CHAPTER

EIGHT

WORKFLOWS AND AUTOMATIZATION

8.1 ORCA Python Interface (OPI)

The ORCA Python interface (OPI) is a versatile tool that provides seamless access to ORCA calculations through Python. The OPI library offers many functions and classes to easily define calculation settings, perform different kinds of ORCA calculations, and process the resulting output data. It is open-source and freely available at GitHub. For detailed installation instructions and usage examples, please refer to the official OPI documentation.

Added in version 6.1.0: The first version of OPI was released with ORCA 6.1. Please note that OPI's open-source development is managed independently of the ORCA software itself. For the latest updates and changes, consult the OPI GitHub repository and the OPI documentation.

8.1.1 Installation

OPI can either be installed directly from PyPI:

```
pip install orca-pi
```

or from GitHub:

```
git clone https://github.com/faccts/opi.git
cd opi
python3 -m venv .venv
source .venv/bin/activate
python3 -m pip install .
```

In order to use OPI properly we recommend to install it in a venv environment. For more instructions refer to the OPI documentation.

8.2 Compound

Compound is a form of sophisticated scripting language that can be used directly in the input of ORCA. Using '*Compound*' the user can combine various parts of a normal ORCA calculation to evaluate custom functions of his own. In order to explain its usage, we will use an example. For a more detailed description of this module the user is referred to section *More Details on Compound*.

8.2.1 example

As a typical example we will use the constrained optimization describing the “umbrella effect” of NH_3. The script will perform a series of calculations and in the end it will print the potential of the movement plus it will identify the minima and the maximum. The corresponding compound script is the one shown below.

```
# ----------------------------------------------
# Umbrella coordinate mapping for NH3
# Author: Frank Neese
# ----------------------------------------------
variable JobName = "NH3-umbrella";
variable amin    = 50.0;
variable amax    = 130.0;
variable nsteps  = 21;
Variable energies[21];

Variable angle;
Variable JobStep;
Variable JobStep_m;
variable step;

Variable method = "BP86";
Variable basis  = "def2-SVP def2/J";

step  = 1.0*(amax-amin)/(nsteps-1);

# Loop over the number of steps
# ----------------------------
for iang from 0 to nsteps-1 do
  angle    = amin + iang*step;
  JobStep  = iang+1;
  JobStep_m= JobStep-1;
  if (iang>0) then
    Read_Geom(JobStep_m);
    New_step
      ! &{method} &{basis} TightSCF Opt
      %base "&{JobName}.step&{JobStep}"
      %geom constraints
        {A 1 0 2 &{angle} C}
        {A 1 0 3 &{angle} C}
        {A 1 0 4 &{angle} C}
        end
      end

    Step_End
  else
    New_step
      ! &{method} &{basis} TightSCF Opt
      %base "&{JobName}.step&{JobStep}"
      %geom constraints
        {A 1 0 2 &{angle} C}
        {A 1 0 3 &{angle} C}
        {A 1 0 4 &{angle} C}
        end
      end

      * int 0 1
      N  0 0 0 0.0 0.0 0.0
      DA 1 0 0 2.0 0.0 0.0
      H  1 2 0 1.06 &{angle} 0.0
      H  1 2 3 1.06 &{angle} 120.0
      H  1 2 3 1.06 &{angle} 240.0
```

(continues on next page)

(continued from previous page)

```
      *
    Step_End
  endif
   Read energies[iang] = SCF_ENERGY[jobStep];
   print(" index: %3d Angle %6.2lf Energy: %16.12lf Eh\n", iang, angle,␣
↪energies[iang]);
EndFor

# Print a summary at the end of the calculation
# ---------------------------------------------
print("////////////////////////////////////////////////////////////\n");
print("// POTENTIAL ENERGY RESULT\n");
print("////////////////////////////////////////////////////////////\n");
variable minimum,maximum;
variable Em,E0,Ep;
variable i0,im,ip;
for iang from 0 to nsteps-1 do
  angle   = amin + 1.0*iang*step;
  JobStep = iang+1;
  minimum = 0;
  maximum = 0;
  i0      = iang;
  im      = iang-1;
  ip      = iang+1;
  E0      = energies[i0];
  Em      = E0;
  Ep      = E0;
  if (iang>0 and iang<nsteps-1) then
    Em = energies[im];
    Ep = energies[ip];
  endif
  if (E0<Em and E0<Ep) then minimum=1; endif
  if (E0>Em and E0>Ep) then maximum=1; endif
  if (minimum = 1 ) then
    print(" %3d  %6.2lf %16.12lf (-)\n",JobStep,angle, E0 );
  endif
  if (maximum = 1 ) then
    print(" %3d  %6.2lf %16.12lf (+)\n",JobStep,angle, E0 );
  endif
  if (minimum=0 and maximum=0) then
    print(" %3d  %6.2lf %16.12lf    \n",JobStep,angle, E0 );
  endif
endfor
print("////////////////////////////////////////////////////////////\n");

End # end of compound block
```

Let's start with how somebody can execute this input. In order to run it, the easiest way is to save it in a normal text file, using the name "umbrella.cmp" and then use the following ORCA input file:

```
%Compound "umbrella.cmp"
```

nothing more is needed. ORCA will read the compound file and act appropriately.

