
  Name: Connectivity:1 Method: Connectivity
  Natoms: 3 Charge: 0 Mult: 1

 9 H    -1.840000    0.000000   -0.650000
 10 O    -2.440000    0.000000    0.080000
 11 H    -3.300000    0.000000   -0.300000
------------------------------------------
```

Each fragmentation procedure applies only to atoms that have not already been assigned to a fragment. Consequently, connectivity-based fragmentation will ignore any bonds to atoms that have been assigned in the input file or by a previous `FragProc`.

For example, if in the previous case example the oxygen atom is manually assigned to fragment 1 in the input file, this manual assignment takes precedence over the `FragProc Connectivity`. As a result, four fragments are created: one fragment containing the manually assigned oxygen atom, two CH3 groups, and one water molecule.

```
%frag
  PrintLevel 3
  FragProc Connectivity
end


*xyz 0 1
 O(1)  0.000000     0.000000     0.000000
 C     0.000000     0.000000     1.380000
 C     1.300000     0.000000    -0.460000
 H    -0.500000     0.870000     1.740000
 H    -0.500000    -0.870000     1.740000
 H     1.000000     0.000000     1.740000
 H     1.300000     0.000000    -1.530000
 H     1.800000     0.870000    -0.100000
 H     1.800000    -0.870000    -0.100000
 H    -1.840000     0.000000    -0.650000
 O    -2.440000     0.000000     0.080000
 H    -3.300000     0.000000    -0.300000
```

(continues on next page)

(continued from previous page)

```
*
```

The output of the automatic fragmentation tool indicates that the first fragment was assigned by the `Orca_Input` procedure, while the remaining three fragments were assigned by the `Connectivity` procedure.

```
===================================================
Tfragmentator: Fragmenting by Orca_input
===================================================

------------------------------------------
 Assigned Fragment: 1
------------------------------------------
  Name: User-defined:0 Method: Orca_input
  Natoms: 1 Charge: 0 Mult: 1 InputId: 1

 0 O     0.000000    0.000000    0.000000
------------------------------------------

===================================================
Tfragmentator: Fragmenting by Connectivity
===================================================

------------------------------------------
 Assigned Fragment: 2
------------------------------------------
  Name: Connectivity:1 Method: Connectivity
  Natoms: 4 Charge: 0 Mult: 1

 1 C     0.000000    0.000000    1.380000
 3 H    -0.500000    0.870000    1.740000
 4 H    -0.500000   -0.870000    1.740000
 5 H     1.000000    0.000000    1.740000
------------------------------------------

------------------------------------------
 Assigned Fragment: 3
------------------------------------------
  Name: Connectivity:2 Method: Connectivity
  Natoms: 4 Charge: 0 Mult: 1

 2 C     1.300000    0.000000   -0.460000
 6 H     1.300000    0.000000   -1.530000
 7 H     1.800000    0.870000   -0.100000
 8 H     1.800000   -0.870000   -0.100000
------------------------------------------

------------------------------------------
 Assigned Fragment: 4
------------------------------------------
  Name: Connectivity:3 Method: Connectivity
  Natoms: 3 Charge: 0 Mult: 1

 9 H    -1.840000    0.000000   -0.650000
 10 O    -2.440000    0.000000    0.080000
 11 H    -3.300000    0.000000   -0.300000
------------------------------------------
```

Automatic Fragmentation: Atomic and NotAssigned

`FragProc Atomic` and `FragProc NotAssigned` are termination procedures. `FragProc Atomic` assigns each previously unassigned atom to its own individual fragment, while `FragProc NotAssigned` assigns all remaining unassigned atoms to a single fragment.

Similar to other `FragProc` methods, the output of the automatic fragmentation tool indicates that the `Atomic` or `Not_assigned` procedure has been used to generate the corresponding fragments.

```
%frag
  PrintLevel 3
  FragProc Atomic
end

*xyz 0 1
 O(1)   0.000000     0.000000     0.000000
 C      0.000000     0.000000     1.380000
 C      1.300000     0.000000    -0.460000
 H     -0.500000     0.870000     1.740000
 H     -0.500000    -0.870000     1.740000
 H      1.000000     0.000000     1.740000
 H      1.300000     0.000000    -1.530000
 H      1.800000     0.870000    -0.100000
 H      1.800000    -0.870000    -0.100000
 H     -1.840000     0.000000    -0.650000
 O     -2.440000     0.000000     0.080000
 H     -3.300000     0.000000    -0.300000
*
```

```
===================================================
Tfragmentator: Fragmenting by Orca_input
===================================================

------------------------------------------
 Assigned Fragment: 1
------------------------------------------
  Name: User-defined:0 Method: Orca_input
  Natoms: 1 Charge: 0 Mult: 1 InputId: 1

