How to use

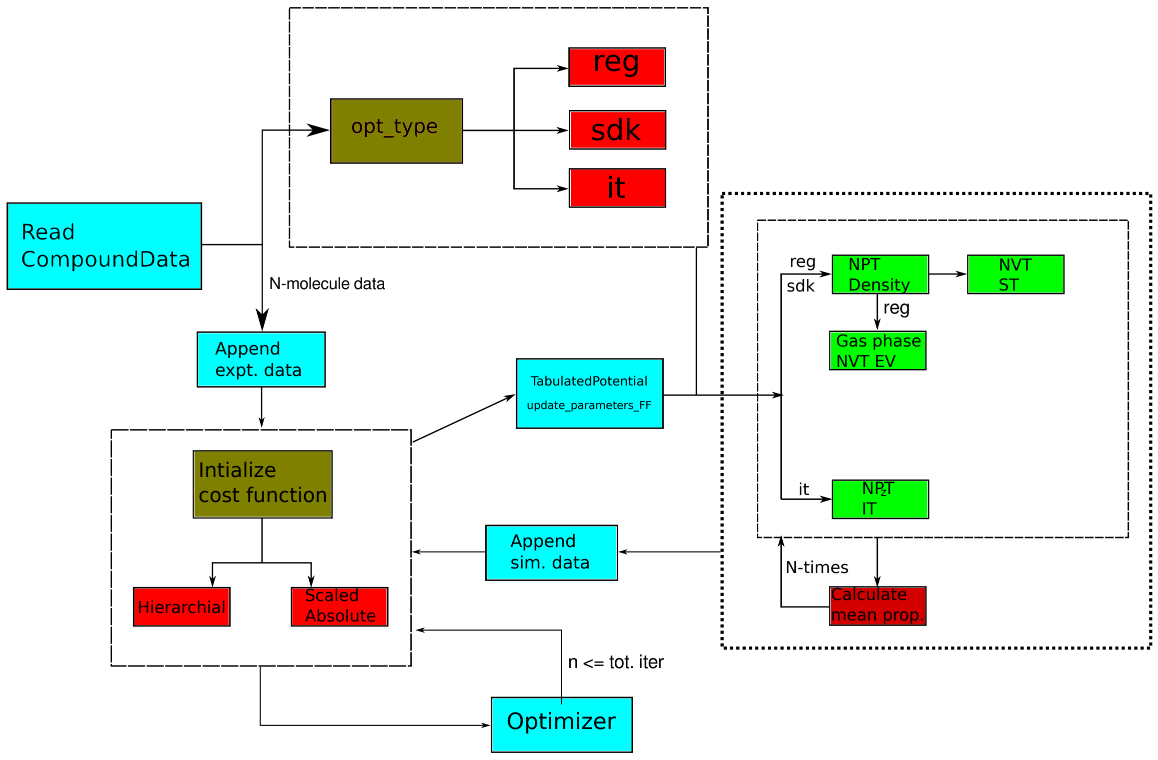

Figure 1.1. The algorithm chart of the pipeline implemented for carrying out Bayesian optimization in this work.

This section provides a summary of the code and the input files (the pipeline is provided in Fig. 9.1). The initialization and deployment are provided in steps. For the present example, 1-Pentanol (PNOL), 1-Hexanol (HXOL) and 4-cis-Decenol (DNOL) molecules are used. Some expertise with GROMACS software is required to implement the code. Additionally, the code, as usual, is never bug-free; similar to all codes, knowledge of Python is a requirement.

STEP 1: INITIALIZATION OF FILES

Folder creation: Prior to executing the code and initializing it in a given directory the following folders with the given names must be created. These are:

1. `MDP_FILES`

• `minim.mdp`	(Minimization)
• `eq1.mdp`	(First NPT equilibration at 2 fs)
• `eq2.mdp`	(Second equilibration at 5 fs)
• `md.mdp`	(Production at 5 to 10 fs)
• `eqnvt.mdp`	(First NVT equilibration at 2 fs)
• `eqnvt1.mdp`	(Second equilibration at 5 fs)
• `mdnvt.mdp`	(Production at 5 to 10 fs)
• `IT_equil.mdp`	(NPAT equilibration at 2 fs)
• `IT_equil1.mdp`	(NPAT equilibration at 5 fs)
• `IT_prod.mdp`	(NPAT production at 5 to 10 fs)

2. `PROPERTY_DATA`
   - `ALC_PNOL_HXOL_DNOL.prp`

     Depending on the type of optimization the properties needed for optimization are added.

