PyPol Manual

Nicholas Francia

January 11, 2021

Chapter 1

Installation

This library works only on Linux OS.

1.1 Requirements: Python programs

All required dependencies should be installed before running PyPol:

- Python3
- numpy & scipy
- matplotlib
- openbabel
- progressbar2
- networkx

1.2 Requirements: Non-Python programs

All the following programs should be installed before running PyPol:

- GROMACS(>=2020)
- AmberTools
- PLUMED2 with the crystallization module

- PLUMED2 - Hack The Tree
- InterMol

When you first use PyPol it will ask for the paths of all these programs binaries and save them for future use. You can insure Python can find pypol.py by adding The PyPol directory to PYTHONPATH with the command:

```
export PYTHONPATH=''$PYTHONPATH:<path_to_pypol>/PyPol/''
```

If you set the path to this file as environment variables, you only have to do it once. You can add the same string to your *.bashrc* file.

Chapter 2

Quick Overview

This chapter is a quick introduction to PyPol and its features.

2.1 Create and manage a new project

The project object contains all the relevant information about the simulations and the different methods used. Initially, you have to create a new Project object by specifying the path of new directory. The function pypol.new_project will make that directory plus a series of other directories that will be used by the program. These are:

- Input: stores the initial structures and input files for the different methods
- Output: stores the output files from pypol modules
- data: contain all the structure folders in which simulations are performed

Most of the objects in pypol have help() functions that shows the accessible attributes and modules together with some examples. Everytime the Project object is modified you must write ¡Project¿.save() at the end of your script to save every variation to a pickle file in the project folder.
- Create a new project and print the help() function:

```
from PyPol import pypol as pp
# Creates a new project in folder /home/Work/Project/
project = pp.new_project(r'/home/Work/Project/', name=project1)
# Print available attributes, methods and examples
project.help()
# Save project to be used later
project.save()
```

Once saved, the Project object can be retrieved by loading it with the function pypol.load_project specifying only the project path. If the project folder is moved to another position or computer, you can change the project directory by changing the attribute Project.working_directory.

- Load an existing project and print information

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Name of the project
print(project.name)
# Path to the project folder
print(project.working_directory)
# Save project to be used later
project.save()
```

- Change the working directory. Copy/Move the project in the new path and then run:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/New_Project_Path/')
# Change the working directory. This might take a while
project.working_directory = r'/home/Work/New_Project_Path/'
# Save project to be used later
project.save()
```

Structures can be added to the project with the Project.add_structures module. This takes the path of a single structure or a folder containing different structures and copy them to the Input/Initial_Structures folder. Then, they are converted to Crystal objects to be used later. During this process structures are converted to the PDB file format with openbabel and the atomtype of each atom is detected with the AmberTools program "atomtype". Files that do not have a format extension compatible with openbabel are ignored.

- Add new crystal structures to the project:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Add all crystal structures in the given folder
project.add_structures(r'/home/Work/Structures/')
# Add the specified crystal structure
project.add_structures(r'/home/Work/Structures/str1.pdb')
# List of structures imported in the project
print(project.initial_crystals)
# Save project to be used later
```

```
project.save()
```

After importing the initial set of structures, we need to create a new Method object to run the simulations. Every method is characterised by the MD package used (only gromacs for now), the molecular forcefield and the list of simulations to be used. Method objects can be created with the Project.new_method function. In the beginning, you just need to specify a label for that method. This can be used to get back that method later on or delete it. Methods are stored in the Project.methods list.

- Create a new method and print its manual:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Creates a new method
gaff = project.new_method('GAFF')
# Print new method manual
gaff.help()
# Save project to be used later
project.save()
```

- Retrieve an existing method:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Get an existing method
gaff = project.get_method('GAFF')
# Print method MD package
print(gaff.package)
# Save project to be used later
project.save()
```

- Delete an existing method:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Delete an existing method
project.del_method('GAFF')
# Save project to be used later
project.save()
```

Once created, the Method object is empty and must be integrated with the topology that will be used in simulations. PyPol requires the molecular .itp file format and the coordinate file of the isolated molecule from which the forcefield is created. The topology file can be added but the only the [defaults] section will be imported as the [system] and [molecules] section depends on the molecular forcefield and the number of molecules in each crystal. The coordinate file of the isoleted

molecule is used to reindex every molecule and force them to have the same order of the one specified in the itp file. Molecular forcefield are imported in the Project with the module Project.Method.new_molecule. In this module, also the potential energy of the isolated molecule can be added and it will be used to calculate the lattice energy of each crystal. Finally, the Project.Method.generate_input module will replicate the unit cells to have supercells of the closest dimensions to the ones specified in the box parameter. If the parameter orthogonalize is true, the most orthogonal non-primitive cell is used instead of the unit cell. After this the data folder will contain all the supercells in the GRO and PDB file formats and the topology files with the correct parameters.

- Import Forcefield and generate structures:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Creates a new method
gaff = project.new_method('GAFF')
# Topology file to be used in simulations
path_top = '/home/Work/Forcefield/topol.top'
# Molecular forcefield file
path_itp = '/home/Work/Forcefield/MOL.itp'
# Isolated molecule file.
path_crd = '/home/Work/Forcefield/molecule.mol2'
# Copy relevant part of the topology to the project folder
gaff.new_topology(path_top)
# Copy molecular forcefield of molecule "MOL"
gaff.new_molecule(path_itp, path_crd, -100.0000)
# Generates the input files for the simulation
gaff.generate_input(box=(50., 50., 50.), orthogonalize=True)
# Save project to be used later
project.save()
```

2.2 Run Simulations and import results

PyPol has 4 different simulation objects that differ in the way inputs are created and results are read by the program. These can be created with the Method.new_simulation function by specifying the required Simulation Object:

- "em": Energy minimization using Gromacs. If no mdp file is specified, the default one is used.
- "cr": Cell relaxation using LAMMPS. If no input or forcefiled file are specified, a new topology is obtained converting the Gromacs one with InterMol.
- "md": Molecular Dynamics using Gromacs. If no mdp file is specified the default ones are used. Check the PyPol/-data/Defaults/Gromacs folder to see or modify them.

- "wtmd": Well-Tempered Metadynamics simulations (TODO)

In general, all the simulation objects have:

- help() module with specifics and examples
- generate_input() module to copy the mdp file to all the folders in the data directory and to generate a bash script to run all the simulations.
- get_results() module to verify the normal termination of the simulation and calculate the lattice or potential energy

The first one is the EnergyMinimization object can be used with all the Gromacs energy minimization algorithms. To use default mdp files use the names "em" and "relax". Alternatively, specify the path to the Gromacs mdp file to be used. Default mdp files are stored in the PyPol installation directory in PyPol/data/Defaults/Gromacs and can be modified if needed. If bash_script is set to True, ascript to run all simulations is created in:
<project_folder>/data/<method_name>/run_<simulation_name>.sh
After performing all simulations, the get_results module calculate the lattice energy of the crystal and change the crystal status from "incomplete" to "complete".

- Create a new EnergyMinimization object using default mdp files and print its manual:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Create a new simulation using defaults em.mdp file
em = gaff.new_simulation("em", simtype="em")
# Print new simulation manual
em.help()
# Save project to be used later
project.save()
```

- Alternatively, create a new EnergyMinimization object using a specified mdp file:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Gromacs mdp file
path_mdp = '/home/Work/InputFiles/em.mdp'
# Create a new simulation
em = gaff.new_simulation("em", "em", path_mdp)
```