 0 O     0.000000    0.000000    0.000000
------------------------------------------

===================================================
Tfragmentator: Fragmenting by Atomic
===================================================

------------------------------------------
 Match: 1, Assigned Fragment: 2
------------------------------------------
  Name: C  Method: Atomic
  Natoms: 1 Charge: 0 Mult: 1

 1 C     0.000000    0.000000    1.380000
------------------------------------------


------------------------------------------
 Match: 2, Assigned Fragment: 3
------------------------------------------
  Name: C  Method: Atomic
  Natoms: 1 Charge: 0 Mult: 1

 2 C     1.300000    0.000000   -0.460000
```

(continues on next page)

(continued from previous page)

```
------------------------------------------

...

------------------------------------------
 Match: 11, Assigned Fragment: 12
------------------------------------------
  Name: H  Method: Atomic
  Natoms: 1 Charge: 0 Mult: 1

 11 H    -3.300000    0.000000   -0.300000
------------------------------------------
```

```
%frag
  PrintLevel 3
  FragProc NotAssigned
end


*xyz 0 1
 O(1)  0.000000     0.000000     0.000000
 C     0.000000     0.000000     1.380000
 C     1.300000     0.000000    -0.460000
 H    -0.500000     0.870000     1.740000
 H    -0.500000    -0.870000     1.740000
 H     1.000000     0.000000     1.740000
 H     1.300000     0.000000    -1.530000
 H     1.800000     0.870000    -0.100000
 H     1.800000    -0.870000    -0.100000
 H    -1.840000     0.000000    -0.650000
 O    -2.440000     0.000000     0.080000
 H    -3.300000     0.000000    -0.300000
*
```

```
===================================================
Tfragmentator: Fragmenting by Orca_input
===================================================

------------------------------------------
 Assigned Fragment: 1
------------------------------------------
  Name: User-defined:0 Method: Orca_input
  Natoms: 1 Charge: 0 Mult: 1 InputId: 1

 0 O     0.000000    0.000000    0.000000
------------------------------------------

===================================================
Tfragmentator: Setting not assigned atoms to a fragment
===================================================

------------------------------------------
 Assigned Fragment: 1
------------------------------------------
  Name: Not Assigned Method: Not_assigned
  Natoms: 11 Charge: 0 Mult: 1

 1 C     0.000000    0.000000    1.380000
 2 C     1.300000    0.000000   -0.460000
 3 H    -0.500000    0.870000    1.740000
```

(continues on next page)

(continued from previous page)

```
 4  H    -0.500000   -0.870000    1.740000
 5  H     1.000000    0.000000    1.740000
 6  H     1.300000    0.000000   -1.530000
 7  H     1.800000    0.870000   -0.100000
 8  H     1.800000   -0.870000   -0.100000
 9  H    -1.840000    0.000000   -0.650000
 10 O     -2.440000    0.000000    0.080000
 11 H     -3.300000    0.000000   -0.300000
-----------------------------------------
```

Note

- `FragProc NotAssigned` is always applied at the end of all `FragProc` procedures to ensure that no atoms remain without a fragment assignment.

Automatic Fragmentation: Internal Libraries

ORCA includes a series of internal libraries containing definitions of many common molecular structures. In all cases, structure recognition is performed using a VF2 subgraph isomorphism algorithm applied to molecular graphs constructed from Cartesian coordinates.

The available fragmentation procedures that make use of internal libraries are listed in Table 2.60.

Table 2.60: Simple input keywords for Fragment detection

Fragment detection Keyword	Description
`Function`	Contains a list of the most common organic functional groups.
`Aminoaci`	Contains a list of all amino acids, including all common protonation states, but excluding zwitterionic forms.
`AABackbo`	Contains fragment definitions for amino acid backbone detection.
`Backbone`	Performs `AABackbone` followed by merging all fragments into a single protein backbone fragment.
`SeqBackb`	Similar to `AABackbone`, but peptide bonds are assigned as separate fragments.
`AASidech`	Contains a list of all amino acid side chains.
`AASCFine`	Contains a detailed list of organic functional groups within amino acid side chains.
`NABackbo`	Contains fragment definitions for DNA/RNA backbones; the phosphate group is assigned to the 3′ position.
`SEQNABac`	Contains fragment definitions for DNA/RNA backbones; the phosphate group is assigned to the 5′ position.
`NABBFine`	Same as `NABackbone`, but further splits the phosphate group.
`Nucleoti`	Contains a list of all nucleic acids.
`NASidech`	Contains a list of all nucleic acid side chains.
`Solvents`	Contains definitions for common solvents: 1-octanol, n-hexane, cyclohexane, toluene, chlorobenzene, tetrahydrofuran, benzene, N,N-dimethylformamide, pyridine, dimethyl sulfoxide, acetone, ethanol, acetonitrile, methanol, chloroform, carbon tetrachloride, dichloromethane, ammonia, and water.
`Water`	Contains definitions of water molecules, faster than `Solvents` if only fragment water molecules.

Similar to other fragmentation procedures, listing multiple libraries in `FragProc` will apply the procedures consecutively. Using the first example listed in *Automatic Fragmentation* (*input*), and including `PrintLevel 3` in the `%frag` block to increase verbosity, the output will indicate the assignment of fragments by the `Water` and `Functional_groups` procedures.