     `reg:` Density, Surface tension, Enthalpy of vaporization

     `sdk:` Density, Surface tension

     `it:` Interfacial tension
3. `MOLECULE_ITP`
   - `PNOL.itp, HXOL.itp, DNOL.itp`

     Topology files of all the molecules needed for the optimization procedure.
4. `FORCEFIELD_ITP`
   - `Parameters.itp`

     The parameter file contains all the relevant parameters for bonded interactions and nonbonded interactions. However, for the nonbonded parameters which need to be optimized, a modification is required, as follows:

     ```
     O1    CM    1   disp_o1_cm_   rep_o1_cm_
     ```
   - `Parameters_updated.itp`

     The parameters needed to run the simulation for a given iteration cycle are updated in the current file. This file is read by the `topol.top` file as the FF parameters.
5. `TABULATED_POTENTIAL`
   - `table.xvg, table_X_X.xvg, table_X_Y.xvg, table_Y_Y.xvg`

Since the non-standard Mie potential is implemented, tabulated potential is needed. To learn more about tabulated potential, GROMACS manual can be referred.

6. `PNOL,HXOL,DNOL`
   - `to_min.gro`
     Initial file to minimize. This file contains the positions of *N* molecules in the GRO format.
   - `topol.top`
     Topology file of the *N* molecules.
   - `index.ndx`
     The index file for *N* molecules.
   - `topol_gas.top`
     Topology file for a single molecule for gas phase simulations.
   - `index_gas.ndx`
     The index file for a single gas phase molecule.
7. `PROTOCOLS`
   - `file_transfer_protocol.file`
     All files needed to carry out the simulations for each iteration are stored here, and the workflow transfers them to each parameter set folder.
   - `nptlog.file`
     GROMACS workflow to carry out NPT simulations for density are scripted here.
   - `nvtgas.file`
     GROMACS workflow to carry out NVT simulations for gas phase simulations are scripted here.
   - `nvtlog.file`
     GROMACS workflow to carry out NVT simulations for surface tension are scripted here.
   - `snptlog.file`
     GROMACS workflow to carry out NPAT simulations for interfacial tension are scripted here.

STEP 2: DESCRIPTION OF CODE METHODS

1. **CompoundData:**

In this version, the `CompoundData` class is used to reading and updating the experimental data. The function is now a method of the `CompoundData` class, and all of the data lists are now instance variables. To use this code, you would create an instance of the `CompoundData` class and call the `read_compound_data` method. Here's an example:

```
compound_data=offb.CompoundData('data.txt', opt_type='reg')
compound_data.read_compound_data()

# Access the data lists as instance variables of the         #
print(compound_data.MOLECULE_NAME)
print(compound_data.DENSITY)
```

The `CompoundData` object is created with the file name and `opt_type` argument, and the `read_compound_data` method is called to update the data. Finally, the data lists can be accessed as instance variables of the `compound_data` object.

2. **update_mdp_with_code:**

`update_mdp_with_code()` updates MDP files with specific codes and groups. It takes two arguments, `opt_type` and `reason`, which determines the type of simulation and the reason for the simulation, respectively. The function first accesses two lists called `CODE` and `GROUP` from an instance of the class `CompoundData`. These lists contain codes and groups that will be used to update the MDP files.

The function then loops through each code in the `CODE` list and switches to the corresponding directory. Within the directory, the function checks the `opt_type` and `reason` to determine which MDP files to update and how. For example, if `opt_type=reg` and `reason=homo`, the function will loop through a list called `mdp_lists` and copy each file to the current directory.