```
# Save project to be used later
project.save()
```

- Generate inputs for all the crystals, including a bash script to run all simulations:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve an existing simulation
em = gaff.get_simulation("em")
# Copy the MDP file in each crystal directory
em.generate_input(bash_script=True)
# Save project to be used later
project.save()
```

- Check the normal termination of each simulation and get the energy landscape:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve an existing simulation
em = gaff.get_simulation("em")
# Check normal termination and import potential energy
em.get_results()
# Print the lattice energy of each crystal
for crystal in em.crystals:
    print(crystal.name, crystal.energy)
# Save project to be used later
project.save()
```

Since Gromacs does not allow for the simulation box to relax during the energy minimization, the optimised structure can be converted to the LAMMPS format for which this feature is available. The topology conversion for a single molecule is done using InterMol and then replicated for each molecule inside the crystal. As before, the get_results module is used to check the simulations and upload the results. A good approach is to optmize the initial atomic positions with Gromacs according to the imported topology, then relax the cell parameters using LAMMPS and finally rerun an energy minimization simulation with gromacs. The last step is done to avoid possible differences between the two MD packages.

- Create a new CellRelaxation object, using InterMol to generate LAMMPS topology:

```
from PyPol import pypol as pp
# Load project from the specified folder
```

```
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Create a new simulation
cr = gaff.new_simulation("cr", "cr")
# Save project to be used later
project.save()
```

MD simulations inputs are created by the MolecularDynamics object. Default mdp files can be used by typing the following names withoud specifying any mdp path:

- "nvt": 3ns simulation in the NVT ensemble
- "berendsen": 1ns simulation in the NPT ensemble using the Berendsen barostat
- "parrinello": 3ns simulation in the NPT ensemble using the Parrinello-Rahman barostat
- "md": 5ns simulation in the NPT ensemble using the Parrinello-Rahman barostat

- Create a new MolecularDynamics objects:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Gromacs mdp file
path_mdp = '/home/Work/InputFiles/nvt.mdp'
# Create a new simulation
nvt = gaff.new_simulation("nvt", "md", path_mdp)
# Save project to be used later
project.save()
```

2.3 Trajectory analysis

2.3.1 Distributions: Molecular orientation and identification of melted structures

Between an MD step and another, it is useful to check if some of the structures are melted. A good way to do it is to check the orientational disorder of the molecules inside the crystal. We can identify one vector that connects two atoms for each molecule and calculate the intermolecular angle between each of the pairs. The distribution of these angles should exhibit well defined peaks if the system is in the crystal form. Melted structures have molecules oriented randomly and a distribution trend given by 0.5*sin(x). By comparing this reference distribution with the structure one, melted structures are

easily identified. Molecular orientation distributions can also be used as part of the crystal fingerprints in the clustering analysis.

- Select atoms of the orientational vectors and create plumed inputs:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Retrieve the CV Object
mo = gaff.get_cv("mo")
# Use the first two atoms to define the orientational vect
mo.set_atoms((0, 1), molecule)
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
mo.generate_input(npt)
# Save project
project.save()
```

- Import distributions once the plumed driver analysis is finished:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Retrieve the CV Object
mo = gaff.get_cv("mo")
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
mo.get_results(npt, plot=False)
# Save project
project.save()
```

- Check orientational disorder:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
```

```
gaff = project.get_method('GAFF')
# Retrieve the CV Object
mo = gaff.get_cv("mo")
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Identify melted structures
mo.identify_orientational_disorder(npt)
# Save project
project.save()
```

2.3.2 Distributions: Radial Distribution Function

The RDF object generates plumed inputs to calculate the Radial Distribution Function of molecules in each crystal. The coordinates to use can be the geometrical center of the molecule or the center of mass. Hydrogen atoms can be ignored in this calculation. The resulting distribution can be used as part of the crystal fingerprint in the clustering analysis.

- Create a new Collective Variable object, modify some of his attributes and generate the plumed inputs:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Create the RDF object for the radial distribution func
rdf = gaff.new_cv("rdf-center", "rdf")
# Use the geometrical center instead of center of mass
rdf.center = "geometrical"
# Use all atoms to calculate the molecule geometric center
rdf.set_atoms("all", molecule)
# Retrieve a completed simulation
nvt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
rdf.generate_input(npt)
# Save project to be used later
project.save()
```

- Retrieve an existing CV and check if plumed drive analysis is completed for all crystals:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
```

```
gaff = project.get_method('GAFF')
# Retrieve the RDF object
rdf = gaff.get_cv("rdf-center")
# Retrieve a completed simulation
nvt = gaff.get_simulation("nvt")
# Check and import resulting distributions
rdf.get_results(nvt)
# Save project to be used later
project.save()
```

- Generate inputs for the RDF of molecules center of mass excluding hydrogen atoms:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve the CV Object
rdf = gaff.get_cv("rdf")
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Select the atoms in the molecule
rdf.set_atoms("non-hydrogen", molecule)
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
rdf.generate_input(npt)
# Save project
project.save()
```

- Import distributions once the plumed driver analysis is finished:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Retrieve the CV Object
rdf = gaff.get_cv("rdf")
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
rdf.get_results(npt, plot=False)
# Save project
```

```
project.save()
```

2.3.3 Distributions: Torsions

With Torsions object, plumed inputs to calculate the distribution of torsional angles can be created. The resulting distribution can be used as part of the crystal fingerprint in the clustering analysis.

- Select atoms of the torsional angles and create plumed inputs:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Retrieve the CV Object
tor = gaff.get_cv("tor")
# Use the first four atoms to define the torsional angle
tor.set_atoms((0, 1, 2, 3), molecule)
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
tor.generate_input(npt)
# Save project
project.save()
```

- Import distributions once the plumed driver analysis is finished:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule("MOL")
# Retrieve the CV Object
tor = gaff.get_cv("tor")
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generate plumed driver input for the selected simulation
tor.get_results(npt, plot=False)
# Save project
project.save()
```

2.3.4 Combine 1D distributions in ND distribution

Two or more 1D torsional or intermolecular angles can be combined together to create multidimentsional distributions.

- Create two Torsions CV and combine them in 2D distributions:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Use molecular forcefield info for the CV
molecule = gaff.get_molecule(0)
# Create a new Torsions object
tor1 = gaff.new_cv("tor1", "tor")
# Define the atoms for the torsional angle
tor1.set_atoms((7, 12, 3, 6), molecule)
# Create a new Torsions object
tor2 = gaff.new_cv("tor2", "tor")
# Define the atoms for the torsional angle
tor2.set_atoms((4, 8, 16, 5), molecule)
# Combine two CV
tor = gaff.combine_cv("2d-tor", (tor1, tor2))
# Retrieve a completed simulation
nvt = gaff.get_simulation("nvt")
# Generate plumed driver input for the selected simulation
tor.generate_input(nvt)
# Save project to be used later
project.save()
```

2.3.5 Groups

Sometimes it is useful to divide the set of structures in subset based on distribution similarities or specific attributes of the crystal. One good example is to calculate the distribution of relevant torsional angles in a molecule and then divide the set of structures based on the different conformation of the molecule. This can be done by comparing all pairs of distributions and cluster together the similar ones. Alternatively, you can select a set of boundaries in the CV space and classify the structures based on which areas of the space are non-zero. If a group object is included in the clustering analysis, structures are firstly divided into the calculated groups. Then, fingerprint comparison is done only between members of the same group.