```
===================================================
Tfragmentator: Fragmenting by Water
===================================================

------------------------------------------
 Match: 1, Assigned Fragment: 1
------------------------------------------
  Name:  WATER Method: Water
  Natoms: 3 Charge: 0 Mult: 1

 11 H    -0.920000    0.850000   -2.430000
 12 O    -1.690000    0.830000   -3.000000
 13 H    -1.640000    1.630000   -3.510000
------------------------------------------

===================================================
Tfragmentator: Fragmenting by Functional_groups
===================================================

------------------------------------------
 Match: 1, Assigned Fragment: 2
------------------------------------------
  Name:  CH3 Method: Functional_groups
  Natoms: 4 Charge: 0 Mult: 1

 0 C     0.000000    1.270000   -0.260000
 3 H    -0.890000    1.320000   -0.910000
 4 H     0.890000    1.320000   -0.910000
 5 H     0.000000    2.180000    0.370000
------------------------------------------

------------------------------------------
 Match: 2, Assigned Fragment: 3
------------------------------------------
  Name:  CH3 Method: Functional_groups
  Natoms: 4 Charge: 0 Mult: 1

 2 C     0.000000   -1.270000   -0.260000
 8 H     0.890000   -1.320000   -0.910000
 9 H     0.000000   -2.180000    0.370000
 10 H    -0.880000   -1.300000   -0.910000
------------------------------------------

------------------------------------------
 Match: 1, Assigned Fragment: 4
------------------------------------------
  Name:  CH2 Method: Functional_groups
  Natoms: 3 Charge: 0 Mult: 1

 1 C     0.000000   -0.000000    0.580000
 6 H     0.880000   -0.000000    1.250000
 7 H    -0.880000   -0.000000    1.250000
------------------------------------------
```

Automatic Fragmentation: External Libraries

The automated fragmentator also allows users to supply `.xyz` files via the `XZYFRAGLIB` variable in `%frag` block, containing geometries that should be recognized as fragments. The `FragProc Extlib` procedure automatically converts each provided geometry into a molecular graph and then applies a VF2 subgraph isomorphism algorithm—just as is done with the internal libraries.

The following example uses definitions of CH3O and CH3 fragments from the file `Mylib.xyz` to identify and generate fragments within the dimethyl ether geometry provided below:

```
%frag
  PrintLevel 3
  FragProc Extlib
  XZYFRAGLIB "Mylib.xyz"
end

*xyz 0 1
 O     0.000000     0.000000     0.000000
 C     0.000000     0.000000     1.380000
 C     1.300000     0.000000    -0.460000
 H    -0.500000     0.870000     1.740000
 H    -0.500000    -0.870000     1.740000
 H     1.000000     0.000000     1.740000
 H     1.300000     0.000000    -1.530000
 H     1.800000     0.870000    -0.100000
 H     1.800000    -0.870000    -0.100000
*
```

Where `Mylib.xyz` contains definitions for methyl and methoxy groups.

```
    5
CHARGE 0 MULT 1 NAME CH3O
 O     0.000000     0.000000     0.000000
 C     0.000000     0.000000     1.380000
 H     1.008807     0.000000     1.736663
 H    -0.328435     0.953845     1.736667
 H    -0.794950    -0.621083     1.736667
    4
CHARGE 0 MULT 1 NAME CH3
 C     0.000000     0.000000     0.000000
 H     0.000000     0.000000     1.070000
 H     1.008807     0.000000    -0.356663
 H    -0.328435    -0.953845    -0.356667
```

The result is the assignment of atoms 0, 1, 3, 4, and 5 to a CH3O fragment, with the remaining atoms assigned to a CH3 by the `Ext_lib` procedure.

```
===================================================
Tfragmentator: Fragmenting by Ext_lib
===================================================
****
**** There are 2 Ref structures found in file Mylib.xyz
****

------------------------------------------
 Match: 1, Assigned Fragment: 1
------------------------------------------
  Name:  CH3O Method: Ext_lib
  Natoms: 5 Charge: 0 Mult: 1

 0 O     0.000000    0.000000    0.000000
 1 C     0.000000    0.000000    1.380000
```

(continues on next page)