3. **TabulatedPotential:**

This code defines a class called `TabulatedPotential` that creates a table of potential energy values for a given set of parameters. The class has a method called `update` that saves the table to a file in a specific format. The class can be used to generate tables for different parameter sets by creating an instance of the class with the desired parameters and calling the `update` method. The example usage provided shows how to create a specific table and save it to a file.

Example usage of the code can be noted below.

```
tp = TabulatedPotential(12.0, 6.0)
tp.update(table='specific', group='mygroup')
```

4. **<u>update_temperature_in_mdp_files</u>:**

`update_temperature_in_mdp_files` function takes in the arguments `temp_K`, `opt_type`, and `reason`. Following which, it first determines which MDP files to update based on the `opt_type` and `reason` arguments. It then calls the `replace_temperature_in_mdp_file` function for each MDP file.

5. **<u>FileTransfer:</u>**

The code defines a class `FileTransfer` that has two methods: `mkdir_and_cd` and `mkdir_and_transferfiles`. The object instance initializes the argument `dirpath`, `opt_type`, and `reason`.

The `mkdir_and_cd` method creates a new directory and copies a workflow file to the new directory based on the `opt_type` argument. After creating the directory, the method changes the current working directory to the newly created directory.

6. **<u>update_parameters_of_FF:</u>**

The function `update_parameters_of_FF` updates specific parameters in a FF ITP file based on the user-provided input. The function takes a list of parameter groups (`parameter_groups`) and additional arguments in `kwargs` to specify values for epsilon, sigma, and the lambdas. The initialization FF file is assumed to be `Parameters.itp` in the `FORCEFIELD_ITP/` folder.

Then the code searches for the `disp_x_x_` and `rep_x_x_`, and updates these into simulation readable numbers. The output is saved into a new file called `Parameters_updated.itp` in the `FORCEFIELD_ITP/` folder.

An example for user defined input is provided here:

```
    parameter_groups = [
            ('MOL1',   'MOL1',   {'eps':   'eps_1',   'sig':   'sig_1',
'lamb_1': 'lamb1_1', 'lamb_2': 'lamb2_1'}),
            ('MOL2',   'MOL2',   {'eps':   'eps_2',   'sig':   'sig_2',
'lamb_1': 'lamb1_2', 'lamb_2': 'lamb2_2'})
        ]
    kwargs = {
            'eps_1': 1.5,
            'sig_1': 3.0,
            'lamb1_1': 1.8,
            'lamb2_1': 1.2,
            'eps_2': 2.0,
            'sig_2': 2.5,
            'lamb1_2': 2.5,
            'lamb2_2': 3.0
        }
    update_parameters_of_FF(parameter_groups, **kwargs)
```

7. **Simulation:**

The class takes in three arguments during initialization - `dirpath`, `opt_type`, and `reason`.

The `sim_density` method performs a density simulation by calling other methods based on the `opt_type` specified. If the `opt_type` is `reg` or `sdk`, it performs an NPT simulation followed by an NVT simulation. If the `opt_type` is `it`, it performs an interfacial tension simulation. The `gas_phase_nvt_box_simulation` method selects a single molecule for gas phase and runs the simulation in the gas phase if the `opt_type` is `reg`.

The xyz_npt_nvt method returns the dimensions of the simulation box. If the opt_type is reg or sdk, it returns the box dimensions for an NVT simulation, where the z-direction of the box is converted to 3 times that of the original NPT box. The sim_st method performs a surface tension simulation, if the opt_type is reg or sdk. The sim_it method performs an interfacial tension simulation if the opt_type is it.

8. **SimulationData:**

The class SimulationData contains several methods for calculating various properties of a molecular simulation. To create an object instance of this class a directory path containing the relevant simulation output files is needed as an argument for initializations. The methods in this class use this directory path to access the simulation output files.

The first method, _run_gmx_energy is a helper method that runs the GROMACS gmx energy command with the given arguments and input string. It captures the command output and raises an error if the command fails.

