AFFCK User's Manual

BY Huanchen ZHAI

February 6, 2015

Contents

Chapter 1

General Information

1.1 Contributions and Acknowledgments

AFFCK (Adaptive Force Field-Assisted *ab initio* Coalescent Kick Method) is a program package designed for searching global minimum molecular structures. The original program (FF part) was written in `C#`, while CK part of the program was written in `Fortran 90`. These were finished at UCLA during the summer of 2014. Later, they were translated by myself to `C++` and formed this package.

The following people in Dr. Alexandrova's group have offered me many useful suggestions and helps for this project:

Dr. Anastassia N. Alexandrova, Dr. Jonny Dadras, and Mai-Anh Ha.

1.2 Functions of AFFCK

AFFCK can be built and run on both Windows and Linux system. Main functions of AFFCK are:

1. Gas phase initial structures generation using Coalescent Kick method.

2. Initial structures generation for cluster put on a given surface.

3. Force Field Fitting of a list of given structures and their corresponding energies.

4. Test the validity of the fitted formula on another list of structures.

5. Structure optimization using the fitted formula.

There are also some tools offered by AFFCK package and the usage of them will be covered in the following chapters.

1.3 Installation of AFFCK

The AFFCK is normally distributed as tar file, so first you need to unpack the package:

```
tar -xzvf affck-1.0.tar.gz
```

Then you can enter the generated directory and use ‘make’ command to compile the source code:

```
cd affck-1.0
make
```

This will generate the executable file ‘affck.exe’ in the ‘bin’ sub-directory and some object files in the ‘src’ sub-directory. The source codes are located in ‘src’. Some useful Python scripts are in ‘tools’. This manual is in ‘docs’. And ‘res’ sub-directory includes a file named ‘default.txt’ which contains information about the default value of all parameters and should be referred when running the program. The ‘test’ sub-directory includes some test cases.

The following will remove all files and directories that ‘make’ generates:

```
make clean
```

1.4 Quick Start

Please make a directory named ‘run’ under the ‘affck-1.0’ directory, so that we will use the ‘run’ directory as our current work directory.

1.4.1 CK gas phase

For every calculation you need to first create an input file. Create a file named ‘*input-ck.txt*’ and copy the following into it:

```

! input file for a CK gas phase calculation

{ parameter: ck_name = Pt8 }
{ execute: ck }

```

The input file uses a relatively free format. The characters after ‘!’ will be understood as comments. The brackets ‘{’ and ‘}’ denote the beginning and the end of a block, respectively. In each block, you should first write the *block name* and a ‘:’. Then we can specify the parameters as ‘*item name* = *item value*’. Some *item name* do not need a value.

For the above example, we set the value of parameter ‘ck_name’ to be ‘Pt8’, which means I want to calculate for Pt_8 cluster. The parameters for all programs should be written on ‘parameter’ block. And in ‘execute’ block, we can specify which job we would like to run. In this case, the job name is ‘ck’. All other parameters will be set to its default value.

Now we will run the affck program. I assume that the work directory is ‘affck-1.0/run/’ now. Enter:

```
../bin/affck.exe input-ck.txt ../res/default.txt
```

We can see that ‘affck.exe’ takes two arguments. The first argument contains the path to input file. The second argument contains the path to the default value file.

The detailed calculation information will be written on the screen. And by default, the generated ck structured will be stored at ‘data/ck_structs’ directory, relative to the current work directory. You can find all these generated ck structures there. By default, the program will generate 100 such structures, named ‘pos_0.xyz’ through ‘pos_99.xyz’.

1.4.2 CK cluster with surface

If you would like to include a surface you should write the input file as following (named, for example, *input-ckcs.txt*):

```
! input file for a CK cluster with surface calculation

{ parameter:
   ck_name         = Pt7B;
   ck_symmetry     = C3z;
   ck_surface_file = ../test/data/surface-al2o3.xyz;
}
{ execute: ck }
```

Here we added a symmetry parameter just for demonstration, and you can also remove it. For a surface calculation you must prepare a xyz format file describing the surface structure by yourself. The surface plane must be perpendicular to z axis so that the program will know where to put the cluster. You should give the path to this file using parameter ‘ck_surface_file’. Here we use a existing file on ‘test’ sub-directory for demonstration. When we have to set more than one parameters, we should divided them with ‘;’. And whether they are written in one line or not will not be significant.

To run the program:

```
../bin/affck.exe input-ckcs.txt ../res/default.txt
```

If there is already a directory named ‘ck_structs’ in the current work directory, the new generated contents will be automatically stored to another directory named ‘ck_structs_2’ as you can see from this case. This setting is designed to avoid overriding.

1.4.3 FF fitting

To start a Force Field fitting calculation, you should first put your structures into one directory, with each structure named regularly. For example, the name of your structures could be ‘pos_*id*.xyz’, where *id* is a unique number denoting that structure. And you also need to run single point energy calculations for these structures using some *ab initio* programs, for example, Turbomole. AFFCK does not invoke any *ab initio* programs internally.

The calculated single point energy should be sorted and listed in an *energy list file*, in which each line corresponding to one structure. The *energy list file* might look like this:

```
0   -954.5871666256
1   -954.5692389659
2   -954.5088627820
3   -954.6216275522
4   -954.6216574405
5   -954.5137650930
...
```

It is good to sort this file making it order by energy from small to large, so that it will be convenient to exclude some high-energy terms. For example, you can use ‘sort’ command in the Shell Script:

```
sort -k 2n energy.txt > energy_sorted.txt
```

Then *energy_sorted.txt* looks like:

```
23   -954.6672007768
22   -954.6564973158
88   -954.6507867720
24   -954.6468389287
10   -954.6397134115
9   -954.6363227675
...
```

For demonstration, we have already prepared a sample *energy list file* ('test/data/singlet-dscf/energy.txt') and the corresponding *directory of structures* ('test/data/singlet-dscf'). You can check the structures in this directory. The 100 structure is the same as what we have generated at 1.4.1. And the energies are calculated for singlets in Turbomole using pbe0 DFT functional.

To fit these energies, we write the following input file *input-fit.txt*:

```
! input file for a FF fitting calculation

{ parameter:
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
}
{ execute: ff }
```

Here the job name is 'ff' and we have set up three parameters. 'standard_list_file' gives the path to *energy list file*. 'standard_xyz_files' gives the path to all the corresponding xyz files, in which we use a '#' denoting the *id* of every single structure. When the program run, it will automatically replace the '#' with some *id* numbers read from *energy list file*. 'standard_fitted_output' tells the program where to write fitted energies information. We will call this output file *output energy list file*. If you do not specify this parameter, the program will just not write the fitted energy information.