A few notes about this ORCA input. First, there is no simple input line, (starting with *"!"*). A simple input is not required when one uses the *Compound* feature, but In case the user adds a simple input, all the information from the simple input will be passed to the actual compound jobs.

In addition, if one does not want to create a separate compound text file, it is perfecly possible in ORCA to use the compound feature as any other ORCA block. This means that after the *%Compound* directive, instead of giving the filename one can append the contents of the Compound file.

As we will see, inside the compound script file each compound job can contain all information of a normal ORCA input file. There are two very important exceptions here: The number of processors and the *MaxCore*. These information should be set in the initial ORCA input file and not in the actual compound files.

The Compound block has the same structure like all ORCA blocks. It starts with a *"%"* and ends with *"End"*, if the input is not read from a file. In case the compound directives are in a file, as in the example above, then simply the filename inside brackets is needed and no final *END*.

8.2.2 Defining variables

As we pointed out already, it is possible to either give all the information for the calculations and the manipulation of the data inside the Compound block or create a normal text file with all the details and let ORCA read it. The latter option has the advantage that one can use the same file with more than one geometries. In the previous example we refer ORCA to an external file. The file *"umbrella.cmp"*, that contains all necessary information.

Let's try to analyse now the Compound *"umbrella.cmp"* file.

```
# ----------------------------------------------
# Umbrella coordinate mapping for NH3
# Author: Frank Neese
# ----------------------------------------------
variable JobName = "NH3-umbrella";
variable amin    = 50.0;
variable amax    = 130.0;
variable nsteps  = 21;
Variable energies[21];

Variable angle;
Variable JobStep;
Variable JobStep_m;
variable step;

Variable method = "BP86";
Variable basis  = "def2-SVP def2/J";

step  = 1.0*(amax-amin)/(nsteps-1);
```

The first part contains some general comments and variable definitions. For the comments we use the same syntax as in the normal ORCA input, through the *"#"* symbol. Plase not that more than one *"#"* symbols in the same line cause an error.

After the initial comments we see some declarations and definitions. There are many different ways to declare variables described in detail in section *Variables - General*.

All variable declarations begin with the directive *Variable* which is a sign for the program to expect the declaration of one or more new variables. Then there are many options, including defining more than one variable, assigning also a value to the variable or using a list of values. Nevertheless all declarations **MUST** finish with the *;* symbol. This symbol is a message to the program that this is the end of the current command. The need of the *;* symbol in the end of each command is a general requirement in *Compound* and there are only very few exceptions to it.

8.2.3 Running calculations

```
# Loop over the number of steps
# ----------------------------
for iang from 0 to nsteps-1 do
  angle    = amin + iang*step;
  JobStep  = iang+1;
  JobStep_m= JobStep-1;
  if (iang>0) then
    Read_Geom(JobStep_m);
    New_step
```

(continues on next page)

(continued from previous page)

```
      ! &{method} &{basis} TightSCF Opt
      %base "&{JobName}.step&{JobStep}"
      %geom
        constraints
          {A 1 0 2 &{angle} C}
          {A 1 0 3 &{angle} C}
          {A 1 0 4 &{angle} C}
        end
      end
    Step_End
  else
    New_step
      ! &{method} &{basis} TightSCF Opt
      %base "&{JobName}.step&{JobStep}"
      %geom
        constraints
          {A 1 0 2 &{angle} C}
          {A 1 0 3 &{angle} C}
          {A 1 0 4 &{angle} C}
        end
      end
      * int 0 1
        N 0 0 0 0.0 0.0 0.0
        DA 1 0 0 2.0 0.0 0.0
        H 1 2 0 1.06 &{angle} 0.0
        H 1 2 3 1.06 &{angle} 120.0
        H 1 2 3 1.06 &{angle} 240.0
      *
    Step_End
  endif
  Read energies[iang] = SCF_ENERGY[jobStep];
  print(" index: %3d Angle %6.2lf Energy: %16.12lf Eh\textbackslash{}n", iang,␣
↪angle, energies[iang]);
EndFor
```

Then we have the most information dense part. We start with the definition of a *for* loop. The syntax in compound for *for* loops is:

For *variable* **From** *startValue* **To** *endValue* **Do**
directives
EndFor

As we can see in the example above, the *startValue* and *endValue* can be constant numbers or previously defined variables, or even functions of these variables. Keep in mind that they have to be integers. The signal that the loop has reached it's end is the *EndFor* directive. For more details with regard to the *for* loops please refer to section For.