The method, calculate_mean_density, calculates the average density of the simulation by running the _run_gmx_energy method with different arguments and input string. It then reads the data from the output file, calculates the mean density, and returns it. The method, calculate_mean_st, calculates the average surface tension of the simulation. It then reads the data from the output file, calculates the mean surface tension, and returns it. Finally, the calculate_mean_enthalpy_of_vaporization, first runs the _run_gmx_energy to generate two output files of potential energy in the liquid phase and the in the gas phase. Then it evaluates the mean enthalpy of vaporization and returns it.

The calculate_mean_it method is similar to calculate_mean_st.

9. **cost_function:**

The cost_function evaluates the predictions made by a model for the physical properties of the bulk phase of molecules. It takes the predicted values of density, surface tension, evaporation enthalpy, and interfacial tension, depending on the opt_type and computes an objective value that represents the cost of these predictions.

The cost function can use two types of cost functions: hierarchical and scaled_absolute. The hierarchical cost function penalizes predictions based on the percentage error between the predicted and experimental values. It assigns a reward value of 1000 if the error falls within a certain tolerance percentage and decreases the reward linearly for errors outside the tolerance. The scaled_absolute cost function scales the absolute error between the simulation and experiment by the mean experimental value of the corresponding property over all molecules.

The objective value represents the cost/loss of the model's predictions and is used for parameter optimization.

10. <u>run_simulations_over_properties:</u>

The run_simulations_over_properties function performs sequential simulations of properties (1. Density, 2. EV and 3. ST) for the molecules included in the parameterization. One call of this function corresponds to one iteration.

The function takes several arguments:

- iniguess: A list of initial guess parameters.
- input_data: A dictionary containing input data.
- cost_kind: A string representing the type of cost calculation. It defaults to hierarchial.
- opt_type: A string representing the type of optimization. It defaults to reg.
- reason: A string representing the reason for simulation. It defaults to homo.

The function defines an internal execution function that performs the main simulation steps. It initializes empty lists for storing simulation results. It iterates over each compound in the compound data dictionary. Inside the iteration, it sets up the directory path, creates objects of Simulation and SimulationData classes, and evaluates the mean values depending on the opt_type and based on the availability of simulation files. The calculated values are appended to the corresponding lists. The total cost is calculated using the cost_function with appropriate arguments based on the opt_type and kind. Finally, the execution function returns the total cost, that is updated to a logging json file.

If an error occurs during execution, it changes the current working directory back to the original directory and retries the execution. If an error still occurs, it gives the iteration a value of 0 and tries the next set.

11. **data_file_writer_simulations:**

Writes data from iterations to a file in `json` format. This function takes the provided cost value and any additional key-value pairs from `kwargs` and writes them to a file specified by `iteration_output_file` in `json` format. Each iteration's data is appended to the file.