- Retrieve an existing CV and create groups of crystals from it:

```
from PyPol import pypol as pp
```

```
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Retrieve the CV Object
tor = gaff.get_cv("tor")
# Import plumed output and check normal termination
tor.get_results(npt)
# Create the GGFD object
conf = gaff.ggfd("conf", tor)
# Use the similarity grouping method
conf.grouping_method = "similarity"
# Use the simps method to calculate the Hellinger distance
conf.integration_type = "simps"
# Generates groups
conf.run(tor)
project.save()
```

- Group structures by similarity:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve the CV Object
tor = gaff.get_cv("tor")
# Create the GGFD object
conf = gaff.ggfd("conf", tor)
# Use the similarity grouping method
conf.grouping_method = "similarity"
# Use the simps method to calculate the hellinger distance
conf.integration_type = "simps"
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generates groups
conf.run(tor)
project.save()
```

- Group structures using boundaries:

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
```

```
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrieve the CV Object (2D torsional angle)
tor = gaff.get_cv("tor")
# Create the GGFD object
conf = gaff.ggfd("conf", tor)
# Use the "groups" grouping method
conf.grouping_method = "groups"
# Cutoff for determining which areas are occupied
conf.group_threshold = 0.1
# Define the group boundaries in the (2D) CV space
conf.set_group_bins((-2., 0, 2.),(-1., 2.), periodic=True)
# Retrieve a completed simulation
npt = gaff.get_simulation("npt")
# Generates groups
conf.run(tor)
project.save()
```

2.3.6 Molecular and crystal attributes

User can add custom made features to Crystal and Molecule objects using the Crystal.set_attribute and Molecule.set_attribute. In the following example, the chirality of ibuprofen molecules is identified and saved as molecular attribute. Then, enantiopure and racemic structures are distinguished using this information. The crystal attribute can be then used by the GGFA (Generate Groups From Attributes) object to classify the crystals.

```
from PyPol import pypol as pp
from rdkit import Chem
project = pp.load_project(r'/home/nicolas/Work/Ibuprofen/project_tmp/')

for crystal in project.crystals:
    ChiralDict[crystal._name] = []
    molecules = crystal.molecules
    # Identify the R or S configuration of each molecule and assign the molecule attribute
    for molecule in molecules:
        molecule._save_gro(crystal.path + molecule.residue + ".gro")

        from openbabel import openbabel
        ob_conversion = openbabel.OBConversion()
        ob_conversion.SetInAndOutFormats("gro", "mol")
        mol = openbabel.OBMol()
        ob_conversion.ReadFile(mol, crystal.path + molecule.residue + ".gro")
        ob_conversion.WriteFile(mol, crystal.path + molecule.residue + ".mol")
```

```
        mol1 = Chem.MolFromMolFile(crystal.path + molecule.residue + ".mol")
        Chem.AssignAtomChiralTagsFromStructure(mol1)
        c = Chem.FindMolChiralCenters(mol1)[0][1]
        ChiralDict[crystal._name].append(c)
        molecule.set_attribute("Chirality", c)
    crystal.update_molecules(molecules)
    # Assign a crystal attribute to each crystal depending if it is enantioure or racemic
    if all(k == ChiralDict[crystal._name][0] for k in ChiralDict[crystal._name]):
        crystal.set_attribute("Chirality", "Eniantiopure")
    else:
        crystal.set_attribute("Chirality", "Racemic")

project.save()
```

2.3.7 Clustering analysis

After having calculated the different distributions and groups, you can perform a clustering analysis using the crystal fingerprints to identify those structures that convert to the same geometry. The resulting cluster can be seen in the Output directory together with the Distance matrix and the FSFDP outputs.

```
from PyPol import pypol as pp
# Load project from the specified folder
project = pp.load_project(r'/home/Work/Project/')
# Retrieve an existing method
gaff = project.get_method('GAFF')
# Retrive the different distributions and groups objects
rdf = mdvv.get_cv("RDF")
mo = mdvv.get_cv("mo")
chi = mdvv.get_cv("Chirality")
tor = mdvv.get_cv("Stor")
conf = mdvv.get_cv("SConformation")
# Create a clustring object using the distributions and group objects needed
cm = mdvv.new_clustering_parameters("Cluster1", (chi, conf, rdf, mo, tor))
# Modify the clustering algorithm parameters
cm.d_c_neighbors_fraction = 0.01
cm.sigma_cutoff = 0.15
# Run the clustering analysis
cm.run(meta)
```

Chapter 3

Documentation

3.1 pypol

```
def load_project(project_folder: str, use_backup=False):
    """
    Load an existing project. The Project object is saved in the project directory every time the
    command Project.save() is used.
    :param project_folder: project folder specified in the new_project function
    :param use_backup: Use the Project object for the previous save
    :return: Project object
    """


def new_project(path_working_directory: str, name="project", overwrite=False):
    """
    Create a new Project object and generate all the fundamental directories in the project folder.
    :param name: Name of the new project.
    :param path_working_directory: Path to the new project directory. A new directory will be created.
    :param overwrite: if True, delete the previous folder.
    :return: Project object
```

3.1.1 pypol.Project

```
class Project(object):
    """
    The project class that stores and manage all information and methods.

    Attributes:
```

```
- name: Name of the project
- working_directory: path to the project folder
- methods: list of methods used in the project
- initial_crystals: list of structures imported in the project
- path_progress: general output file of the project
- path_input: Folder that contains initial structures and methods input files
- path_output: Folder that contains results and outputs not present in the progress file
- path_data: Folder that contains all simulations data files

Methods:
- help(): print attributes and methods available with examples of how to use them
- new_project(overwrite=False): creates a new project in the working_directory path
- save(): Save current project
- change_working_directory(path, reset_program_paths=False): Change project working directory path
- add_structures(path_structures, gen_unit_cell=False): Add crystal structures to the project
- new_method(name, package="gromacs"): Create a new Method object
- get_method(method_name): Get a Method object stored in the project
- del_method(method_name): Delete a stored Method object and all the dat, Input and Output
  folders used for it
"""

@staticmethod
def help():
    """
    Print attributes and methods available with examples of how to use them
    """

def save(self):
    """
    Save project object to project folder.
    :return:
    """

def change_working_directory(self, path: str, reset_program_paths: bool=False):
    """
    Change the working directory and all the paths in the project.
    :param path: New Project Path
    :param reset_program_paths: If True, check packages paths
    :return:
    """

def add_structures(self, path_structures: str):
    """
    Add a new structure (if path_structures is a file) or a set of structures
```

```
        (if path_structures is a folder) in the project_folder/Input/Sets/Set_name directory.
        Structures are converted to the pdb files using openbabel. The GAFF atomtype of each atom
        is detected using the atomtype program from AmberTools. This is required for the
        reindexing of atoms once a molecular forcefield is uploaded but has no impact
        on the forcefield used for simulations. Finally, for each structure, a Crystal object is
        created and stored in the Project object. A list of stored structures can be printed by typing
        "print(<project_name>.initial_crystals)".

        :param path_structures: Path to a structure file or a folder containing the structures.
        """

    def new_method(self, name: str, package="gromacs", _import=False):
        """
        Add a new method to the project. Type <method_name>.help() for more information about
        how to use it.
        :param name: str, label of the method, it will be used to retrieve the method later on
        :param package: MD package used to perform simulations
        :param _import:
        :return: Method object
        """

    def get_method(self, method_name):
        """
        Find an existing method by its name.
        :param method_name: Label given to the Method object
        :return: Method object
        """

    def del_method(self, method_name):
        """
        Delete an existing method.
        :param method_name: Label given to the Method object
        """
```