Then we proceed to assign some variables.

```
angle     = amin + iang*step;
JobStep   = iang+1;
JobStep_m = JobStep-1;
```

The syntax of the variable assignement is like in every programming language with a variable, followed with the = symbol and then the value or an equation. Please keep in mind, that the assignement **must** always finish with the *;* symbol.

The next step is another significant part of every programming language, namely the *if* block. The syntax of the *if* block is the following:

if (*expression to evaluate* **)** **Then**
directives
else if (*expression to evaluate* **)** **Then**
directives

else
directives
EndIf

The *else if* and *else* parts of the block are optional but the final *EndIf* must always signal the end of the *if* block. For more details concerning the usage of the *if* block please refer to section *If* of the manual.

Next we have a command which is specific for compound and not a part of a normal programming language. This is the *ReadGeom* command. It's syntax is:

Read_Geom(*integer value*);

Before explaining this command we will proceed with the next one in the compound script and return for this one.

The next command is the basis of all compound scripts. This is the *New_Step* Command. This command signals compound that a normal ORCA calculation follows. It's syntax is:

New_Command *Normal ORCA input* **Step_End**

Some comments about the *New_Step* command. Firstly, inside the *New_Step* - *Step_End* commands one can add all possilbe commands that a normal ORCA input accepts. We should remember here that the commands that define the number of processors and the *MaxCore* command will be ignored.

A second point to keep in mind is the idea of the *step*. Every *New_Step* - *Step_End* structure corresponds to a step, starting counting from 1 (The first ORCA calculation). This helps us define the property file that this calculation will create, so that we can use it to retrieve information from it.

A significant feature in the *New_Step* - *Step_End* block. is the usage of the structure **&{*variable*}** . This structure allows the user to use variables that are defined outside the *New_Step* - *Step_End* block inside it, making the ORCA input more generic. For example, in the script given above, we build the *basename* of the calculations

```
  %base "&{JobName}.step&{JobStep}"
```

using the defined variables *JobName* and *JobStep*. For more details regarding the usage of the **&{}** structure please refer to section *&* while for the *New_Step* - *Step_End* structure please refer to the section *NewStep*.

Finally, a few comments about the geometries of the calculation. There are 3 ways to provide a geometry to a *New_Step* - *Step_End* calculation. The first one is the traditional ORCA input way, where we can give the coordinates or the name of a file with coordinates, like we do in all ORCA inputs. In *Compound* though, if we do not pass any information concerning the geometry of the calculation, then *Compound* will automatically try to read the geometry of the previous calculation. This is the second (implicit) way to give a geometry to a compound Step. Then there is a third way and this is the one we used in the example above. This is the **Read_Geom** command. The syntat of this command is:
Read_Geom (*Step number*);
We can use this command when we want to pass a specific geometry to a calculation that is not explicitly given inside the *New_Step* - *Step_End* structure and it is also not the one from the previous step. Then we just pass the number of the step of the calculation we are interesting in just before we run our new calculation. For more details regarding the *Read_Geom* command please refer to section *Read_Geom*.

8.2.4 Data manipulation

One of the most powerfull features of *Compound* is it's direct access to properties of the calculation. In order to use these properties we defined the *Read* command. In the previous example we use it to read the SCF energy of the calculation:

```
Read energies[iang] = SCF\_ENERGY[jobStep];
```

The syntax of the command is:

Read *variable name* **=** *property*

where *variable name* is the name of a variable that is already defined, *property* is the property from the known properties found in table *List of known Properties* and *step* is the step of the calculation we are interested in. For more details in the *Read* command please refer to section sec:workflowsautomatization.compound_detailed.commands.propertyFile.read.

```
# Print a summary at the end of the calculation
  # -------------------------------------------
  print("//////////////////////////////////////////////////////////\\n");
  print("// POTENTIAL ENERGY RESULT\\n");
  print("//////////////////////////////////////////////////////////\\n");
  variable minimum,maximum;
  variable Em,E0,Ep;
  variable i0,im,ip;
  for iang from 0 to nsteps-1 do
    angle   = amin + 1.0*iang*step;
    JobStep = iang+1;
    minimum = 0;
    maximum = 0;
    i0      = iang;
    im      = iang-1;
    ip      = iang+1;
    E0      = energies[i0];
    Em      = E0;
    Ep      = E0;
    if (iang>0 and iang<nsteps-1) then
      Em = energies[im];
      Ep = energies[ip];
    endif
    if (E0<Em and E0<Ep) then minimum=1; endif
    if (E0>Em and E0>Ep) then maximum=1; endif
    if (minimum = 1 ) then
      print(" %3d  %6.2lf %16.12lf (-)\textbackslash{}n",JobStep,angle, E0 );
    endif
    if (maximum = 1 ) then
      print(" %3d  %6.2lf %16.12lf (+)\textbackslash{}n",JobStep,angle, E0 );
    endif
    if (minimum=0 and maximum=0) then
      print(" %3d  %6.2lf %16.12lf    \textbackslash{}n",JobStep,angle, E0 );
    endif
  endfor
  print("//////////////////////////////////////////////////////////\\n");
```

Once all data are available we can use them in equations like in any programming language.

The syntax of the print statement is:

print(*format string, [variables]* **);**

For example in the previous script we use it like:

```
print(" %3d  %6.2lf %16.12lf  \n",JobStep,angle, E0 );
```

where *%3d, %6.2lf* and *%16.2lf* are format identifiers and *JobStep, angle* and *E0* are previously defined variables. The syntax follows closely the widely accepted syntax of the *printf* command in the programming language C. For more details regarding the *print* statememnt please refer to section: *Print*.

Similar to the *print* command are the *write2file* and *write2string* commands that are used to write instead of the output file, either to a file we choose or to produce a new string.

Finally it is really importnat not to forget that every compound file should finish with a final **End**.

Once we run the previous example we get the following output:

```
////////////////////////////////////////////////////////
// POTENTIAL ENERGY RESULT
////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
1   50.00 -56.486626696200
2   54.00 -56.498074637200
3   58.00 -56.505200120800
4   62.00 -56.508823168800
5   66.00 -56.509732863600 (-)
6   70.00 -56.508724734300
7   74.00 -56.506590613800
8   78.00 -56.504070086000
9   82.00 -56.501791816800
10   86.00 -56.500229017900
11   90.00 -56.499674856600 (+)
12   94.00 -56.500229018100
13   98.00 -56.501791817200
14  102.00 -56.504070082800
15  106.00 -56.506590613300
16  110.00 -56.508724733100
17  114.00 -56.509732863700 (-)
18  118.00 -56.508823172900
19  122.00 -56.505200132200
20  126.00 -56.498074642900
21  130.00 -56.486626729200
/////////////////////////////////////////////////////////
```

with the step, the angle for the corresponding step, the energy of the constrained optimized energy plus the symbols for the two minima and the maximum in the potential.

8.3 More Details on Compound

In this part we describe “Compound” in more detail.

8.3.1 Commands

Below is a list of all available commands available in *Compound*

———————————— Dataset Related ————————————

- Dataset(*Dataset*)
- MakeReferenceFromDir(*D.MakeReferenceFromDir*)
- Print(*D.Print*)

——————————— File Handling Related ———————————

- CloseFile (*CloseFile*)
- OpenFile (*OpenFile*)

———————————- *For* Loop Related ———————————-

- Break (*Break*)
- Continue (*Continue*)
- EndFor (*EndFor*)
- For (*For*)

———————————— Geometry Related ———————————-

- Geometry(*Geometry*)
- BohrToAngs(*G.BohrToAngs*)

- CreateBSSE(*G.CreateBSSE*)
- CreateFragments(*G.CreateFragments*)
- DisplaceAtom(*G.DisplaceAtom*)
- FollowNormalMode(*G.FollowNormalMode*)
- GetAngle(*G.GetAngle*)
- GetAtomicNumbers(*G.GetAtomicNumbers*)
- GetBondDistance(*G.GetBondDistance*)
- GetCartesians(*G.GetCartesians*)
- GetGhostAtoms(*G.GetGhostAtoms*)
- GetNumOfAtoms(*G.GetNumOfAtoms*)
- MoveAtomToCenter(*G.MoveAtomToCenter*)
- Read(*G.Read*)
- RemoveAtoms(*G.RemoveAtoms*)
- RemoveElements(*G.RemoveElements*)
- WriteXYZFile(*G.WriteXYZFile*)