To run the program:

```
../bin/affck.exe input-fit.txt ../res/default.txt
```

Some general information about the fitting will be shown on the screen, for example, in this case part of the output on the screen will be:

```
Sum of squared residual: 0.00918275
Standard deviation: 0.00983161
Standard deviation (* htr->ev): 0.267532
Fitted energies have been written.
```

"*Sum of squared residual*" ($r = \sum(\Delta E)^2$) and "*Standard deviation*" ($\sigma = \sqrt{r/n}$) are measured with the unit as same as that of the *energy list file*. "*Standard deviation (* htr->ev)*" ($\sigma^* = 27.21138505\sigma$) is the value in eV when "*Standard deviation*" is in Hartree. So if σ is not in hartree, the value of σ^* is meaningless.

Note that there are 100 xyz files in directory 'test/data/singlet-dscf' while we only listed 95 energies in the file 'test/data/singlet-dscf/energy.txt' ($n = 95$). This should be normal since not all single point energy calculation converging. And the program will not see the unlisted items.

The second part of the output on the screen are the fitted functions:

```
Fitted functions:
constant:
  = $-3 (# 95 ) GConst = -953.8
stretching ~0: pt-pt:
  = $10 (# 871 ) GP2C = -0.01969 * l^2 - 2*l*-0.05838 + -0.2153
bending ~0: pt-pt-pt:
  = $30 (# 1674 ) GP2C = -0.0001937 * l^2 - 2*l*0.001309 + 0.009685
van der Waals ~3: pt-pt:
  = $53 (# 785 ) GLJ = 423.5 / l^9 - 39.54 / l^6
van der Waals ~4: pt-pt:
  = $54 (# 595 ) GLJ = 283.1 / l^9 - 31.13 / l^6
van der Waals ~5: pt-pt:
  = $55 (# 540 ) GLJ = 262.7 / l^9 - 28.49 / l^6
van der Waals ~6: pt-pt:
  = $56 (# 371 ) GLJ = 263.2 / l^9 - 27.61 / l^6
van der Waals ~7: pt-pt:
  = $57 (# 369 ) GLJ = 276.9 / l^9 - 26.36 / l^6
```

Here we have one constant term, one stretching term, one bending term and five van der Waals terms. The following chapters will tell you how to include more terms or exclude some terms. You should make sure that the fitted coefficients are all physical. For example, the coefficients of quadratic term of bending and stretching terms must be positive. If not, you might need to adjust some parameters to get a better fitting.

Finally, we look at the generated *output energy list file*. The first several lines are:

```
0 -954.6059541
1 -954.5508281
2 -954.5184263
3 -954.609534
4 -954.6173802
5 -954.494547
...
```

They are just like the format of input *energy list file*, except the energies listed are fitted energies.

1.4.4 FF testing

Testing is using an existent FF formula to predict the energy of some other structures. Typically, you need to run a FF fitting job before testing (how to save the fitted results to file will be discussed in following chapters but not here, see 5.2.7).

After you have the FF formula, the FF testing input file is just like that for FF fitting. Particularly, you can test the structures used for fitting themselves, and this will give the identical results as before. Note that FF testing will out output the functions on the screen, since it assumes you have learned this information when fitting. Here is the input file *input-test.txt* for FF testing:

```
! input file for a FF testing calculation

{ parameter:
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;

   test_list_file         = ../test/data/singlet-dscf/energy.txt;
   test_xyz_files         = ../test/data/singlet-dscf/pos_#.xyz;
   test_fitted_output     = data/testout.txt;
}
{ execute: ff; test }
```

We listed two jobs in the 'execute' block since we need to run FF fitting before we run FF testing. And the program will calculating the *standard deviation* for testing structures as well. If you have some other structures for testing rather than the ones used to fit, they might or might not have their pre-calculated DFT energies. So in 'test_list_file' you can also list only *ids*. Suppose we have a file named *ids.txt* that contains all *ids*:

```
0
1
2
3
...
```

You should write the input file as:

```
! input file for a FF testing calculation 2
```

```
{ parameter:
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;

   test_list_file         = ids.txt;
   test_xyz_files         = ../test/data/singlet-dscf/pos_#.xyz;
   test_list_file_energy_column  = -1;
   test_fitted_output     = data/testout.txt;
}
{ execute: ff; test }
```

Note that in this case you must set the parameter 'test_list_file_energy_column' to -1 which is to tell the program not to find an energy column. The default value of the parameter is 1, which means the energy is on the second column, and we count columns from zero.

You might find it boring to just write a file that contains only 0 to 99, with one number at each line. Actually we have an convenience grammar for this, you can simply write 'seq 0 99' for the value of 'test_list_file' instead of giving a file path, where 'seq *a* *b*' will generate an integer list from *a* to *b*.

In the case that you do not give the pre-calculated DFT energies for testing, the program will assume all the structures to have DFT energies zero. This will make the output *standard deviation* should be meaningless. Which is meaningful, in this case, is the fitted energies written in the 'test_fitted_output' file. The fitted energies for these testing structures are independent to their pre-calculated DFT energies.

1.4.5 FF optimization

As long as you have your fitted FF formulas, you will be able to optimizing some structures including the structures used for fitting themselves. You can write the input file as follows if you would like to optimizing structures used for fitting (*input-opt.txt*):

```
! input file for a FF optimization calculation

{ parameter:
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
   opt_list_file          = seq 0 99;
   opt_xyz_files          = ../test/data/singlet-dscf/pos_#.xyz;
   opt_output_directory   = data/opt;
   opt_output_xyz_files   = pos_#.xyz;
   opt_output_list_file   = data/opt/optimized_energy.txt;

}
{ execute: ff; opt }
```

The job name for FF optimization is 'opt'. And if you would like to optimize some other structures, just change 'opt_xyz_files' to somewhere else. For optimization, you do not need to give the DFT energies for the input structures, so here using 'seq' notation is safe for 'opt_list_file'. Last thing to note about the input file is that for parameter 'opt_output_xyz_files' you should not give the full absolute or relative path to the output files, but only a filename. The location of these output files should be given by the parameter 'opt_output_directory'. However, for 'opt_output_list_file' you need to write a full path, since whether to put the list file together with generated structures, or not, is your freedom.

You will see the information of optimizing each structure on the screen after you run the program:

```
../bin/affck.exe input-opt.txt ../res/default.txt
```

Part of the output on the screen is:

```
opt 1 of 100 (8) -954.606 -954.862
opt 2 of 100 (6) -954.551 -954.72
opt 3 of 100 (4) -954.518 -954.622
opt 4 of 100 (5) -954.61 -954.779
opt 5 of 100 (8) -954.617 -954.819
```

For each structure, we presented the serial number (not *id*) of the structure, the total number of structures, number of steps for optimization, initial FF energy, and final FF energy. Final structures will be written as separate files into 'opt_output_directory'. And the *id*, and initial and final energies of each structure will be written into 'opt_output_list_file', given that this parameter is not empty.

Chapter 2

AFFCK Input

2.1 Executable arguments

The executable ‘bin/affck.exe’ can take zero, one, or two arguments:

```
../bin/affck.exe [input-file-path] [default-file-path]
```

If you omit the second argument, the program will look for a file named *default.txt* in current work path, so the following two commands are identical:

```
../bin/affck.exe input.txt
../bin/affck.exe input.txt default.txt
```

If you omit both arguments, the program will look for *default.txt* in the same way as stated above, and read input from keyboard. So if you write:

```
../bin/affck.exe
```

The program will write the program head and waiting for your input:

```
#################################################
#                AFFCK Program                  #
#             Author: Huanchen ZHAI             #
#               stczhc@gmail.com                #
#                Jan. 29, 2015                  #
#################################################

```

You should type your input and end it with only three characters ‘EOF’ in a line. For example, you type

```
! input for a CK gas phase calculation
{ parameter: ck_name = Pt8 }
{ execute: ck }
```

```
EOF
```

Then the program will continue running using what you typed as the input file and then end.

2.2 Blocks

The AFFCK input file is organized using blocks. You should write the *block name* explicitly at the beginning of a block. The brackets '{' and '}' denote the beginning and the end of a block, respectively. In each block, ':' is used to separate the *block name* and *block body*. The *block name* tells what type the block is. There are four types of blocks: **parameter**, **execute**, **weight**, and **covalent**. These blocks can appear in any order and multiple times. **parameter** block is used to set parameters. **execute** block tells which job to run and in which order. **weight** and **covalent** blocks are used in *default.txt* to set the value of atomic weight and covalent radius of elements, respectively. Unless you need to change some of these values, you will not need to write about the last two blocks in your own input file.

The parameter block can be written without '{' and '}' notations and *block name*. So the following

```
ck_name = Pt8
{ execute: ck }
ck_name = H2O
{ execute: ck }
```

is equivalent to:

```
{ parameter: ck_name = Pt8 }
{ execute: ck }
{ parameter: ck_name = H2O }
{ execute: ck }
```

There is an exception with this simplified notation: the contents without '{' and '}' protected should not contains a ':' as one or more of its parameter's value. For example, you might run AFFCK in Windows and for some parameters you use the absolute path, like 'C:/a.txt'. In this case, you must write 'parameter', '{' and '}' explicitly.

The above example is also showing that you can run a job or multiple jobs sequentially with writing one input file, and you can change the value of parameters multiple times for different jobs.

Some *block names* have their abbreviated words: **execute** = **exe**, **covalent** = **cov**, and **weight** = **wei**. **parameter** does not need an abbreviated words since itself can be omitted in most cases. So the following is valid and has the same meaning as the previous version:

```
ck_name = Pt8
{ exe: ck }
ck_name = H2O
{ exe: ck }
```

You can use '!' to make all contents in the same line and after this character become comments. You can put your input contents in one line or multiple lines, the only restriction is that you should not break words. The following is okay, however hard to read:

```
ck_name =
Pt8
{ exe: ck }
ck_name
=
H2O
{
exe
:
ck }
```

2.2.1 Execution groups

The input file will be divided to several logical execution groups when the program analyzing it. One typical execution group contains some **parameter**, **covalent**, and **weight** blocks and then some **execute** blocks. In each execution group there should not be any **parameter**, **covalent**, or **weight** block after **execute** blocks. Two or more successive **execute** blocks will be put into one execution group. But an empty **parameter** block can separate them. For example,

```
{ parameter:
 ck_name = Pt8;
 ck_symmetry = C2z;
 standard_list_file     = ../test/data/singlet-dscf/energy.txt;
 standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
 standard_fitted_output = data/stdout.txt;
}
{ exe: ck }
{ exe: ff }
{ parameter:
 test_list_file     = ../test/data/singlet-dscf/energy.txt;
 test_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
 test_fitted_output = data/testout.txt;
}
{exe: test }
```

will be understood as:

```

! Execution group 1
  { parameter:
   ck_name = Pt8;
   ck_symmetry = C2z;
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
  }
  { exe: ck }
  { exe: ff }

! Execution group 2
  { parameter:
   test_list_file     = ../test/data/singlet-dscf/energy.txt;
   test_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   test_fitted_output = data/testout.txt;
  }
  {exe: test }
```

However, if you insert an empty **parameter** block into it,

```
{ parameter:
 ck_name = Pt8;
 ck_symmetry = C2z;
 standard_list_file     = ../test/data/singlet-dscf/energy.txt;
 standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
 standard_fitted_output = data/stdout.txt;
}
{ exe: ck }
{ parameter: }
{ exe: ff }
{ parameter:
 test_list_file     = ../test/data/singlet-dscf/energy.txt;
 test_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
 test_fitted_output = data/testout.txt;
}
{exe: test }
```

This will be understood as:

```

! Execution group 1
  { parameter:
   ck_name = Pt8;
   ck_symmetry = C2z;
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
  }
  { exe: ck }

! Execution group 2
  { parameter: }
  { exe: ff }

! Execution group 3
  { parameter:
   test_list_file     = ../test/data/singlet-dscf/energy.txt;
   test_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   test_fitted_output = data/testout.txt;
  }
  { exe: test }
```

Though the two different execution group division make no difference when running the program, in this case. The program will echo the parameters setting groupwisely. You can check the group division by looking at the output when running the program.

2.2.2 Parameter overriding

You should take care of the order of parameters when writing input files. Parameter values in the user's input file will override that defined in the default file. Newly defined parameters will override previous ones. For example, if you would like to execute a CK with surface calculation, and then a gas phase CK calculation, the following input file will give the wrong result:

```
{ parameter:
   ck_name = Pt8;
   ck_surface_file = ../test/data/surface-al2o3.xyz;
}
{ execute: ck }
{ parameter: ck_name = Pt8; }
{ execute: ck }
```

Instead, you should write:

```
{ parameter:
   ck_name = Pt8;
   ck_surface_file = ../test/data/surface-al2o3.xyz;
}
{ execute: ck }
{ parameter:
   ck_surface_file = ;
}
{ execute: ck }
```

That is, for a gas phase calculation you should reset 'ck_surface_file' to its default value, empty. Also, since the value of parameters will preserve from one execution group to another, you need not to set 'ck_name' twice for the same value in this case, however you can do so.

2.2.3 Multi-value expanding

You can give multi-values to a single parameter at one time, using delimiter '|'. The multi-value term will be expanded, within each execution group, at runtime. For example,

```
{ parameter:
   ck_name = Pt8;
   ck_symmetry = C3z | C2z | C1;
}
{ execute: ck }
```

will be expanded to this at runtime:

```

! parameter set 1
  { parameter:
   ck_name = Pt8;
   ck_symmetry = C3z;
  }
  { execute: ck }

```

```
! parameter set 2
  { parameter:
   ck_symmetry = C2z;
  }
  { execute: ck }

! parameter set 3
  { parameter:
   ck_symmetry = C1;
  }
  { execute: ck }
```

You can also find this information from the output of the program when running it. Note that **execute** blocks will be copied to each parameter set so that the job can be done multi-times. The parameters that have only one value will only appear at first block, since their value will be inherited by parameter sets afterwards. The multi-value will be expand to single value terms and be put into each parameter sets.