3.2 crystal

3.2.1 crystal.Crystal

```
class Crystal(object):
    """
    This object stores the relevant information about the crystal, including the molecules
    and atoms in it.

    Attributes:
```

```
- name: ID of the crystal as saved in the project
- label: Alternative name to identify special structures in crystals set.
- index: Crystal index in the set
- path: Folder with the simulations
- box: 3x3 Matrix of the three box vectors a, b, and c with components ay = az = bz = 0
- cell_parameters: Lattice parameters, a, b, c, alpha, beta, gamma
- volume: Volume of the simulation box
- Z: Number of molecules in the cell
- nmoltypes: Types of different molecules in the crystal (For now only one is allowed)
- rank: Rank of the structure in the method crystal set.
- energy: Pot. energy divided by the number of atoms and rescaled by the energy
    of an isolated molecule.
- state: State of the crystal:
    - incomplete: the simulation is not finished
    - complete: the simulation is finished and ready for analysis
    - melted: The simulation is completed but the structure is melted
    - clusterID: Name of the cluster the structure belongs.
- cvs: Collective Variables calculated for this structure.

Methods:
- help(): print attributes and methods available
- set_attribute(att, val): Set a crystal attribute
- get_attribute(att): get the attribute value of the specified attribute label
"""

@staticmethod
def help():
    """
    Print attributes and methods available.
    """

def set_attribute(self, att, val):
    """
    Create a custom attribute for the Crystal.
    :param att: Attribute label
    :param val: Attribute value
    :return:
    """
def get_attribute(self, att):
    """
    Retrieve an existing attribute from a Molecule object
    :param att: Attribute label
    :return: str, attribute value
    """
```

3.2.2 crystal.Molecule

```
class Molecule(object):
    """
    This class stores relevant details about a molecule and the atoms in it.
    Parameters can be derived from a coordinate or a topology file depending on its purpose.

    Attributes:
    - residue: The molecule label as specified in the forcefield
    - index: Index of the molecule inside the crystal
    - atoms: List of Atoms objects
    - natoms: Number of atoms in the molecule
    - centroid: The geometrical center of the molecule
    - contact_matrix: A NxN matrix (with N=natoms) with 1 elements if atoms are bonded and 0 if not.

    Methods:
    - help(): print attributes and methods available
    - set_attribute(att, val): Set a molecule attribute
    - get_attribute(att): get the attribute value of the specified attribute label
    """

    @staticmethod
    def help():
        """
        Print attributes and methods available
        """

    def set_attribute(self, att, val):
        """
        Create a custom attribute for the molecule.
        :param att: Attribute label
        :param val: Attribute value
        :return:
        """

    def get_attribute(self, att):
        """
        Retrieve an existing attribute from a Molecule object
        :param att: Attribute label
        :return:
        """
```

3.2.3 crystal.Atom

```
class Atom(object):
    """
    The Atom Class which stores relevant info about each atom of the molecule.

    Attributes:
    - label: Atom label as in the original coordinate file or in the forcefield
        if index reassignation is performed.
    - index: Index of the atom inside the molecule.
    - ff_type: Atom type as written in the forcefield.
    - type: Atom type identified by the AmberTools program 'atomtype'.
    - coordinates: Coordinates of the atom.
    - element: Element of the atom.
    - bonds: Index of atoms bonded to it.
    - charge: Charge of the atom
    - mass: Mass of the atom.

    Methods:
    - help(): print attributes and methods available
    """

    @staticmethod
    def help():
        """
        Print attributes and methods available
        """
```