——————————— GOAT Related ————————–

- GOAT(*GOAT*)
- Get_Energy (*Goat.Get_Energy*)
- Get_Num_Of_Geometries (*Goat.Get_Num_Of_Geometries*)
- ParseEensembleFfile (*Goat.ParseEnsembleFile*)
- Set_Basename (*Goat.Set_Basename*)
- Print (*Goat.Print*)
- WriteXYZFile (*Goat.WriteXYZFile*)

——————————— *If* block Related ————————–

- If (*If*)

——————————– Linear Algebra Related ————————–

- Diagonalize(*Diagonalize*)
- InvertMatrix(*InvertMatrix*)
- Mat_p_Mat(*Mat_p_Mat*)
- Mat_x_Mat(*Mat_x_Mat*)
- Mat_x_Scal(*Mat_x_Scal*)

——————————— ORCA calculation Related ————————–

- Basenames(*Basenames*)
- ReadMOs(*ReadMOs*)

——————————— Program flow Related ————————–

- Abort (*Abort*)
- End (*End*)
- EndRun (*EndRun*)
- GoTo (*GoTo*)

————————— **Property File Related** —————————-

- GetNumOfInstances(*GetNumOfInstances*)
- ReadProperty(*ReadProperty*)

————————— **String Handling Related** —————————

- GetBasename (*S.GetBasename*)
- GetChar (*S.GetChar*)
- GetSuffix (*S.GetSuffix*)
- Print (*Print*)
- Write2File (*Write2File*)
- Write2String (*Write2String*)

————————— **Step Related** —————————

- & (*&*)
- Alias (*Alias*)

————————— **Timer Related** —————————

- Timer (*Timer*)
- Last (*T.Last*)
- Reset (*T.Reset*)
- Start (*T.Start*)
- Stop (*T.Stop*)
- Total (*T.Total*)

————————— **Variables Related** —————————-

- Variables (*Variables - General*)
- Variables - Assignment (*Variables - Assignment*)
- Variables - Declaration (*Variables - Declaration*)
- Variables - Functions (*Variables - Functions*)
- Variable - With (*With*)
- GetBool() *V.GetBool()*
- GetDim1() *V.GetDim1()*
- GetDim2() *V.GetDim2()*
- GetDouble() *V.GetDouble()*
- GetInteger() *V.GetInteger()*
- GetSize() *V.GetSize()*
- GetString() *V.GetString()*
- PrintMatrix() *V.PrintMatrix()*
- Write2File (*Write2File*)
- Write2String (*Write2String*)

&

The *&* symbol has a special meaning in the compound block. Using this symbol inside the *NewStep* - *StepEnd* block the user can use variables that are defined outside the block. Both string and numerical variables are allowed.

Syntax:

&{*variable*}

Example

```
# -----------------------------------------------
# This script checks the options for the
#              '&' symbol
# -----------------------------------------------
%Compound

  Variable method   = "BP86";              #string variable'
  Variable basis    = "def2-TZVP def2/J";
  Variable name     = "base";
  Variable number   = 0;                   #integer variable
  Variable distance = 0.8;                 #double variable
  New_step
    ! &{method} &{basis} TightSCF
    %base "&{name}_&{number}"              #combination of variables
    *xyz 0 1
      H 0.0 0.0 0.0
      H 0.0 0.0 &{distance}
    *
  Step_End
  # ---------------------------------------------
  # Add some printing
  # ---------------------------------------------
  print("SUMMARY OF VARIABLES\n");
  print("Method:   %s\n",    method);
  print("Basis:    %s\n",    basis);
  print("Name:     %s\n",    name);
  print("Number:   %d\n",    number);
  print("Distance: %.2lf\n", distance);
End
```

Abort

Abort is used when the user wants to exit the program instantly. **Syntax:**
Abort;
or alternatively:
Abort

Example:

```
%Compound
for i from 0 to 4 do
  print("i: %d\n", i);
  if (i=2) then
    abort;
  endif
endfor
End
```

Alias

Alias is used to replace an integer number with a more representative string. It is useful when one performs more than one calculations and the step numbers become too complicated to evaluate. In this case using Alias_Step after the Step_End command will connect the preceeding calculation step number with the provided name.