If you have two or more multi-value parameters, the program will try to form any possible combination of these parameters when expanding. For example,

```
{ parameter:
   ck_name = Pt8 | Pt7B;
   ck_symmetry = C3z | C2z | C1;
   ck_number = 5;
}
{ execute: ck }
```

will be expanded to this at runtime:

```


! parameter set 1
  { parameter:
   ck_name = Pt8;
   ck_symmetry = C3z;
   ck_number = 5;
  }
  { execute: ck }

! parameter set 2
  { parameter: ck_name = Pt8; ck_symmetry = C2z; }
  { execute: ck }

! parameter set 3
```

```
  { parameter: ck_name = Pt8; ck_symmetry = C1; }
  { execute: ck }

! parameter set 4
  { parameter: ck_name = Pt7B; ck_symmetry = C3z; }
  { execute: ck }

! parameter set 5
  { parameter: ck_name = Pt7B; ck_symmetry = C2z; }
  { execute: ck }

! parameter set 6
  { parameter: ck_name = Pt7B; ck_symmetry = C1; }
  { execute: ck }
```

This will generate $5*6 = 30$ structures altogether (considering the path overriding is avoided, see 2.4). Finally, note that this expansion will only occur inside each execution group, so multi-values of different execution groups will not be combined (But parameters from different **parameter** blocks in the same execution group will be combined). As an example,

```
{ parameter: ck_name = Pt8 | Pt7B; }
{ parameter: ck_symmetry = C3z | C2z; ck_number = 5; }
{ execute: ck }
{ parameter:  ck_number = 10 | 20; }
{ execute: ck }
```

will be expanded to this at runtime:

```
! Execution group 1
  ! parameter set 1
    { parameter: ck_name = Pt8; }
    { parameter: ck_symmetry = C3z; ck_number = 5; }
    { execute: ck }
  ! parameter set 2
    { parameter: ck_name = Pt8; }
    { parameter: ck_symmetry = C2z; }
    { execute: ck }
  ! parameter set 3
    { parameter: ck_name = Pt7B; }
    { parameter: ck_symmetry = C3z; }
    { execute: ck }
  ! parameter set 4
```

```
    { parameter: ck_name = Pt7B; }
    { parameter: ck_symmetry = C2z; }
    { execute: ck }

! Execution group 2
  ! parameter set 1
    { parameter: ck_number = 10; }
    { execute: ck }
  ! parameter set 2
    { parameter: ck_number = 20; }
    { execute: ck }
```

Note that in this case all the calculation in execution group 2 will be done for Pt_7B, because of the parameter overriding. Multi-value notation is just a special grammar for simplifying the input file. The form after expansion is just equivalent to the multi-value form, and you can use both form in your input file.

In addition, multi-value does not work for empty values. And all empty values you give in a multi-value expression will be discarded. So the following two are equivalent and they both have three possible values offered:

```
{ parameter: ck_surface_file = aa.xyz|bb.xyz|cc.xyz; }
{ parameter: ck_surface_file = |aa.xyz|bb.xyz|cc.xyz; }
```

2.3 Value types

Each parameter has its value type. We have six value types: **Boolean**, **Double**, **Integer**, **Length**, **String**, and **File**. (The parameter 'functions' has a special value type **IntegerList**, which will be discussed when introducing that parameter.) The value type of a particular parameter is predefined, and you will not be able to change it. In following chapters we will give you the value type of every parameter.

Note that usually the parameter name is case insensitive, but the parameter value is case sensitive.

2.3.1 Boolean

Boolean type parameters can have value *true* or *false*. Normally you can write:

```
file_override = false;
path_override = true;
```

Particularly all the following will all set 'file_override' to *false*:

```
file_override = F; file_override = f;
file_override = false; file_override = 0;
```

All other values will be understood as *true*.

2.3.2 Double

Double type parameters are used to describe physical quantities that are not **Length**. The following will be okay for a **Double** parameter:

```
ck_drel = 0.03;
cg_epsilon_n = 1e-6;
```

You cannot write an unit name for the **Double** type parameter. **Length** can be regarded as an extension of **Double** and it can accept length units.

2.3.3 Integer

For example,

```
cg_imax = 200;
standard_list_file_energy_column = -1;
```

And the following

```
ck_number = .5;
ck_number = 0.5;
ck_number = 1.5;
ck_number = 2.6;
```

will be understood as 0, 0, 1, and 2, respectively.

2.3.4 Length

Double type parameters include a **Double** value and an unit of length (with a white space separated). Valid unit names are *a*, *ang*, *angstrom*, *bohr*, *au*, *a.u.*, *pm* and *m*. They are transformed internally as follows: $1\,a = 1\,ang = 1\,angstrom, 1\,bohr = 1\,au = 1\,a.u. = 0.5291772108\,angstrom, 1\,pm = 0.01\,angstrom, 1\,m = 1e10\,angstrom$.

You can also omit the unit, then the unit will be determined by the value of parameter 'default_unit'. The default value of 'default_unit' is *angstrom*. The following are all equivalent if 'default_unit' is *angstrom*:

```
ck_shift_length = 0.20 angstrom;
ck_shift_length = 0.20 ang;
ck_shift_length = 0.20;
ck_shift_length = 0.10583544216 bohr;
ck_shift_length = 20 pm;
```

The value of ‘default_unit’ can be changed dynamically. Note that whatever ‘default_unit’ is, all length quantities will be transformed to *angstrom* internally.

2.3.5 String

The values of **String** type parameters are case sensitive. Some **String** type parameters are filenames, and for these parameters the value can be ‘empty’. An ‘empty’ value usually means that you do not need to offer an input file, or you do not want the program to write a output file. You have the following two ways to write an ‘empty’ string:

```
ck_surface_file = ;
ck_surface_file;
```

The ‘default_unit’ itself is a **String** type parameter (‘defu’ is the short name for the parameter name ‘default_unit’, which can be used when writing input file):

1. `default_unit/defu` [type: **String**] [default: angstrom]
   Determine which length unit will use when the user does not write an unit explicitly for the value of a **Length** type parameter.

2.3.6 File

File is an extension of **String** type for input list file parameters. When you need to give a list file, you can give the filename, or just generate a simple list. Suppose that you have a file named ‘a.txt’ with the contents:

```
3
4
5
...
29
30
```

Then the following are equivalent:

```
opt_list_file = a.txt;
opt_list_file = seq 3 30;
```

‘seq’ command can take one, two or three arguments. ‘seq *a*’ will generate a file with only one number *a*. ‘seq *a* *b*’ will generate an integer list from *a* to *b*. ‘seq *a* *b* *c*’ will generate an integer list from *a* to *b*, with the step *c*. For example,

```
opt_list_file = seq 9 -3 -5;
```

will generate the following file:

```
9
```

```
4
-1
```

If you have some irregular numbers, you can write them explicitly as follows:

```
opt_list_file = 1 2 3 5 8 13;
```

When you have only one number, you must use 'seq a' grammar to write it (you cannot write it explicitly), since otherwise the program will understood that number as a filename. Note that because of this special grammar, your list file path should not contain any white space.