3.3 gromacs

3.3.1 gromacs.Method

```
class Method(_GroDef):
    """
    The Method object defines the forcefield and the simulations to be used in the analysis.
    Gromacs is used for MD simulations.

    Attributes:
    - name: Name used to specify the object and print outputs
    - package: The package used for MD.
    - topology: Path to the Gromacs topology file .top.
    - nmolecules: Number of molecules (or .itp files) in the topology
    - crystals: list of Crystal objects contained in the method. This refers to the crystals
        prior to any simulation.
```

```
- gromacs: Gromacs command line
- mdrun_options: Options to be added to Gromacs mdrun command. For example '-v',
    '-v -nt 1', '-plumed plumed.dat'.
- atomtype: 'atomtype' command line
- intermol: Path of the 'convert.py' InterMol program
- lammps: LAMMPS command line
- pypol_directory: PyPol directory with defaults inputs
- path_data: data folder in which simulations are performed
- path_output: Output folder in which results are written
- path_input: Input folder to store inputs

Methods:
- help(): print attributes and methods available
- new_topology(path_top): Save the [ defaults ] section of a .top file and imports other
    than .itp files.
- new_molecule(path_itp, path_crd, potential_energy=0.0): Import forcefield parameters from
    .itp file.
    Parameters:
    - path_itp: Path of the .itp file
    - path_crd: Path of the coordinate file used to generate the forcefield. The conformation of
        the isolated molecule is not relevant but the atom order MUST be the same one of
        the forcefield.
    - potential_energy: Potential energy of an isolated molecule. This is used to calculate the lattice
        energy.
- get_molecule(name): return the molecule object with the specified name.
- generate_input(self, box=(40., 40., 40.), orthogonalize=True): Generate the initial
    coordinate and the topology files.
    Parameters:
    - box: Supercells are generated so that their box vectors are close in module to the ones specified.
        Distance values are in Angstrom.
    - orthogonalize: If True, cells parameters are modified to have a nearly orthogonal cell using the
        box parameter as limits in the search.
- new_simulation(name, simtype, path_mdp=None, path_lmp_in=None, path_lmp_ff=None, overwrite=False):
    Creates a new simulation object of the specified type:
        - "em":   Energy minimization using Gromacs. If no mdp file is specified,
                  the default one is used.
        - "cr":   Cell relaxation using LAMMPS. If no input or forcefiled file are specified,
                  a new topology is obtained converting the Gromacs one with InterMol.
        - "md":   Molecular Dynamics using Gromacs. If no mdp file is specified the default
                  ones are used. Check the PyPol/data/Defaults/Gromacs folder to see or modify them.
        - "wtmd": Well-Tempered Metadynamics simulations (TODO)
    If no path_mdp, path_lmp_in, path_lmp_ff are given, default input files will be used.
    Use the <simulation>.help() method to obtain details on how to use it.
- get_simulation(name): return the simulation object with the specified name.
```

```
- del_simulation(name): Remove simulation from project and the related folders.
- new_cv(name, cv_type): Generates a new cv of the specified name and type:
            Available types:
                - "tor":     Torsional angle.
                - "mo":      Intermolecular torsional angle.
                - "rdf":     Radial Distribution Function.
                - "density": Density
                - "energy":  Potential Energy
    Use the <cv>.help() method to obtain details on how to use it.
- combine_cvs(name, cvs): Torsions ("tor") and MolecularOrientation ("mo") objects are
    combined in ND distributions.
- ggfd(name, cv): Sort crystals in groups according to their similarity in the distribution used
    or to predefined group boundaries.
- get_cv(name): return the CV parameters object with the specified name.
- del_cv(name): delete the CV parameters object with the specified name.
- new_clustering_parameters(name, cvs): Creates a new clustering parameters object.
    CVs are divided in group and distribution types. Initially, crystals are sorted according
    to the group they belong. A distance matrix is generated and a clustering analysis using
    the FSFDP algorithm is then performed in each group. Use the <clustering>.help() method
    to obtain details on how to use it.
- get_clustering_parameters(name): return the clustering parameters object with the specified name.
- del_clustering_parameters(name): delete the clustering parameters object with the specified name.
"""

@staticmethod
def help():
    """
    Print attributes and methods available
    """

def new_molecule(self, path_itp: str, path_crd: str, potential_energy=0.0):
    """
    Define the molecular forcefield. The coordinate file used to generate the force field is necessary to
    identify atom properties, index order and bonds. If it is not a .mol2 file, it is converted to it with
    openbabel. \n
    :param path_itp: Path to the .itp file containing the molecular forcefield.
    :param path_crd: Path of the coordinate file used to generate the forcefield.
    :param potential_energy: Potential energy of an isolated molecule used to calculate the Lattice energy of
    Crystals
    """

def get_molecule(self, mol_name):
    """
    Retrieve the Molecule object imported from the molecular forcefield (.itp file)
```

```
    :param mol_name: label of the molecule
    :return: Molecule object
    """

def del_molecule(self, mol_name):

    """
    Delete the Molecule object with the specified label.
    :param mol_name: label of the molecule
    :return:
    """

def new_topology(self, path_top: str):
    """
    Add the topology file to the project. Only the [ defaults ] section is included. [ system ] and [ molecules ]
    section will be added to the topology file of each crystal.
    :param path_top:
    :return:
    """

def del_topology(self):
    """
    Delete the topology file used for simulations.
    """

def generate_input(self, box=(4., 4., 4.), orthogonalize=True):
    """
    Generate the coordinate and the topology files to be used for energy minimization simulations.

    :param box: Target length in the three direction. The number of replicas depends on the Crystal box parameters.
    :param orthogonalize: Find the most orthogonal, non-primitive cell
    :return:
    """

def new_simulation(self, name: str, simtype: str, path_mdp=None, path_lmp_in=None, path_lmp_ff=None,
                   crystals="all", catt=None):
    """
    Creates a new simulation object of the specified type:
        - "em":   Energy minimization using Gromacs. Use name="em" or name="relax" without specifying the path_mdp
                  variable to use the defaul mdp file.
        - "cr":   Cell relaxation using LAMMPS. If no input or forcefiled file are specified, a new topology
                  is obtained converting the Gromacs one with InterMol.
        - "md":   Molecular Dynamics using Gromacs. Use names "nvt", "berendsen", "parrinello", "md" without
                  specifying path_mdp to use the default ones.
```

```
                Check the PyPol/data/Defaults/Gromacs folder to see or modify them.
        - "wtmd": Well-Tempered Metadynamics simulations
    If no path_mdp, path_lmp_in, path_lmp_ff are given, default input files will be used.
    Use the <simulation>.help() method to obtain details on how to use it.
    :param name: Label of the new simulation object.
    :param simtype: Specify which simulation object to use.
    :param path_mdp: Path to the gromacs mdp file to be used in the
    :param path_lmp_in: Path to the LAMMPS input file
    :param path_lmp_ff: Path to the LAMMPS topology file
    :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                    "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                     a specific subset of crystals by listing crystal names.
    :param catt: (dict) Specify The custom attributes the crystal must have in to be added to the next simulation.
                 It must be in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}
    :return: Simulation object (EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics)
    """

def get_simulation(self, simulation_name: str):
    """
    Find an existing simulation by its name.\n
    :param simulation_name: Name assigned to the simulation
    :return: Simulation object
    """

def del_simulation(self, simulation_name: str):
    """
    Delete an existing simulation by its name.\n
    :param simulation_name: Name assigned to the simulation
    :return:
    """

def new_cv(self, name, cv_type):
    """
    Add a new Distribution Object or Collective Variable Object to the CV's list.
    Available Distribution types:
                - "tor":     Torsional angle.
                - "mo":      Intermolecular torsional angle.
                - "rdf":     Radial Distribution Function.
    Available Collective Variables:
                - "density": Density of the crystal.
                - "energy":  Potential Energy of the crystal.
    Use the <cv>.help() method to obtain details on how to use it.
    You can also add the AvoidScrewedBox object to the collective variables in order to avoid too tilted boxes.
    This is done by typing:
```

```
    asb= <method_name>.new_cv("asb", cv_type="asb")
    asb.generate_input(<simulation_name>)

    :param name: Object label
    :param cv_type: Specify which CV object to use
    :return:
    """

def combine_cvs(self, name, cvs: Union[tuple, list]):
    """
    Multiple 1-D Torsions ("tor") or MolecularOrientation ("mo") objects are combined in ND distributions.
    :param name: Object label
    :param cvs: Tuple or list of 1-D Distributions (all from the same class) to combine in ND distribution.
    :return: Combine object
    """

def ggfd(self, name, cv):
    """
    Generate Groups from Distributions. Sort crystals in groups according to their similarity in the distribution
    used or to predefined group boundaries. Use the GGFD.help() module for more detailed information.
    :param name: object label
    :param cv: Distribution to be used to define the different groups
    :return: GGFD object
    """

def ggfa(self, name, attribute):
    """
    Generate Groups from Attributes. Sort crystals in groups according to their attributes.
    Use the GGFA.help() module for more detailed information.
    :param name: object label
    :param attribute: dict with the attribute to use for classification
    :return: GGFA object
    """

def get_cv(self, cv_name: str):
    """
    Find an existing CV by its name.
    :param cv_name: Name assigned to the collective variable
    :return: Distribution or CollectiveVariable object
    """

def del_cv(self, cv_name: str):
    """
    Delete an existing Distribution or CollectiveVariable object.
```

```
        :param cv_name: object label
        :return:
        """

    def new_clustering_parameters(self, name: str, cvs: Union[list, tuple]):
        """
        Creates a new clustering parameters object. CVs are divided in group and distribution types.
        Initially, crystals are sorted according to the group they belong. A distance matrix is generated and a
        clustering analysis using the FSFDP algorithm is then performed in each group.
        Use the <clustering>.help() method to obtain details on how to use it.
        :param name: object label
        :param cvs: list or tuple of Distribution/Group object to be used for the clustering
        :return: Cluster object
        """

    def get_clustering_parameters(self, clustering_parameter_name: str):
        """
        Find an existing clustering parameters by its name.\n
        :param clustering_parameter_name: Name assigned to the clustering method
        :return: Cluster object
        """

    def del_clustering_parameters(self, clustering_method: str):
        """
        Delete an existing clustering parameters by.\n
        :param clustering_method: Name assigned to the clustering method
        :return:
        """
```

3.3.2 gromacs.EnergyMinimization

```
class EnergyMinimization(_GroSim):
    """
    Perform Energy Minimization simulations using Gromacs.

    Attributes:
    - name: Name used to specify the object and print outputs
    - gromacs: Gromacs command line
    - mdrun_options: Options to be added to Gromacs mdrun command. For example '-v', '-v -nt 1', '-plumed plumed.dat'.
    - atomtype: 'atomtype' command line
    - pypol_directory: PyPol directory with defaults inputs
    - path_data: data folder in which simulations are performed
    - path_output: Output folder in which results are written
    - path_input: Input folder to store inputs
```

```
    - intermol: Path of the 'convert.py' InterMol program
    - lammps: LAMMPS command line
    - type: "Energy Minimization"
    - crystals: List of crystals on which simulation will be performed
    - sim_index: index of the simulation, define the position of the simulation in the workflow
    - mdp: Molecular Dynamics Parameters file path
    - completed: True if all crystall are not in the "incomplete" state
    - global_minima: Global minima of the set, available if the simulation is completed
    - hide: show or not the the relative potential energy file in the output file

    Methods:
    - help(): Print attributes and methods
    - generate_input(bash_script=False, crystals="all"): copy the .mdp file in each crystal folder.
    - get_results(crystals="all"): check if simulations ended or a rerun is necessary.
    """

    @staticmethod
    def help():
        """
        Print attributes and methods
        """

    def generate_input(self, bash_script=False, crystals="all"):
        """
        Copy the Gromacs .mdp file to each crystal path.
        :param bash_script: If bash_script=True, a bash script is generated to run all simulations
        :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter
        :return:
        """

    def get_results(self, crystals="all"):
        """
        Verify if the simulation ended correctly and upload new crystal properties.
        :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter.
                 Alternatively, you can use:
                 - "all": Select all non-melted structures
                 - "incomplete": Select crystals whose simulation normal ending has not been detected before.
        :return:
        """
```