Syntax:
Alias *name*;
or alternatively:
Alias *name*
Example:

```
# ----------------------------------
# This script checks 'alias' keyword
# ----------------------------------
Variable numOfSteps = 20;
Variable Range      = 4.0;
Variable distStart  = 0.4;
Variable Step       = Range/numOfSteps;
Variable distance;
Variable Energies[numOfSteps];
Variable simpleInput  = "BP86 def2-SVP def2/J";

For index from 0 to numOfSteps-1 Do
  Distance =  distStart+ index*Step;
  New_Step
    !&{simpleInput}
    *xyz 0 1
      H 0.0 0.0 0.0
      H 0.0 0.0 &{Distance}
    *
  Step_End
  Alias currStep;
  print("Current step: %d\n", currStep);
  Read Energies[index] = JOB_INFO_TOTAL_EN[currStep];
EndFor

print("--------------------------------------\n");
print("       Compound Printing              \n");
print("%s  %12s  %16s \n","Step",  "Distance", "Energy");
print("--------------------------------------\n");
For index from 0 to numOfSteps - 1 Do
  Distance = distStart + index*Step;
  print("%4d  %12.4lf   %16.8lf \n", index, Distance, Energies[index]);
EndFor

End
```

Note

For the **Alias** command the final ';' is optional.

Basenames

Basenames in a variable that is automatically created in *Compound* everytime a *NewStep* is used and it holds the name of this step. It can be used to recover this name during running of a calculation. It practically is a vector were value zero corresponds to the main calculation before any *new_step* command.

Example:

```
*xyz 0 1
  H 0.0 0.0 0.0
  H 0.0 0.0 0.8
*
%Compound
  new_step
    !BP86
  step_end
  alias lala;
  print("Basaenames[0]: %s\n", basenames[0]);
  print("Basaenames[1]: %s\n", basenames[lala]);
End
```

Break

Break can be used inside a *For* loop (see (see *For*)) when one needs to break the loop under certain conditions. The syntax is the following:

Syntax:
For variable From Start value To End value Do
commands
break ;
commands
EndFor;

NOTE both versions *break* and *break;* are legal.

What break actually does is to set the running index of the loop to the last allowed value and then jump to the *EndFor* (see *EndFor*)).

Example:

```
#
# This a script to check Compound 'break' command
#
%Compound
print(" Test for 'break'\n");
print(" It should print 0, 1 and 2\n");
for i from 0 to 6 Do
  if (2*i > 4) then
    break
  endIF
  print("index: %d\n", i);
EndFor

print("Continued outside the 'for' loop\n");

End
```

CloseFile

When a file is opened in *Compound* using the *openFile* command (see *OpenFile*), then it must be closed using the *closeFile* command.

Syntax:
closeFile(file);

file is the file pointer created from the *openFile* command. For an example see paragraph *OpenFile*.

Continue

Continue can be used inside a *For* loop (see (see *For*)) when one needs to skip the current step of the loop and proceed to the next one. The syntax is the following:

Syntax:
For variable From Start value To End value Do
commands
continue ;
commands
EndFor;

NOTE both versions *continue* and *continue;* are legal.

What continue actually does is to jump to the *EndFor* (see *EndFor*)).

Example:

```
# -----------------------------------------------------
# This is a script to check Compound 'continue' command
# ----------------------------------------------------
%Compound
print(" Test for 'continue'\n");
print(" It should print 0, 1, 2 and 4\n");
for i from 0 to 4 Do
  if ( i=3) then
    continue;
  endIf
  print("index: %d\n", i);
EndFor
End
```

Dataset

In *Compound* we have *Dataset* objects. These objects can be treated like normal variables of type '*compDataset*'. An important difference between normal variables and *Dataset* variables is the declaration. Instead of the normal:

```
Variable x;
```

we excplicitly have to declare that this is a dataset. So the syntax for a dataset declaration is:

Syntax:

```
Dataset mySet;
```

NOTE in the case of datasets we do not allow multiple dataset declarations per line.

Below is a list of functions that work on *Dataset*.

- MakeReferenceFromDir(*D.MakeReferenceFromDir*)
- Print(*D.Print*)

Example:

```
# -----------------------------------------------------
# This is an example script for dataset definition
# ------------------------------------------------------
%Compound
  dataset mySet;
  mySet.Print();
End
```

D.MakeReferenceFromDir

MakeReferenceFromDir command acts on a dataset object (see *Dataset*). It creates a json reference file based on the *xyz files of the current folder. **NOTE** By default all charges and multplicities will be set to 0 and 1 respectively.

Syntax:
mySet.MakeReferenceFromDir(dirName);

Where:

- *mySet* is a dataset object that is already declared
- *dirName* The name of a directory that should contain some xyz files.

Example:

```
%Compound
  # ------------------------------------
  #   This is a compound script that will
  #   check dataset.MakeReferenceFromDir
  #   function
  #   NOTE: The script assumes that some
  #         xyz files rest in the current
  #         directory
  # ------------------------------------

  # First some definitions
  Variable name = "mySet";
  Variable numOfMolecules = 0;
  dataset mySet;
  Variable myDir="./";
  mySet.MakeReferenceFromDir(myDir);
  mySet.ReadReferenceFile();
  mySet.Print();
End
```

D.Print

Print command acts on a dataset object (see *Dataset*). It prints all details of the specific dataset object.