STEP 3: EXECUTION EXAMPLE

Assuming, that all the initialization is carried out in accordance with the Sec. 9.1. Then the execution of the code is carried out in the parent directory, as follows:

```
import Optimization_FF_Bayesian as offbcg
from bayes_opt import BayesianOptimization

prop_alc_file = "PROPERTY_DATA/ALC_PNOL_HXOL_DNOL.prp"
alc_opt_type = "reg"
Iter_outfile =
"Iteration_hier_comp_out_current_FF_PNOL_HXOL_DNOL.json"
cost_kind = "hierarchial"
reason = "hetero"
alcdata = offbcg.CompoundData(prop_tog_file,opt_type=tog_opt_type)
alcdata.read_compound_data()
input_alc_data=alcdata.data
tp = offbcg.TabulatedPotential(lamb_1=8.9901, lamb_2=6.9347)
tpp = offbcg.TabulatedPotential(lamb_1=8.9901, lamb_2=6.9347)
tp.update(table='specific', group='ALK_ALK')
tp.update(table='specific', group='ALK_ALC')
tpp.update(table='specific', group='ALC_ALC')
tp.update(table='default')
```

```
def ft(e_o1_ct2,s_o1_ct2,e_o1_cm,s_o1_cm,e_o1_cmd2,s_o1_cmd2):


      # Code implementation...
      lamb_1_alk_alc = 8.9901
      lamb_2_alk_alc = 6.9347

      parameter_tog_groups = [('o1', 'ct2', {'eps': 'e_o1_ct2',
'sig': 'sig_o1_ct2', 'lamb_1': lamb_1_alk_alc, 'lamb_2':
lamb_2_alk_alc}),
('o1', 'cm', {'eps': 'e_o1_cm', 'sig': 'sig_o1_cm', 'lamb_1':
lamb_1_alk_alc, 'lamb_2': lamb_2_alk_alc}),
('o1','cmd2',{'eps': 'e_o1_cmd2', 'sig': 'sig_o1_cmd2', 'lamb_1':
lamb_1_alk_alc, 'lamb_2': lamb_2_alk_alc})]


      kwargs = {
'e_o1_ct2': e_o1_ct2,
'sig_o1_ct2': s_o1_ct2,
'e_o1_cm': e_o1_cm,
'sig_o1_cm': s_o1_cm,
'e_o1_cmd2': e_o1_cmd2,
'sig_o1_cmd2': s_o1_cmd2
}


      offbcg.update_parameters_of_FF(parameter_alc_groups,**kwargs)
      ini =
np.array([e_o1_ct2,s_o1_ct2,e_o1_cm,s_o1_cm,e_o1_cmd2,s_o1_cmd2])
      k =
offbcg.run_simulations_over_properties(ini,input_data=input_alc_data
,cost_kind=cost_kind,opt_type=tog_alc_type,reason=reason)
offbcg.data_file_writer_iterations(iteration_output_file=Iter_outfil
e,cost=k,**kwargs)
      return k
```

```
bounds =
{'e_o1_ct2':(0.7,1.5),'s_o1_ct2':(3.5,4.2),'e_o1_cm':(0.7,1.5),'s_o1
_cm':(3.9,4.4),'e_o1_cmd2':(0.8,1.9),'s_o1_cmd2':(3.5,4.3)}
optimizer =
BayesianOptimization(f=ft,pbounds=bounds,random_state=1000)
optimizer.probe(params={'e_o1_ct2':0.9573,'s_o1_ct2':3.900,'e_o1_cm'
:1.3793,'s_o1_cm':4.001,'e_o1_cmd2':1.2,'s_o1_cmd2':3.850},lazy=True
)
optimizer.maximize(init_points=30,n_iter=50,acq='ucb',kappa=2)
```

The key components and steps needed to execute the code include:

1. **Importing modules:** The following modules are imported. The module includes the optimization workflow (in-house) called `Optimization_FF_Bayesian`, and the `BayesianOptimization` from the module `bayes_opt` (open-source,https://github.com/bayesian-optimization/BayesianOptimization.git).

2. **Setting up initial configuration:** This includes the input files needed to setup the optimization process. The properties are saved in the instance `prop_alc_file`, the optimization type `reg` is stored as a variable `alc_opt_type`.

3. **Creating data objects:** Initializing instances of data objects (`alcdata`, `tp`, `tpp`) from the module `Optimization_FF_Bayesian`. These objects are to store compound data, potential energy parameters and tabulated potentials.

4. **Defining the optimization function:** The function `ft` is defined to run simulations and update parameters for the optimization. It takes the input parameters per iteration as arguments, updates the FF file based on the input, carries out the simulations, and returns a cost value. The output is stored in a `json` file.

5. **Setting parameter bounds and initializing the Bayesian optimization:** The parameter bounds of the optimization are defined in the instance `bounds`, and the Bayesian optimization instance `optimizer` is created using the defined function `ft`, `bounds` and `random_seed`.

6. **Probing initial points:** Based on chemical intuition, probes can be carried out on the optimization surface, which will be updated in the prior. This can be done with the `probe` method of the `optimizer` object.

7. **Running Bayesian optimization:** To run Bayesian optimization the method `maximize`, is implemented. One can assign a specified number of initial exploration points before the optimization as `init_points`, and the subsequent iterations, when the optimization is turned on, is specified in `n_iter`, utilizing the upper confidence bound acquisition function and a exploration-exploitation (κ) trade-off factor `kappa`. In the case described here the initial exploration/random exploration is equal to 30 points. Following this, 50 iterations of optimization take place with the κ = 2.0. The iterations can be increased subsequently, along with the tuning of the κ factor.