3.3.3 gromacs.CellRelaxation

```
class CellRelaxation(_GroSim):
```

```
"""
Perform Energy Minimization and Cell Relaxation simulations with LAMMPS. This is used to optimize the simulation
box parameters, feature not available in Gromacs. If no input is given, it convert Gromacs input files to the
LAMMPS ones with InterMol.

Attributes:
- name: Name used to specify the object and print outputs
- gromacs: Gromacs command line
- mdrun_options: Options to be added to Gromacs mdrun command. For example '-v', '-v -nt 1', '-plumed plumed.dat'.
- atomtype: 'atomtype' command line
- pypol_directory: PyPol directory with defaults inputs
- path_data: data folder in which simulations are performed
- path_output: Output folder in which results are written
- path_input: Input folder to store inputs
- intermol: Path of the 'convert.py' InterMol program
- lammps: LAMMPS command line
- type: "Cell Relaxation"
- crystals: List of crystals on which simulation will be performed
- sim_index: index of the simulation, define the position of the simulation in the workflow
- mdp: Molecular Dynamics Parameters file path
- completed: True if all crystall are not in the "incomplete" state
- global_minima: Global minima of the set, available if the simulation is completed
- hide: show or not the the relative potential energy file in the output file
- path_lmp_in: Input file for LAMMPS
- path_lmp_ff: LAMMPS Topology file for molecule

Methods:
- generate_input(bash_script=False, crystals="all"): copy the .mdp file in each crystal folder.
- get_results(crystals="all"): check if simulations ended or a rerun is necessary.
"""

@staticmethod
def help():
    """
    Print attributes and methods
    """

def generate_input(self, bash_script=False, crystals="all"):
    """
    Generate LAMMPS inputs. If no topology is given, a LAMMPS topology is generated from the gromacs one
    using Intermol. The latter is applied to a single molecule and then replicated for each molecule of the crystal.

    :param bash_script: If bash_script=True, a bash script is generated to run all simulations
    :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter
```

```
        :return:
        """

    def get_results(self, crystals="all"):
        """
        Verify if the simulation ended correctly and upload new crystal properties.
        Convert files back to the Gromacs file format.
        :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter.
                         Alternatively, you can use:
                         - "all": Select all non-melted structures
                         - "incomplete": Select crystals whose simulation normal ending has not been detected before.
        :return:
        """
```

3.3.4 gromacs.MolecularDynamics

```
class MolecularDynamics(_GroSim):
    """
    Perform MD simulations using Gromacs.

    Attributes:
    - name: Name used to specify the object and print outputs
    - gromacs: Gromacs command line
    - mdrun_options: Options to be added to Gromacs mdrun command. For example '-v', '-v -nt 1', '-plumed plumed.dat'.
    - atomtype: 'atomtype' command line
    - pypol_directory: PyPol directory with defaults inputs
    - path_data: data folder in which simulations are performed
    - path_output: Output folder in which results are written
    - path_input: Input folder to store inputs
    - intermol: Path of the 'convert.py' InterMol program
    - lammps: LAMMPS command line
    - type: "Energy Minimization"
    - crystals: List of crystals on which simulation will be performed
    - sim_index: index of the simulation, define the position of the simulation in the workflow
    - mdp: Molecular Dynamics Parameters file path
    - completed: True if all crystall are not in the "incomplete" state
    - global_minima: Global minima of the set, available if the simulation is completed
    - hide: show or not the the relative potential energy file in the output file

    Methods:
    - help(): Print attributes and methods
    - generate_input(bash_script=False, crystals="all"): copy the .mdp file in each crystal folder.
```

```
    - get_results(crystals="all"): check if simulations ended or a rerun is necessary.
    """

    @staticmethod
    def help():
        """
        Print attributes and methods
        """

    def generate_input(self, bash_script=False, crystals="all", catt=None):
        """
        Copy the Gromacs .mdp file to each crystal path.
        :param bash_script: If bash_script=True, a bash script is generated to run all simulations
        :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter
        :param catt: Use crystal attributes to select the crystal list
        """

    def get_results(self, crystals="all", timeinterval=200):
        """
        Verify if the simulation ended correctly and upload new crystal properties.
        :param crystals: You can select a specific subset of crystals by listing crystal names in the crystal parameter.
                         Alternatively, you can use:
                         - "all": Select all non-melted structures
                         - "incomplete": Select crystals whose simulation normal ending has not been detected before.
        :param timeinterval: int, Time interval in ps to use for calculating different properties averages.
                             For example, a timeinterval of 500 means that the last 500 ps of the simulation are used
                             to calculate the average potential energy of the system.
        """
```

3.4 analysis

3.4.1 analysis.Torsions

```
class Torsions(_CollectiveVariable):
    """
    Generates a distribution of the torsional angles of the selected atoms.
    Attributes:\n
    - name: name of the CV.
    - type: Type of the CV.
    - plumed: Command line for plumed.
    - clustering_type: How is it treated by clustering algorithms.
    - kernel: kernel function to use in the histogram generation.
    - bandwidth: the bandwidths for kernel density estimation.
```

```
- grid_min: the lower bounds for the grid.
- grid_max: the upper bounds for the grid.
- grid_bins: the number of bins for the grid.
- grid_space: the approximate grid spacing for the grid.
- timeinterval: Simulation time interval to generate the distribution.
- atoms: the 4 atom index of the molecular forcefield object used to generate the set of torsional angles
- molecule: the molecular forcefield object from which atoms are selected

Methods:\n
- help(): Print attributes and methods
- set_atoms(atoms, molecule): Select the 4 atom index from the Molecule obj to generate the set of torsional angles
- generate_input(simulation, bash_script=True): Generate the plumed input files
- get_results(simulation, crystal='all', plot=True): Check if the plumed driver analysis is ended and store results
"""

@staticmethod
def help():
    """
    Print attributes and methods.
    """

def set_atoms(self, atoms: Union[list, tuple], molecule: Molecule):
    """
    Select atom indices of the reference molecule. This is used to identify the torsions of each molecule in the
    crystal.
    :param atoms: list, Atom indexes. All atoms indices are available in the project output file after the topology
    is defined.
    :param molecule: obj, Reference molecule
    :return:
    """

def generate_input(self,
                   simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                   bash_script=True,
                   crystals="all",
                   catt=None,
                   matt=None):
    """
    Generate the plumed input files. If the catt option is used, only crystals with the specified attribute are
    used. If the matt option is used only molecules with the specified attributes are used. In both cases,
    attributes must be specified in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}.

    :param simulation: Simulation object
    :param bash_script: If True, generate a bash script to run simulations
```

```
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :param matt: Use Molecular attributes to select the molecules list
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """

    def get_results(self,
                    simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                    crystals: Union[str, list, tuple] = "all",
                    plot: bool = True, catt=None):
        """
        Verify if the distribution has been correctly generated and store the result. If the distribution is taken over
        different frames, the average is calculated.
        :param simulation: Simulation object
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :param plot: If true, generate a plot of the distribution.
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```

3.4.2 analysis.MolecularOrientation

```
class MolecularOrientation(_CollectiveVariable):
    """
    Generates a distribution of the intermolecular torsional angles of the selected atoms.
    Attributes:
    - name: name of the CV.
    - type: Type of the CV.
    - plumed: Command line for plumed.
    - clustering_type: How is it treated by clustering algorithms.
    - kernel: kernel function to use in the histogram generation.
    - bandwidth: the bandwidths for kernel density estimation.
    - grid_min: the lower bounds for the grid.
    - grid_max: the upper bounds for the grid.
    - grid_bins: the number of bins for the grid.
    - grid_space: the approximate grid spacing for the grid.
    - timeinterval: Simulation time interval to generate the distribution.
    - atoms: the 2 atom index of the molecular forcefield object used to generate the set of orientational vectors
    - molecules: the molecular forcefield object from which atoms are selected
```

```
Methods:
- help(): Print attributes and methods
- set_atoms(atoms, molecule): Select the 2 atom index from the Molecule obj to generate the set of orientational vectors
- generate_input(simulation, bash_script=True): Generate the plumed input files
- get_results(simulation, crystal='all', plot=True): Check if the plumed driver analysis is ended and store results
- identify_orientational_disorder(simulation, crystals = "all", cutoff = 0.1, catt=None): Identify melted
    structures.
"""

@staticmethod
def help():
    """
    Print attributes and methods.
    """

def set_atoms(self, atoms: Union[list, tuple], molecule: Molecule):
    """
    Select the 2 atom index from the Molecule obj to generate the set of orientational vectors
    :param atoms: list, Atom indexes. All atoms indices are available in the project output file after the topology
    is defined.
    :param molecule: obj, Reference molecule
    :return:
    """

def generate_input(self, simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                   bash_script=True, crystals="all", catt=None, matt=None):
    """
    Generate the plumed input files. If the catt option is used, only crystals with the specified attribute are
    used. If the matt option is used only molecules with the specified attributes are used. In both cases,
    attributes must be specified in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}.

    :param matt: Use Molecular attributes to select the molecules list
    :param catt: Use crystal attributes to select the crystal list
    :param simulation: Simulation object
    :param bash_script: If True, generate a bash script to run simulations
    :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                     "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                     a specific subset of crystals by listing crystal names.
    :return:
    """

def get_results(self,
                simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                crystals: Union[str, list, tuple] = "all",
```

```
                plot: bool = True, catt=None):
        """
        Verify if the distribution has been correctly generated and store the result. If the distribution is taken over
        different frames, the average is calculated.
        :param simulation: Simulation object
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :param plot: If true, generate a plot of the distribution.
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """

    def identify_orientational_disorder(self,
                                        simulation: Union[EnergyMinimization, CellRelaxation,
                                                          MolecularDynamics, Metadynamics],
                                        crystals: Union[str, list, tuple] = "all",
                                        cutoff: float = 0.1, catt=None):
        """
        Given the intermolecular angle distribution obtained for each crystal in a simulation, it compares
        it with an homogeneous distribution (typical of melted systems) to identify possible orientational disorder. If the
        distance between a crystal MO distribution and the reference one is less than the specified cutoff, the structure is
        considered melted and will not be used in the next simulation.
        :param simulation: Simulation object
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :param cutoff: Distance cutoff from melted to be used for identifying melted structures
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```

3.4.3 analysis.Combine

```
class Combine(object):
    """
    Combine torsional or intermolecular torsional angles in multidimensional distributions.

    Attributes:\n
    - name: name of the CV.
    - type: Type of the CV.
    - clustering_type: How is it treated by clustering algorithms.
    - kernel: kernel function to use in the histogram generation.
    - timeinterval: Simulation time interval to generate the distribution.
```

```
- cvs: List CVs names

Methods:\n
- help(): Print attributes and methods
- generate_input(simulation, bash_script=True): Generate the plumed input files
- get_results(simulation, crystal='all', plot=True): Check if the plumed driver analysis is ended and store results
"""

@staticmethod
def help():
    """
    Print attributes and methods
    """

def generate_input(self,
                   simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                   bash_script: bool = True,
                   crystals="all",
                   catt=None, matt=None):
    """
    Generate the plumed input files. If the catt option is used, only crystals with the specified attribute are
    used. If the matt option is used only molecules with the specified attributes are used. In both cases,
    attributes must be specified in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}.

    :param simulation: Simulation object
    :param bash_script: If True, generate a bash script to run simulations
    :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                     "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                     a specific subset of crystals by listing crystal names.
    :param matt: Use Molecular attributes to select the molecules list
    :param catt: Use crystal attributes to select the crystal list
    :return:
    """


def get_results(self,
                simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                crystals: Union[str, list, tuple] = "all",
                plot: bool = True,
                catt=None):
    """
    Verify if the distribution has been correctly generated and store the result. If the distribution is taken over
    different frames, the average is calculated.
    :param simulation: Simulation object
```

```
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :param plot: If true, generate a plot of the distribution. This is not available for 3D distributions or higher.
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```

3.4.4 analysis.RDF

```
class RDF(_CollectiveVariable):
    """
    Generates a distribution of the intermolecular torsional angles of the selected atoms. NB: The grid_max variable is given
    by half of the smallest diagonal element of the box matrix and is not specified.
    Attributes:\n
    - name: name of the CV.
    - type: Type of the CV.
    - plumed: Command line for plumed.
    - clustering_type: How is it treated by clustering algorithms.
    - kernel: kernel function to use in the histogram generation.
    - bandwidth: the bandwidths for kernel density estimation.
    - grid_min: the lower bounds for the grid.
    - grid_bins: the number of bins for the grid.
    - grid_space: the approximate grid spacing for the grid.
    - timeinterval: Simulation time interval to generate the distribution.
    - atoms: the atoms index of the molecular forcefield object used to calculate molecule position
    - molecules: the molecular forcefield object from which atoms are selected
    - center: Calculate the molecule position based on geometrical center or center of mass
    - r_0: R_0 parameter

    Methods:\n
    - help(): Print attributes and methods
    - set_atoms(atoms, molecule): Select the atoms from the Molecule obj to generate calculate the molecule position
    - generate_input(simulation, bash_script=True): Generate the plumed input files
    - get_results(simulation, crystal='all', plot=True): Check if the plumed driver analysis is ended and store results
    """

    @staticmethod
    def help():
        """
        Print attributes and methods
        """
```

```
def set_atoms(self, atoms: str, molecule: Molecule, overwrite: bool = True):
    """
    Select the atoms from the Molecule obj to generate calculate the molecule position.
    :param atoms: str. You can use atoms="all" to select all atoms or atoms="non-hydrogen"
                  to select all atoms except H.
    :param molecule: Molecular forcfield Molecule object
    :param overwrite: If True, ignores previous atom settings
    :return:
    """

def generate_input(self,
                   simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                   bash_script: bool = True,
                   crystals="all",
                   catt=None,
                   matt=None):
    """
    Generate the plumed input files. If the catt option is used, only crystals with the specified attribute are
    used. If the matt option is used only molecules with the specified attributes are used. In both cases,
    attributes must be specified in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}.