Syntax:
mySet.Print();

Where:

- *mySet* is a dataset object that is already declared

Example:

```
# -----------------------------------------------------
# This is an example script for dataset definition
# ------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
%Compound
  dataset mySet;
  mySet.Print();
End
```

Diagonalize

Compound can peform matrix algebraic operations, one of the available algebraic operation is matrix diagonalization. Be carefull that the matrix, that is to be diagonalized, **must be** a square symmetric matrix. It is also important to remember that only the upper triangle part of the matrix will be used for the diagonalization. If everything proceeds smoothly then the function will return the eigenvectors and eigenvalues of the matrix.

Syntax:
A.Diagonalize(eigenValues, eigenVectors);

Where:

- *A:* The matrix to be diagonalized.
- *eigenValues:* The vector with the eigenvalues
- *eigenVectors:* The square matrix with the eigenvectros of the initial matrix.

Example:

```
# -----------------------------------------------------
# This is an example script for diagonalization
# ------------------------------------------------------
%Compound
  Variable Dim=3;
  Variable A[Dim][Dim];
  Variable eigenVal;
  Variable eigenVec;
  for i from 0 to Dim-1 Do
    for j from 0 to Dim-1 Do
      if (i<=j) then
        A[i][j] = i+j+1;
      else
        A[i][j] =  A[j][i];
      EndIf
    EndFor
  EndFor
  A.Diagonalize(eigenVal, eigenVec);
  A.PrintMatrix();
  eigenVal.PrintMatrix();
  eigenVec.PrintMatrix();
End
```

End

End is the final command each compound script must have (unless there is an *EndRun* command (see *EndRun*)). After *Compound* executes what is written in the script then it passes control again to normal ORCA input reading. ORCA will continue analyze the input that rests after the *Compound* part but it will not run any calculation.

### EndRun

#EndRun* is an alternative to the *End* command (see *EndRun*) for ending the execution of a *Compound* script. The difference between *end* and *EndRun* is that *EndRun* ignores everything after the *Compound* block. This makes it even easier to use *Compound* as a full workflow run.

EndFor

All *For* loops (see *For*) must finish with *EndFor*. The syntax and an example is shown in the *For* section (see *For*).

NOTE For *EndFor* both *EndFor* and *EndFor;* are possible.

For

For loops are used to perform repetitive tasks. The syntax is the following:

Syntax:

For *variable* **From** *Start value* **To** *End value* **Do**

commands

EndFor or **EndFor;**

Variable should be a variable name not previously defined. *Start value* and *End value* should be integers defining the start and end value of the *variable*. *Start value* and *End value* can be numbers, predefined variables or functions of previously defined variables. The only requirement is that they should be **integers**. Keep in mind that the loop will be performed from the first value to the *End value*, including the *End value*.

Example:

```
# ---------------------------------------
# This is a script to check 'for' loops
# ---------------------------------------

# ---------------------------------
# Some necessary initial definitions
# ----------------------------------
Variable x = {0.0, 1.0, 2.0, 3.0, 4.0};
Variable f;
Variable loopStart;
Variable upLimit;

# -----------------------------------
# Case 1.
# Constant Start / Constant End
# -----------------------------------
print(" --------------  Case 1 --------------\n");
print("     Constant Start / Constant End    \n");
print("          f = index*x[index]          \n");
print( "      for index from 0 to 4 Do      \n");
print(" -------------------------------------\n");
for index from 0 to 4 Do
  f = index*x[index];
  print("Index: %3d    x[index]:  %.2lf   f:   %.2lf\n", index, x[index], f);
EndFor

loopStart = 0;
upLimit   = 4;
print(" --------------  Case 2 --------------\n");
print("     Variable Start / Variable End    \n");
print("          f = index*x[index]          \n");
```

(continues on next page)

(continued from previous page)

```
print( "   for index from loopStart to upLimit Do\n");
print(" ------------------------------------\n");
for index from loopStart to upLimit Do
  f = index*x[index];
  print("Index: %3d    x[index]:  %.2lf   f:   %.2lf\n", index, x[index], f);
EndFor


loopStart  = 1;
upLimit    = 3;
print(" --------------  Case 3 --------------    \n");
print("     function Start / function End        \n");
print("          f = index*x[index]              \n");
print( "   for index from start-1 to upLimit+1 Do\n");
print(" -------------------------------------    \n");
for index from loopStart-1 to upLimit+1 Do
  f = index*x[index];
  print("Index: %3d    x[index]:  %.2lf   f:   %.2lf\n", index, x[index], f);
EndFor

End
```

Geometry

In *Compound* we have *Geometry* objects. These objects can be treated like normal variables of type '*compGeometry*'. An important difference between normal variables and *Geometry* variables is the declaration. Instead of the normal:

```
Variable myGeom;
```

we excplicitly have to declare that this is a geometry. So the syntax for a geometry declaration is:

Syntax:

```
Geometry myGeom;

Geometry myGeom1, myGeom2;
```

Using the second definition one can define two geometry objects in the same line.