2.4 File/path overriding

There are two parameters controlling the overriding behavior:

1. `file_override/fileor` [type: **Boolean**] [default: false]
   If *false*, the program will try to make a new filename when the file for writing already exists. Otherwise, it will override that existing file.

2. `path_override/pathor` [type: **Boolean**] [default: false]
   If *false*, the program will try to make a new directory name when the directory for writing already exists. Otherwise, it will override that existing directory.

If the target directory name is x and it does exist, to avoid overriding, the target file/directory name will changed to x_1. For filename, x.xyz will be changed to x_1.xyz. And if x_n or x_n.xyz already exists, it will be changed to x_ $(n+1)$ or x_$(n+1)$.xyz.

2.5 Random number seed

There is a parameter controlling the random number generating behavior:

1. `random_seed/rands` [type: **Integer**] [default: 1]
   If ≥ 0, the program will set the random number seed to the given number, and the program will generate the same series of random numbers if you do not change the seed manually. If -1, the program will set the random number seed according to current time. In this way the generated numbers will be unpredictable.

Random numbers are only used by 'gas phase ck' and 'ck with surface' job currently. If you would like to repeat one calculation in the future, try to set the seed to be ≥ 0.

Chapter 3

CK Gas Phase

For the CK gas phase calculation, you must set the parameter 'ck__surface__file' to empty.

3.1 Job names

The following job names for **execute** block will all be able to start CK gas phase calculation, and are equivalent:

```
{ exe: ck }
{ exe: ckcs }
```

3.2 Parameters

1. `ck_name/ckname` [type: **String**] [default:][1]
   Specify the element composition of the cluster. This can be a normal chemical formula (case insensitive). These are valid:

   ```
   Pt8, Al2O3, h2o, h2o1, C6H12O6, CHHHH
   ```

   These are illegal:

   ```
   Pt8B0, (H2O)4
   ```

   All element names presented here should have had its 'covalent' and 'weight' values. All common element's values have been set in default file. If you would like to add some strange elements, set these value of them before this parameter. For example,

   ```
   { covalent: Str = 99 pm; }
   ```

[1]This means *empty* **String**.

```
{ weight: Str = 199.99; }
{ parameter: ck_name = Str403; }
{ exe: ck }
```

2. `ck_symmetry/cksym` [type: **String**] [default: C1]
   Specify the required symmetry of the cluster. These are common values:

   ```
   D4zxd, D4zxh, D4zx, S4z,
   D2zxh, D3zxd, D3zxh, D3zx, D2zxd, D2zx,
   C4zh, C2zh, C3zh, C4zvx, C2zvx, C3zvx,
   C4z, C2z, C3z, Ci, Csz, C1
   ```

   These are illegal:

   ```
   D4zz, D4zdx, D4z xd, C4, C3
   ```

   If you would like to generate structures with multiple symmetry kinds, use multi-value grammar (see 2.2.3). Detailed value translation rule:

   (a) **C1**: no symmetry
   (b) **Ci**: reflection
   (c) **Cs**α (α=x, y, z): for example,
   Cs*z*: xy-plane mirror symmetry
   Cs*x*: yz-plane mirror symmetry
   Cs*y*: zx-plane mirror symmetry
   (d) **C*n***α (α=x, y, z, n = 2, 3, 4, 5, 6, ...): n-fold rotary α axis symmetry
   (e) **C*n*αh** (α=x, y, z, n = 2, 3, 4, 5, 6, ...): for example,
   C*nx*h: n-fold axis along x, horizontal mirror plane along yz-plane
   (f) **C*n*αv**β (α, β=x, y, z, n = 2, 3, 4, 5, 6, ...; α and β must be different): for example,
   C*nx*v*y*: n-fold axis along x, vertical mirror plane along xy-plane
   (g) **S*n***α (α=x, y, z, n = 4, 6, 8, 10, ...): n-fold α axis rotary reflection
   (h) **D*n***$\alpha\beta$ (α, β=x, y, z, n = 2, 3, 4, 5, 6, ...; α and β must be different): for example,
   D*nxy*: n-fold axis along x, one of C2 axis along y
   (i) **D*n*$\alpha\beta$h** (α, β=x, y, z, n = 2, 3, 4, 5, 6, ...; α and β must be different): for example,
   D*nxy*h: n-fold axis along x, one of C2 axis along y, horizontal mirror plane vertical to x
   (j) **D*n*$\alpha\beta$d** (α, β=x, y, z, n = 2, 3, 4, 5, 6, ...; α and β must be different): for example,
   D*nxy*d: n-fold axis along x, one of C2 axis along y

3. `ck_dimension/ckdim` [type: **String**] [default: 3]
   Specify the required dimensionality (or degree of freedom) of the cluster. Possible values are:

   ```
   3, 2xy, 2yz, 2zx, 2, 1x, 1y, 1z, 1
   ```

   When ‘ckdim’ = 3, there are no restrictions. When ‘ckdim’ = $2xy$, only x and y coordinates will not be fixed to ‘boxsize’ / 2. When ‘ckdim’ = $1x$, only x coordinates will not be fixed. $1 = 1x, 2 = 2xy$.

4. `ck_surface_file/cksf` [type: **String**] [default:]
   If not empty, the program will conduct a ‘CK cluster with surface’ calculation and this parameter specify the path to the surface xyz file. Otherwise, the calculation will be ‘CK gas phase’.

5. `ck_output_directory/ckod` [type: **String**] [default: data/ck_structs]
   Specify the directory for storing the generated structures. Note that if ‘path_override’ is false, the actual target directory might be changed (see 2.4).

6. `ck_output_xyz_files/ckox` [type: **String**] [default: pos_#.xyz]
   Specify the name pattern for generated structures. Here you should only specify the filename part, since the directory part of the path has been determined by ‘ck_output_directory’. The number of ‘#’ gives at least how many digits will be presented in the generated filename. For example, the pattern

   ```
   pos_#.xyz
   ```

   will produce

   ```
   pos_0.xyz
   pos_1.xyz
   pos_2.xyz
   ...
   pos_10.xyz
   pos_11.xyz
   ...
   pos_100.xyz
   ...
   pos_1000.xyz
   ...
   ```

   But the pattern

   ```
   pos_###.xyz
   ```

   will produce

```
pos_000.xyz
pos_001.xyz
pos_002.xyz
...
pos_010.xyz
pos_011.xyz
...
pos_100.xyz
...
pos_1000.xyz
...
```

And '#'s are not necessarily be continuous:

```
pos##-#.xyz
```

will produce

```
pos00-0.xyz
pos01-1.xyz
pos02-2.xyz
...
pos10-10.xyz
pos11-11.xyz
...
```

Also note that even if you write no '#' in your pattern, the program might generate filenames containing numbers to avoid overriding, given 'file_override' is false (see 2.4).

7. `ck_output_start_number/ckos` [type: **Integer**] [default: 0]
   Specify the first number to replace the '#' in 'ck_output_xyz_files' when generating structures.

8. `ck_number/cknum` [type: **Integer**] [default: 500]
   Specify the number of structures needed to be generated.

9. `ck_fails_limit/ckfl` [type: **Integer**] [default: 10000]
   Specify the maximum number of trials to generate one structure. Since the program will perform a similarity check after generating each structure to avoid producing similar structures, sometimes the generated structures might be discarded and then another trial will be scheduled. When the similarity check or symmetry restriction is strict, it sometimes costs a large number of trials to generate one structure. Note that when 'ck_fails_limit' is achieved, the program will just give a warning and will continue to do next job, instead of stopping.

10. `ck_shift_length/ckshift` [type: **Length**] [default: 0.20 *angstrom*]
    Specify the length of shifting in coalescence procedure.

11. `ck_fragment_pos_eps/ckfpe` [type: **Length**] [default: 1e-6 *angstrom*]
    Specify the maximum admitted error when conclude that the fragment center of mass is overlapped with the cluster center of mass. Sometimes when high symmetry is specified, a fragment can have center of mass just very close to the center of mass of cluster, and in this case the shifting procedure will not be able to put all atoms together any more. If this is identified, the structure will be discarded.

12. `ck_min_distance_factor/ckmf` [type: **Double**] [default: 0.67]
    Specify the minimum distance factor. The generated structure in which any two atoms have a distance less than 'ckmf' * sum of covalent radii of the two atoms will be discarded. This will be checked both before and after the coalescence procedure.

13. `ck_dmax/ckdmax` [type: **Length**] [default: 0.70 *angstrom*]
    Specify the maximum difference of individual interatomic distances.

14. `ck_drel/ckdrel` [type: **Double**] [default: 0.03]
    Specify the accumulated relative difference for all interatomic distances.

15. `ck_boxsize_factor/ckbf` [type: **Double**] [default: 5]
    Specify the box size factor. 'box_size' = 'ckbf' * sum of covalent radii of all atoms in the cluster.

3.3 Outputs

The main part of CK gas phase's output looks like:

```
generating 1 of 50 [1: I=1 ] -> pos_0.xyz
generating 2 of 50 -> pos_1.xyz
generating 3 of 50 -> pos_2.xyz
generating 4 of 50 [1: S=1 ] -> pos_3.xyz
generating 5 of 50 -> pos_4.xyz
generating 6 of 50 -> pos_5.xyz
generating 7 of 50 [2: I=1 S=1 ] -> pos_6.xyz
generating 8 of 50 [4: I=2 S=2 ] -> pos_7.xyz
generating 9 of 50 [3: S=3 ] -> pos_8.xyz
...
```

The output format for each line is:

```
generating <current serial number> of <total number>
    [failure list] -> <target filename>
```

The *failure list* shows the total number of failed trials and the number of failed trials of each kind. The format:

```
<total number>: <type>=<number of this type>
```

Possible types of failed trails are:

1. **I**: Initial Minimum Failure
   Fails to pass the initial minimum distance check.

2. **Fr**: Fragment Failure
   The resulting structure is still fragmented.

3. **S**: Similar Failure
   The resulting structure is similar to one generated before.

3.4 Clear Similarity

The program keeps a structure pool for similarity checking. In a run of the program, every time it finds an unique structure, it will put the structure in the pool. Every new-found structure must be compared with every structure in the pool for similarity checking. Sometimes you might want clear the pool and then start a new CK structure generating job without comparing with former ones. The following job names for **execute** block will all be able to clear structure pool for similarity checking, and are equivalent:

```
{ exe: ckclear }
{ exe: ckcsclear }
{ exe: clear }
```

Chapter 4

CK Cluster with Surface

4.1 Job names

The following job names for **execute** block will all be able to start CK cluster with surface calculation, and are equivalent:

```
{ exe: ck }
{ exe: ckcs }
```

4.2 Parameters

All parameters listed in CK gas phase chapter are still valid for CK cluster with surface calculation. Here we list extra parameters for CK cluster with surface calculation.

1. `ck_top_layer_eps/cktle` [type: **Length**] [default: 1e-5 *angstrom*]
   Specify the maximum error to determine the top layer atoms of the surface. The z-coordinates of these atoms determine the z-coordinate of ghost atoms.

2. `ck_surface_union_random_factor/cksrf` [type: **Double**] [default: 0.20]
   Specify the random factor to combine the surface atoms with the cluster atoms.

3. `ck_surface_union_fails_limit/cksfl` [type: **Integer**] [default: 100]
   Specify the maximum number of trials to combine the surface atoms with the cluster atoms. The position of cluster atoms relative to surface atoms will be randomly assigned, some combinations might fail to pass the final minimum distance check. In this case, another trail will be scheduled. When the maximum number of such trails is achieved, the structure will be discarded.

4. `ck_zgap/ckzgap` [type: **Length**] [default: 0.60 *angstrom*]
   Specify the z-coordinate gap between surface-top and cluster box.

5. $\boxed{\texttt{ck\_zdown/ckzdown}}$ [type: **Length**] [default: 0.02 *angstrom*]
   Specify the z-move-down in each step.

4.3 Outputs

The information given in CK gas phase chapter is still valid for CK cluster with surface calculation. We have some more types of failed trails, however.

1. **G**: Ghost Minimum Failure
   The configuration with ghost atoms fails to pass the minimum distance check.

2. **P**: Surface Passed Failure
   The surface might be so small that the cluster passes beside it.

3. **Fi**: Final Minimum Failure
   The resulting structure after the ghost atoms replaced by the surface atoms, fails to pass the minimum distance check after 'ck_surface_union_fails_limit' times.

Chapter 5

FF Fitting, Testing and Optimizing

5.1 Job names

The following job names for **execute** block will all be able to start FF fitting calculation, and are equivalent:

```
{ exe: ff }
{ exe: fitting }
{ exe: fit }
{ exe: fffit }
```

The following job names for **execute** block will all be able to start FF testing calculation, and are equivalent:

```
{ exe: test }
{ exe: testing }
{ exe: fftest }
```

The following job names for **execute** block will all be able to start FF optimization calculation, and are equivalent:

```
{ exe: opt }
{ exe: optimize }
{ exe: optimizing }
{ exe: optimization }
{ exe: relax }
{ exe: ffopt }
```

5.2 Parameters

The parameters for FF fitting, testing and optimizing can be classified to several groups. **CG** parameters control the accuracy of Nonlinear Conjugate Gradient calculation. **Functions** parameter controls the functions used to fit. **LJ** parameters control the description details of van der Waals interaction. **Fitting/Testing/Optimizing I/O** parameters control how to read/write files for these three jobs. **FF I/O** parameters control where to read/write the fitted formulas.

5.2.1 CG parameters

1. `cg_imax/cgimax` [type: **Integer**] [default: 200]
   The maximum number macro-iteration of CG calculation.
2. `cg_jmax/cgjmax` [type: **Integer**] [default: 10]
   The maximum number micro-iteration of CG calculation.
3. `cg_epsilon/cgeps` [type: **Double**] [default: 1e-7]
   The convergence threshold of macro-iteration of CG calculation.
4. `cg_epsilon_n/cgepsn` [type: **Double**] [default: 1e-6]
   The convergence threshold of micro-iteration of CG calculation.

5.2.2 Functions

AFFCK uses four types of Force Field terms: **constant**, **stretching**, **bending** and **van der Waals/LJ** terms. **Constant** terms are not controlled by parameters and they should always be there. The other three types can be respectively grouped into subtypes according to how many bonds are formed by the two (or three, for bending type) atoms in a pair collectively.

Actually the types can be further grouped by the element types involved. But this will be done by the program automatically and will not be controlled by parameters. Consider all these factors, we can assign an unique integer f to each type:

$$f = TypeBase + BondNumber + 1000 * ElementType \tag{5.2.1}$$

Typically, $BondNumber$ < $TypeBase$ < 1000 * $ElementType$. When determining parameters, we set $ElementType$ to zero, since the actual value will be determined automatically when running. $TypeBase$ has only three possible values: $TypeBase =$ 10 for **stretching** terms, $TypeBase = 30$ for **bending** terms, $TypeBase = 50$ for **LJ** terms. The parameter:

1. `functions/fun` [type: **IntegerList**] [default: 53 54 55 56 57 10 30]
   Specify the functions type used for FF fitting.

Note that when there is only one term listed for a certain type of *TypeBase*, this term will not be specified to a particular sum of bond number. In other word, All interaction of this kind, regardless of the sum bond number, will be fitted to the same coefficient. For example,

```
{ parameter: fun = 10 30 }
```

and

```
{ parameter: fun = 15 33 }
```

are equivalent. When there are a list of successive integers for a certain type of *TypeBase*, the sum of bond number will be considered, but the maximum integer of them will also include the sum of bond number larger than its *BondNumber*, while the minimum integer of them will also include the sum of bond number less than its *BondNumber*. For example,

```
{ parameter: fun = 53 54 55 56 57 }
```

means:

```
53 : LJ terms with sum of bond number <= 3
54 : LJ terms with sum of bond number  = 4
55 : LJ terms with sum of bond number  = 5
56 : LJ terms with sum of bond number  = 6
57 : LJ terms with sum of bond number >= 7
```

When the listed integers are not successive, the bond numbers that are not presented will be merged to the ones after them. In other words, they will share the same fitted coefficients with the terms with larger bond numbers, which are presented in the list. For example,

```
{ parameter: fun = 53 56 57 59 60 }
```

means:

```
53 : LJ terms with sum of bond number <= 3
56 : LJ terms with 4 <= sum of bond number <= 6
57 : LJ terms with sum of bond number  = 7
59 : LJ terms with 8 <= sum of bond number <= 9
60 : LJ terms with sum of bond number >= 10
```

or more explicitly, [1]

```
53 : LJ terms with sum of bond number = 2, 3
56 : LJ terms with sum of bond number = 4, 5, 6
57 : LJ terms with sum of bond number = 7
```

[1]the sum of bond number at least should be $1 + 1 = 2$ because otherwise, one of the two atoms must be isolated.

```
59 : LJ terms with sum of bond number = 8, 9
60 : LJ terms with sum of bond number = 10, 11, ...
```

The integer list is just a convenience notation of functions used by AFFCK. For the same system, the value of 'functions' must be consistent in Fitting, Testing, and Optimizing jobs.

5.2.3 LJ parameters

1. `lj_power/ljp` [type: **Integer**] [default: 9]
   Specify the negative power exponent of first term of LJ interaction. The power exponent of second term is fixed to -6.

2. `lj_distance/ljd` [type: **Integer**] [default: 1]
   The distance of atoms larger or equal than 'ljd' will be considered in LJ interaction. If 'ljd' = 1, all atom pairs will be considered for LJ interaction. If 'ljd' = 2, only atom pairs that are not directly connected by a bond will be considered.

5.2.4 Fitting I/O parameters

Note that though 'standard_list_file's type is **File**, it is not right to use 'seq' grammar here (see 2.3.6), since for fitting we always need a list that contains energies.

1. `standard_list_file/stdlf` [type: **File**] [default:]
   Specify the input list file.

2. `standard_xyz_files/stdxf` [type: **String**] [default:]
   Specify the input structures xyz files path pattern. Use '#' notation for various numbers (see 3.2). You should write full path here.

3. `standard_list_file_id_column/stdli` [type: **Integer**] [default: 0]
   Specify the column number of the input list file containing xyz file *id*s. Column number counts from zero.

4. `standard_list_file_energy_column/stdle` [type: **Integer**] [default: 1]
   Specify the column number of the input list file containing energies. The unit of energies is not important because we do not transform energy units in AFFCK internally.

5. `standard_list_sort/stdsort` [type: **Integer**] [default: 0]
   Specify how to sort the input data. If 'stdsort' = 0, the data will not be sorted. If 'stdsort' = 1, the data will be sorted by file *id*s. If 'stdsort' = 2, the data will be sorted by energies. Sorting itself usually does not make any sense. However, when you use 'start_number', 'accept_number' or 'accept_ratio' parameter,

some data at the head or tail of the list will be removed. Note that the sorting happens before removing terms.

6. `standard_list_start_number/stdstn` [type: **Integer**] [default: 0]
   Specify how many structures should be removed from the head of the list. This happens after the 'accept_number' or 'accept_ratio' parameter acts.

7. `standard_list_accept_number/stdacn` [type: **Integer**] [default: 99999]
   Specify how many structures from the head of the list should be remained.

8. `standard_list_accept_ratio/stdacr` [type: **Double**] [default: 1.00]
   Specify what ratio of structures from the head of the list should be remained. The number of finally accepted structures is:

$$N_{final} = \min(N_{acc}, N_{original} * R_{acc}) - N_{start} \tag{5.2.2}$$

   where N_{final} is the the number of finally accepted structures, N_{acc} is the value of 'accept_number', $N_{original}$ is the original number of structures, R_{acc} is the value of 'accept_ratio', and N_{start} is the value of 'start_number'.

9. `standard_fitted_output/stdout` [type: **String**] [default:]
   Specify the output list file. If not empty, the program will write the fitted energies of all structures there. Otherwise, the fitted energies will not be stored.

10. `standard_fitted_output_sort/stdosort` [type: **Integer**] [default: 0]
    Specify how to sort the output list. If 'stdosort' = 0, the list will not be sorted. If 'stdosort' = 1, the list will be sorted by file *id*s. If 'stdosort' = 2, the list will be sorted by energies.

5.2.5 Testing I/O parameters

Testing I/O parameters have just the same meaning as that of Fitting, despite the prefixes are different. Also since in some case one just need to get the fitted energies, the energies in the input list are unnecessary. You can set 'list_file_energy_column' to -1 and use 'seq' grammar for 'list_file' parameter.

1. `test_list_file/testlf` [type: **File**] [default:]

2. `test_xyz_files/testxf` [type: **String**] [default:]

3. `test_list_file_id_column/testli` [type: **Integer**] [default: 0]

4. `test_list_file_energy_column/testle` [type: **Integer**] [default: 1]

5. $\boxed{\texttt{test\_list\_sort/testsort}}$ [type: **Integer**] [default: 0]
6. $\boxed{\texttt{test\_list\_start\_number/teststn}}$ [type: **Integer**] [default: 0]
7. $\boxed{\texttt{test\_list\_accept\_number/testacn}}$ [type: **Integer**] [default: 99999]
8. $\boxed{\texttt{test\_list\_accept\_ratio/testacr}}$ [type: **Double**] [default: 1.00]
9. $\boxed{\texttt{test\_fitted\_output/testout}}$ [type: **String**] [default:]
10. $\boxed{\texttt{test\_fitted\_output\_sort/testosort}}$ [type: **Integer**] [default: 0]

5.2.6 Optimizing I/O parameters

Most of optimizing I/O parameters have just the same meaning as that of Fitting, despite the prefixes are different. Also since we do not need DFT energies for optimization, the energies in the input list are unnecessary. You can set 'list_file_energy_column' to -1 and use 'seq' grammar for 'list_file' parameter. We do not have the 'list_file_energy_column', 'fitted_output', and 'fitted_output_sort' parameter for optimizing.

1. $\boxed{\texttt{opt\_list\_file/optlf}}$ [type: **File**] [default:]
2. $\boxed{\texttt{opt\_xyz\_files/optxf}}$ [type: **String**] [default:]
3. $\boxed{\texttt{opt\_list\_file\_id\_column/optli}}$ [type: **Integer**] [default: 0]
4. $\boxed{\texttt{opt\_list\_sort/optsort}}$ [type: **Integer**] [default: 0]
5. $\boxed{\texttt{opt\_list\_start\_number/optstn}}$ [type: **Integer**] [default: 0]
6. $\boxed{\texttt{opt\_list\_accept\_number/optacn}}$ [type: **Integer**] [default: 99999]
7. $\boxed{\texttt{opt\_list\_accept\_ratio/optacr}}$ [type: **Double**] [default: 1.00]

However, optimization requires some extra parameters for storing the relaxed structures and energies:

1. $\boxed{\texttt{opt\_output\_directory/optod}}$ [type: **String**] [default:]
   Specify the directory for storing the relaxed structures.
2. $\boxed{\texttt{opt\_output\_xyz\_files/optox}}$ [type: **String**] [default:]
   Specify the filename pattern for storing the relaxed structures.

3. `opt_output_list_file/optol` [type: **String**] [default:]
   Specify the file path to write the initial fitted energies and final relaxed energies. If empty, the energies will not be written.

4. `opt_output_list_sort/optosort` [type: **Integer**] [default: 0]
   Specify how to sort the output list. If 'optosort' = 0, the list will not be sorted. If 'optosort' = 1, the list will be sorted by initial energies. If 'optosort' = 2, the list will be sorted by final energies.

5.2.7 FF I/O

Sometimes you might want to finish the FF fitting and store all the fitted coefficients, and further use the stored coefficients for another testing or optimizing job. **FF I/O** is used to read/write fitted coefficients.

The following job names for **execute** block will all be able to store FF coefficients to a file, and are equivalent:

```
{ exe: ffop }
{ exe: ffoutput }
```

The following job names for **execute** block will all be able to read FF coefficients from a file, and are equivalent:

```
{ exe: ffip }
{ exe: ffinput }
```

Note that you should always execute a FF output job after (not necessarily immediately after) a FF fitting job, and execute a FF input job before (not necessarily immediately before) a FF testing or optimization job. The following parameters are used to specify the file location:

1. `ff_input_file/ffipf` [type: **String**] [default:]
   Specify the file path for reading the FF coefficients.

2. `ff_output_file/ffopf` [type: **String**] [default:]
   Specify the file path for writing the FF coefficients.

For example, you can first run a FF Fitting and then store the coefficients:

```

! input file for a FF fitting and storing calculation

{ parameter:
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
   ff_output_file         = fftmp.txt;
}
{ execute: ff; ffop }
```

And then write and run another input file for FF optimization:

```
! input file for a FF optimization calculation

{ parameter:

   opt_list_file          = seq 0 99;
   opt_xyz_files          = ../test/data/singlet-dscf/pos_#.xyz;
   opt_output_directory   = data/opt;
   opt_output_xyz_files   = pos_#.xyz;
   opt_output_list_file   = data/opt/optimized_energy.txt;
   ff_input_file          = fftmp.txt;

}
{ execute: ffip; opt }
```

But note that the following input file might be dangerous:

```
{ parameter:

   file_override          = false;
   standard_list_file     = ../test/data/singlet-dscf/energy.txt;
   standard_xyz_files     = ../test/data/singlet-dscf/pos_#.xyz;
   standard_fitted_output = data/stdout.txt;
   opt_list_file          = seq 0 99;
   opt_xyz_files          = ../test/data/singlet-dscf/pos_#.xyz;
   opt_output_directory   = data/opt;
   opt_output_xyz_files   = pos_#.xyz;
   opt_output_list_file   = data/opt/optimized_energy.txt;
   ff_output_file         = fftmp.txt;
   ff_input_file          = fftmp.txt;

}
{ execute: ff; ffop; ffip; opt }
```

Since ‘file_override’ is false, the ‘ff_output_file’ can be redirected to somewhere else at runtime, but ‘ff_input_file’ does not change. In this case, the program will read the old version of coefficients.

5.3 Outputs

Part of the output on the screen is:

```
opt 1 of 100 (8) -954.606 -954.862
opt 2 of 100 (6) -954.551 -954.72
opt 3 of 100 (4) -954.518 -954.622
opt 4 of 100 (5) -954.61 -954.779
opt 5 of 100 (8) -954.617 -954.819
```

For each structure, we presented the serial number (not *id*) of the structure, the total number of structures, number of steps for optimization, initial FF energy, and final FF energy. Final structures will be written as separate files into ‘opt_output_directory’. And the *id*, and initial and final energies of each structure will be written into ‘opt_output_list_file’, given that this parameter is not empty.