    :param matt: Use Molecular attributes to select the molecules list
    :param catt: Use crystal attributes to select the crystal list
    :param bash_script: If True, generate a bash script to run simulations
    :param simulation: Simulation object
    :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                     "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                     a specific subset of crystals by listing crystal names.
    :return:
    """

def get_results(self,
                simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
                crystals: Union[str, list, tuple] = "all",
                plot: bool = True, catt=None):
    """
    Verify if the distribution has been correctly generated and store the result. If the distribution is taken over
    different frames, the average is calculated.
    :param simulation: Simulation object
    :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                     "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                     a specific subset of crystals by listing crystal names.
    :param plot: If true, generate a plot of the distribution.
```

```
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```

3.4.5 analysis.Wall

```
class Wall(object):
    """
    Creates a plumed input file for upper and lower lower WALLS
    Attributes:\n
    - name: label for the WALL.
    - type: Wall.
    - arg: Inputs for the wall
    - position: Position of the wall
    - kappa: Force constant of the wall
    - offset: The offset for the start of the wall
    - exp: powers for the wall
    - eps: the rescaling factor
    - stride: If possible, the stride used for printing the bias potential in the output file
    - collective_variable_line: String to be added above the Wall command. This can be used to specify the inputs for
      the wall (ARG)

    Methods:\n
    - add_arg(name, kappa=100000, offset=0.0, exp=2, eps=1, at=0.): Add a new input for the wall
    - reset_arg(name, kappa=100000, offset=0.0, exp=2, eps=1, at=0.): Modify an existing input for the wall
    """

    def add_arg(self, name, kappa=100000, offset=0.0, exp=2, eps=1, at=0.):
        """
        Add an argument to the Wall object.
        :param name: Inputs (arg) for the wall
        :param at: Position of the wall
        :param kappa: Force constant of the wall
        :param offset: The offset for the start of the wall
        :param exp: powers for the wall
        :param eps: the rescaling factor
        :return:
        """

    def reset_arg(self, name, kappa=100000, offset=0.0, exp=2, eps=1, at=0.):
        """
        Modify an existing argument of the Wall object with the default ones, unless they are specified.
        :param name: Inputs (arg) for the wall
```

```
        :param at: Position of the wall
        :param kappa: Force constant of the wall
        :param offset: The offset for the start of the wall
        :param exp: powers for the wall
        :param eps: the rescaling factor
        :return:
        """


class AvoidScrewedBox(Wall):
    """
    Creates a plumed input file forcing the non-diagonal element of the box matrix to stay within a certain range:
    abs(bx) <= 0.5*ax
    abs(cx) <= 0.5*ax
    abs(cy) <= 0.5*by
    This is done by creating three upper walls with the following attributes:
    Attributes:\n
    - name: label for the WALL.
    - type: "Avoid Screwed Box (Wall)"
    - arg: [bx,cx,cy]
    - position: [0.,0.,0.]
    - kappa: [100000,100000,100000]
    - offset: [0.1, 0.1, 0.1]
    - exp: [2,2,2]
    - eps: [1,1,1]
    - stride: If possible, the stride used for printing the bias potential in the output file
    - collective_variable_line: "
    cell: CELL
    bx: MATHEVAL ARG=cell.bx,cell.ax FUNC=abs(x)-0.5*y PERIODIC=NO
    cx: MATHEVAL ARG=cell.cx,cell.ax FUNC=abs(x)-0.5*y PERIODIC=NO
    cy: MATHEVAL ARG=cell.cy,cell.by FUNC=abs(x)-0.5*y PERIODIC=NO"

    Methods:\n
    - generate_input(simulation: MolecularDynamics,crystals="all", catt=None): Generate plumed input
    """

    def generate_input(self, simulation: MolecularDynamics,
                       crystals="all",
                       catt=None):

        """
        Generate the plumed input files. This is particularly useful for crystals with tilted boxes.
        If the catt option is used, only crystals with the specified attribute are used.
        Attributes must be specified in the form of a python dict, menaning catt={"AttributeLabel": "AttributeValue"}.
```

```
        NB: The <simulation>.mdrun_options attribute is modified to include "-plumed plumed_<name>.dat"
        :param catt: Use crystal attributes to select the crystal list
        :param simulation: Simulation object
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
                         "centers", use only cluster centers from the previous simulation. Alternatively, you can select
                         a specific subset of crystals by listing crystal names.
        :return:
        """
```

3.4.6 analysis.Density

```
class Density(_MetaCV):
    """
    Use the density of the crystal as a collective variable
    Attributes:\n
    - name: name of the CV.
    - cv_type: Type of the CV.
    - sigma: the bandwidth for kernel density estimation.
    - grid_min: the lower bounds for the grid.
    - grid_max: the upper bounds for the grid.
    - grid_bins: the number of bins for the grid.
    - grid_space: the approximate grid spacing for the grid.
    - use_walls: Use walls at the upper and lower bounds of the grid to force the system not to escape from it
    - walls: return a list with the upper and lower walls
    - uwall: return the upper wall
    - lwall: return the lower wall
    """
```

3.4.7 analysis.PotentialEnergy

```
class PotentialEnergy(_MetaCV):
    """
    Use the Potential Energy of the crystal as a collective variable
    Attributes:\n
    - name: name of the CV.
    - cv_type: Type of the CV.
    - sigma: the bandwidth for kernel density estimation.
    - grid_min: the lower bounds for the grid.
    - grid_max: the upper bounds for the grid.
    - grid_bins: the number of bins for the grid.
    - grid_space: the approximate grid spacing for the grid.
    """
```

3.4.8 analysis.GGFD

```
class GGFD(_GG):
    """
    Classify structures based on their structural fingerprint.
    Attributes:\n
    - name: name of the CV.
    - type: Type of the CV.
    - clustering_type: How is it treated by clustering algorithms.
    - grouping_method: Classification method. It can be based on the distribution similarity or specific patterns.
    - integration_type: For similarity methods, you can select how the hellinger distance is calculate.
    - group_threshold: For groups method, the tolerance to define the belonging to a group

    Methods:\n
    - help(): returns available attributes and methods.
    - set_group_bins(*args, periodic=True, threshold="auto"): select boundaries for the groups grouping method.
    - run(simulation): Creates groups from the crystal distributions in the simulation object.
    """

    @staticmethod
    def help():
        """
        Print attributes and methods
        """

    def set_group_bins(self, *args: Union[list, tuple], periodic: bool = True):  # , threshold="auto"
        # TODO Change args to dict with key == names of the variables.
        #  As it is right now you have to remember the order in which ND dimensional distribution are added together.
        """
        Select boundaries for the groups grouping method. args must be iterable objects.
        If periodic = False, an additional boundary is put at the grid max and min.
        :param args: A list or a tuple of the dividing boundaries in a distribution. The number of boundaries must be
               equal to the dimension of the distribution.
        :param periodic: If True, periodic conditions are applied to the grouping algorithm. Mixing periodic and
               non-periodic boundaries can be done by setting periodic=True and adding a 0. to the boundary list in
               which it is not true
        :return:
        """

    def run(self,
            simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
            crystals="all", catt=None):
        """
        Creates groups from the crystal distributions in the simulation object.
        :param simulation: Simulation Object (EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics)
```

```
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
               "centers", use only cluster centers from the previous simulation. Alternatively, you can select
               a specific subset of crystals by listing crystal names.
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```

3.4.9 analysis.GGFA

```
class GGFA(_GG):
    """
    Classify structures based on their attributes.
    Attributes:\n
    - name: name of the CV.
    - type: Type of the CV.
    - clustering_type: "classification"
    - attribute: Attribute label to be used for classification

    Methods:\n
    - help(): returns available attributes and methods. (TODO)
    - run(simulation): Creates groups looking at the crystal attributes in the simulation object
    """

    def run(self,
            simulation: Union[EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics],
            crystals="all", catt=None):
        """
        Creates groups from the crystal attributes in the simulation object.
        :param simulation: Simulation Object (EnergyMinimization, CellRelaxation, MolecularDynamics, Metadynamics)
        :param crystals: It can be either "all", use all non-melted Crystal objects from the previous simulation or
               "centers", use only cluster centers from the previous simulation. Alternatively, you can select
               a specific subset of crystals by listing crystal names.
        :param catt: Use crystal attributes to select the crystal list
        :return:
        """
```