Below is a list of functions that work on *Geometry* objects.

- BohrToAngs(*G.BohrToAngs*)
- CreateBSSE (*G.CreateBSSE*)
- CreateFragments(*G.CreateFragments*)
- DisplaceAtom(*G.DisplaceAtom*)
- FollowNormalMode (*G.FollowNormalMode*)
- GetAngle(*G.GetAngle*)
- GetAtomicNumbers(*G.GetAtomicNumbers*)
- GetBondDistance(*G.GetBondDistance*)
- GetCartesians(*G.GetCartesians*)
- GetGhostAtoms(*G.GetGhostAtoms*)
- GetNumOfAtoms(*G.GetNumOfAtoms*)
- MoveAtomToCenter(*G.MoveAtomToCenter*)
- Read(*G.Read*)

- RemoveAtoms(*G.RemoveAtoms*)
- RemoveElements(*G.RemoveElements*)
- WriteXYZFile(*G.WriteXYZFile*)

G.BohrToAngs

BohrToAngs command acts on a geometry object (see see *Geometry*). It will transform the geometry of the loaded geometry object from Bohr to Angstroms. Practically it will just multiply the coordinates with the factor *0.529177249*.

Syntax:
myGeom.BohrToAngs();

Where:

- *myGeom* is a geometry object that already contains a geometry

Example:

```
# ----------------------------------------------------
# This is a script to check the BohrToAngs function
# -----------------------------------------------------
*xyz 0 1
  O      -1.69296787   -0.05579265    0.00556629
  H      -2.01296504    0.84704339   -0.01586469
  H      -0.73325076    0.04238910    0.00084302
*

%Compound
  Geometry myGeom;
  Variable CC;
  New_Step
    !BP86
  Step_End
  myGeom.Read();
  myGeom.BohrToAngs();
  CC = myGeom.GetCartesians();
  CC.PrintMatrix();
End
```

G.CreateBSSE

CreateBSSE command acts on a geometry object (see see *Geometry*). In the case that the geometry object contains *ghost* atoms then *CreateBSSE* will create five new files:

- myFilename_FragmentA.xyz
- myFilename_MonomerA.xyz
- myFilename_FragmentB.xyz
- myFilename_MonomerB.xyz
- myFilename_Total.xyz

Syntax:
myGeom.CreateBSSE(filename=myFilename);

Where:

- *myGeom* is a geometry object that already contains a geometry
- *filename* is a base filename for the created files.

Example:

```
# -----------------------------------------------------------
# This is a scrtipt to check the geom.CreateBSSE command
# ----------------------------------------------------------

*xyz 0 1
o:      -1.69296787   -0.05579265    0.00556629
h:      -2.01296504    0.84704339   -0.01586469
h:      -0.73325076    0.04238910    0.00084302
o        1.23009925    0.02698440   -0.00375550
h        1.60672086   -0.41139567    0.76236888
h        1.60236356   -0.44922858   -0.74915800
*

%Compound
  Geometry monomerA;
  variable myFilename     = "BSSE";
  Variable method         = "BP86";

  # ---------------------------------------
  # Calculation for Fragment A
  # ---------------------------------------
  New_Step
    !&{method}
  Step_End

  # --------------------------------------
  # Read the geometry of Fragment A
  # --------------------------------------
  monomerA.Read();

  # --------------------------------------
  # Create the missing xyz files
  # --------------------------------------
  monomerA.CreateBSSE(filename=myFilename);

End
```

NOTE The files will contain XYZ geometries in **BOHRS**.

G.CreateFragments

CreateBSSE command acts on a geometry object (see see *Geometry*). In the case that the geometry object contains *ghost* atoms then *CreateBSSE* will create five new files:

- myFilename_FragmentA.xyz
- myFilename_MonomerA.xyz
- myFilename_FragmentB.xyz
- myFilename_MonomerB.xyz
- myFilename_Total.xyz

Syntax:
myGeom.CreateBSSE(filename=myFilename);

Where:

- *myGeom* is a geometry object that already contains a geometry
- *filename* is a base filename for the created files.

Example: