CAVIAR Guide for Developers

Morad Biagooi

2020-07-17

Prerequisite knowledge for CAVIAR developers

- C++ programming (Polymorphism, Preprocessors)

- CMake

- Molecular Dynamics basic algorithms

Part I: a C++ review for CAVIAR

CAVIAR source code looks complicated, why is it developed like that?

We had the following goals while developing CAVIAR:

- Creation of the most flexible design

- Automation the code as much as possible to reduce typos, errors and mental workload for developers

- An input script with a functionality that make the user independent of changing C++ source code.

The above items resulted in:

- Creation of an MD library (such as Espresso) instead of an MD software (like LAMMPS (LAMMPS also can be used as a c++ library))

- Developing almost everything as C++ classes with polymorphism

- Creation of CASL language and interpreter

What do we mean by “library”?

- We instantiate objects of the library by using the constructor functions defined in the class descriptions of the libraries:

```
class Lj {
public:
  Lj (type1 arg1, type2 arg2) {…}
…
};


Lj my_force(arg1, arg2); // class instantiation
```

What do we mean by “library”?

- The instantiated objects does not know each other unless we tell them.

Objects instantiation:

```
Lj my_force(arg1, arg2);
Verlet_list my_nbl(arg3, arg4);
```

Setting Variables:

```
my_force.neighborlist = &my_nbl;  // C++
my_force.set_neighborlist (my_nbl);  // C++

my_force set_neighborlist my_nbl   #CASL Script

my_force.set_neighborlist(my_nbl)   #Python Script
my_force.neighborlist=my_nbl          #Python Script
```

Polymorphism in CAVIAR

```
class Force_field : public Pointers {
  ...
  void calculate_acceleration() = 0; // Force_field becomes an abstract class by setting to zero.
  Atom_data *atom_data;
  ...
};

class Lj : public Force_field {
  ...
  void calculate_acceleration() {...}
  ...
};

class Electrostatic : public Force_field {
  ...
  void calculate_acceleration() {...}
  ...
};
```

Why polymorphism?

```
Class Md_simulator : public Pointers{
  ...
  std::vector<Force_field *> force_fields;
  ...
};

int main () {
  ...
  md_simulator::Basic my_simulator (arg1);

  force_field::Lj my_force_1 (arg2);
  force_field::Electrostatic_ewald my_force_2 (arg3);


  my_simulator.force_fields.push_back(&my_force_1);  // Possible with polymorphism
  my_simulator.force_fields.push_back(&my_force_2);  // Possible with polymorphism
  ...
}
```

Why polymorphism?

```
class Force_field : public Pointers {
  ...
  void calculate_acceleration() = 0;
  ...
};

Class Md_simulator : public Pointers{
  ...
  std::vector<Force_field *> force_fields;
  void time_step () {
    ...
    For (auto f : force_fields) {
      f → calculate_acceleration(); // Possible only with polymorphism
    }
    ...
  }
  ...
};
```

A Base Class Example: Class declarations

```
#ifndef CAVIAR_OBJECTS_FORCEFIELD_H
#define CAVIAR_OBJECTS_FORCEFIELD_H
#include "caviar/utility/objects_common_headers.h"
namespace caviar {
namespace objects {
class Atom_data;      // Class Declaration (I):     Instead of #include "Atom_data.h" → less compile time
class Domain;         // Class Declaration (II):    Instead of #include "Domain.h" → less compile time
class Neighborlist;   // Class Declaration (III):   Instead of #include "Neighborlist.h" → less compile time
class Force_field  : public Pointers {
 public:
  Force_field (class CAVIAR *);
  virtual bool read (class caviar::interpreter::Parser *) = 0;
  virtual void calculate_acceleration () = 0;
  virtual double potential (const Vector<double> &);
  virtual Vector<double> field (const Vector<double> &);
  …
  class objects::Atom_data *atom_data;          // It can be defined because of the declaration (I)
  class objects::Domain *domain;                // It can be defined because of the declaration (II)
  class objects::Neighborlist *neighborlist;    // It can be defined because of the declaration (III)
  …
};
} //objects
} // namespace caviar
#endif
```

Class declarations. Reason

If a header is changed, all of the files that has included it are need to be recompiled.

For example If you change ‘Pointer.h’ or ‘CAVIAR.h’ in the package, almost all of the project source files needs to be recompiled.

CAVIAR have more than 500 source files and re-compiling all of them takes a long time.

Using declarations instead of include is possible when we are working with pointers.
Less include -> less dependency

CAVIAR Major Namespace

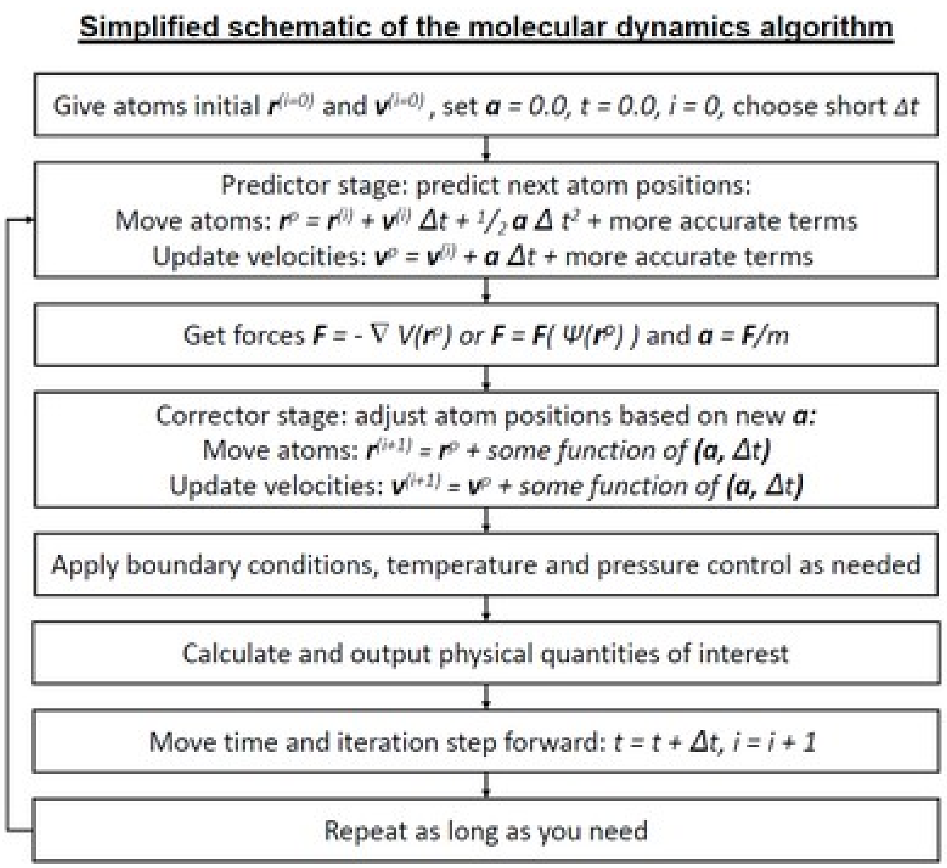
```
namespace caviar {
  namespace objects {
  … // (containing the classes that have different
          algorithms for CAVIAR MD simulations)
  }

  namespace interpreter {
  … // (containing the input/output classes that are
          used to create and call objects classes)
  }
}
```

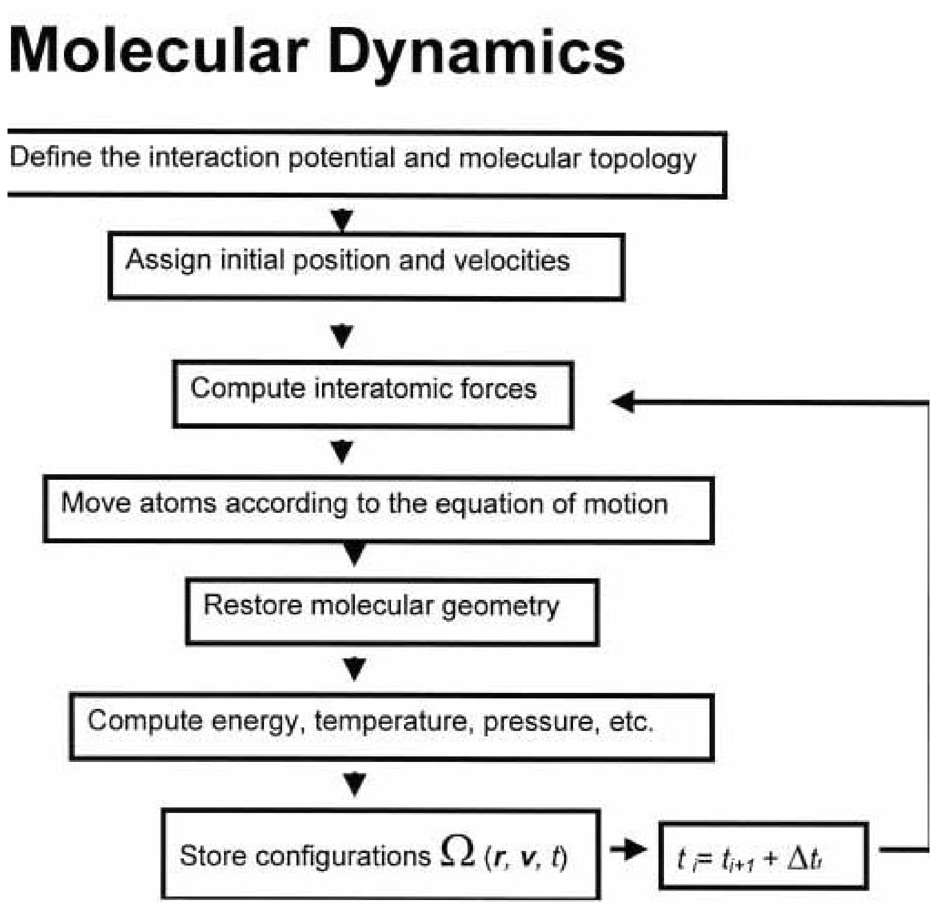
CAVIAR Major Namespace

```
namespace caviar {
 namespace objects {
  class Atom_data;
  namespace atom_data {
    class Basic;
  }
  class Force_field;
  namespace force_field {
   class Lj;
   class Electrostatic;
   class Plt_dealii;
   ...
  }
  ...
 }
}
```

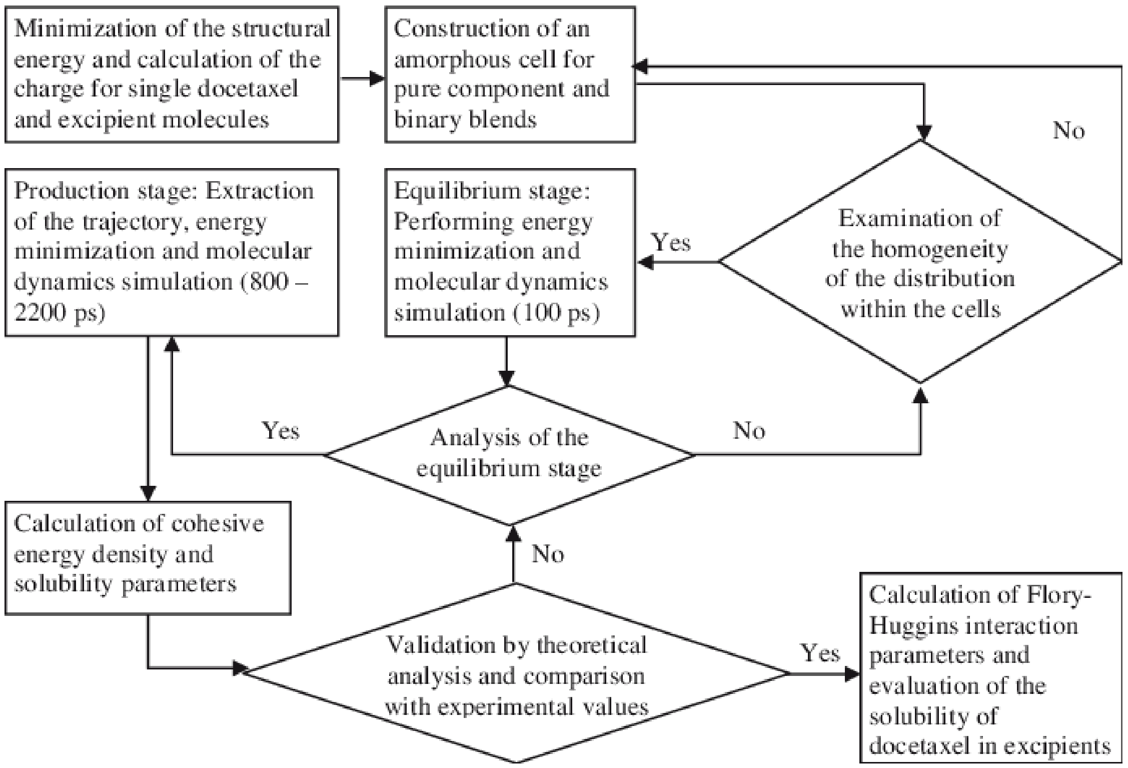
Part II: CAVIAR MD Pipeline

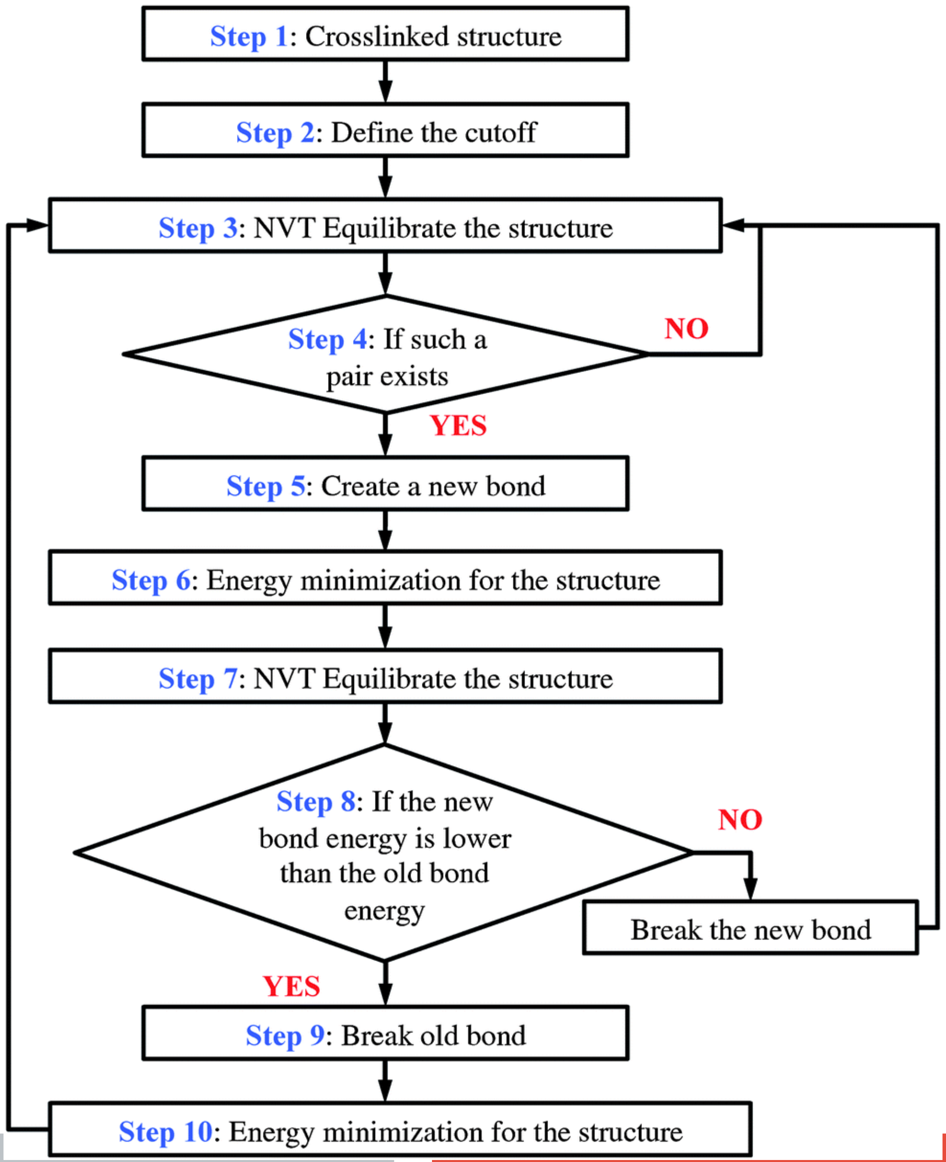
Molecular Dynamics Flowcharts


Simplified schematic of the molecular dynamics algorithm
Give atoms initial $\mathbf{r}^{(i=0)}$ and $\mathbf{v}^{(i=0)}$, set $\mathbf{a} = 0.0$, $t = 0.0$, $i = 0$, choose short Δt
Predictor stage: predict next atom positions:
Move atoms: $\mathbf{r}^{p} = \mathbf{r}^{(i)} + \mathbf{v}^{(i)} \Delta t + {}^{1}/_{2}\, \mathbf{a}\, \Delta t^{2}$ + more accurate terms
Update velocities: $\mathbf{v}^{p} = \mathbf{v}^{(i)} + \mathbf{a}\, \Delta t$ + more accurate terms
Get forces $\mathbf{F} = -\nabla V(\mathbf{r}^{p})$ or $\mathbf{F} = \mathbf{F}(\Psi(\mathbf{r}^{p}))$ and $\mathbf{a} = \mathbf{F}/m$
Corrector stage: adjust atom positions based on new $\mathbf{a}$:
Move atoms: $\mathbf{r}^{(i+1)} = \mathbf{r}^{p}$ + some function of $(\mathbf{a}, \Delta t)$
Update velocities: $\mathbf{v}^{(i+1)} = \mathbf{v}^{p}$ + some function of $(\mathbf{a}, \Delta t)$
Apply boundary conditions, temperature and pressure control as needed
Calculate and output physical quantities of interest
Move time and iteration step forward: $t = t + \Delta t$, $i = i + 1$
Repeat as long as you need

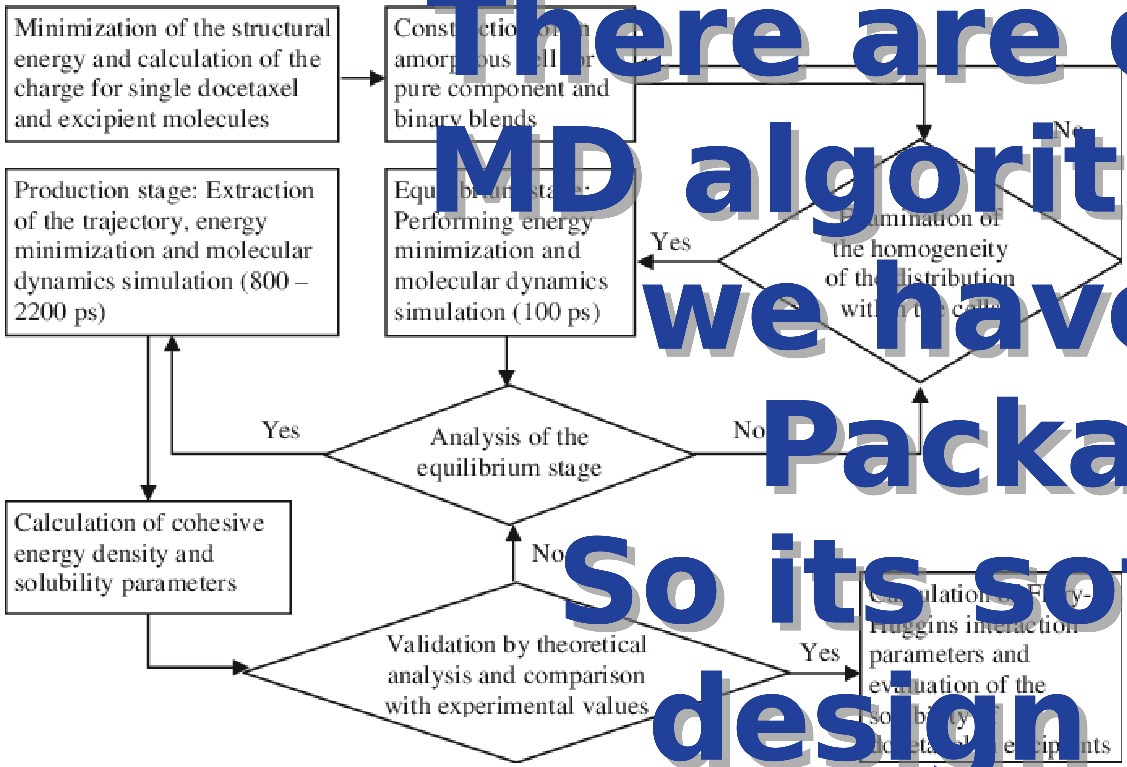

Molecular Dynamics
Define the interaction potential and molecular topology
Assign initial position and velocities
Compute interatomic forces
Move atoms according to the equation of motion
Restore molecular geometry
Compute energy, temperature, pressure, etc.
Store configurations Ω ($\mathbf{r}$, $\mathbf{v}$, t)
$t_i = t_{i+1} + \Delta t_i$

Molecular Dynamics Flowcharts

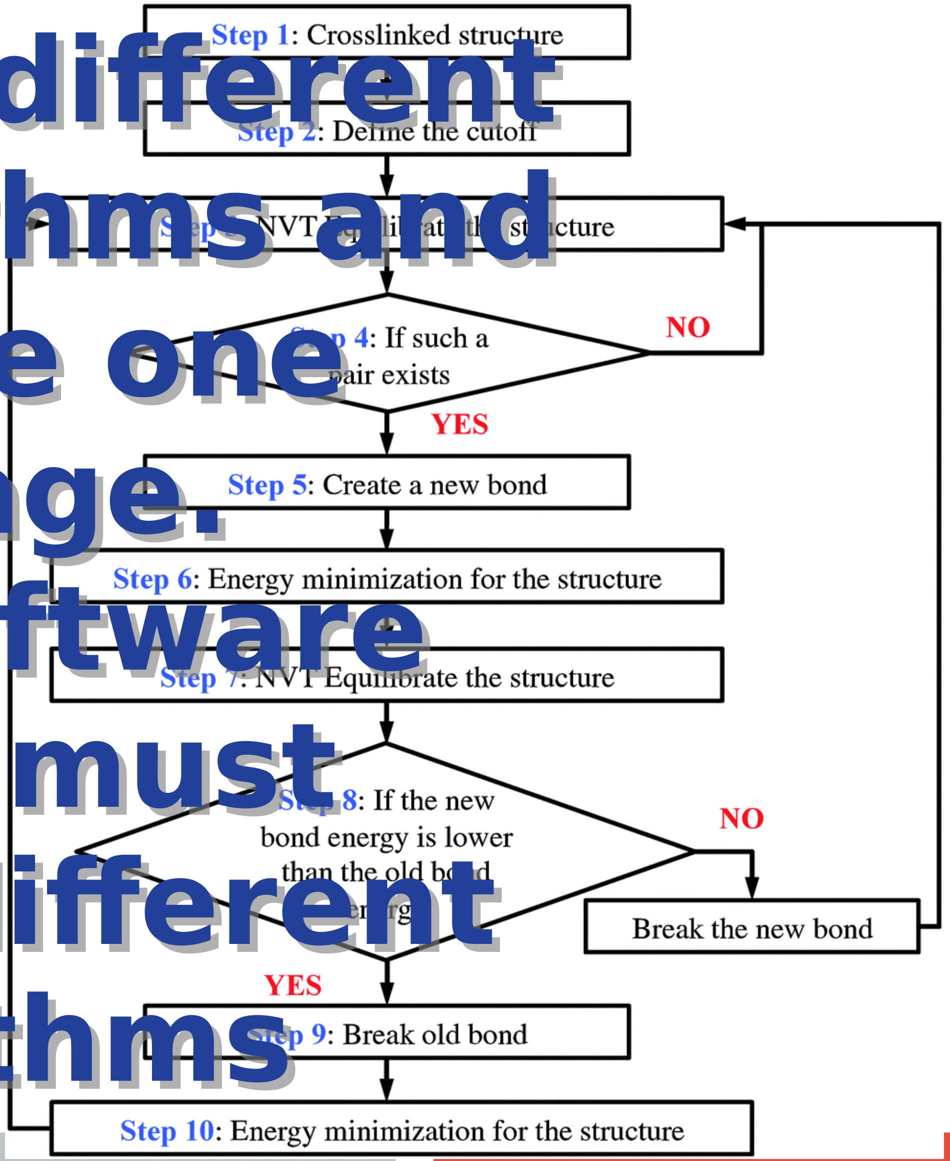

Minimization of the structural energy and calculation of the charge for single docetaxel and excipient molecules
Construction of an amorphous cell for pure component and binary blends
No
Examination of the homogeneity of the distribution within the cells
Yes
Equilibrium stage: Performing energy minimization and molecular dynamics simulation (100 ps)
Production stage: Extraction of the trajectory, energy minimization and molecular dynamics simulation (800 – 2200 ps)
Yes
Analysis of the equilibrium stage
No
Calculation of cohesive energy density and solubility parameters
No
Validation by theoretical analysis and comparison with experimental values
Yes
Calculation of Flory-Huggins interaction parameters and evaluation of the solubility of docetaxel in excipients


Step 1: Crosslinked structure
Step 2: Define the cutoff
Step 3: NVT Equilibrate the structure
Step 4: If such a pair exists
NO
YES
Step 5: Create a new bond
Step 6: Energy minimization for the structure
Step 7: NVT Equilibrate the structure
Step 8: If the new bond energy is lower than the old bond energy
NO
Break the new bond
YES
Step 9: Break old bond
Step 10: Energy minimization for the structure

Molecular Dynamics Flowcharts



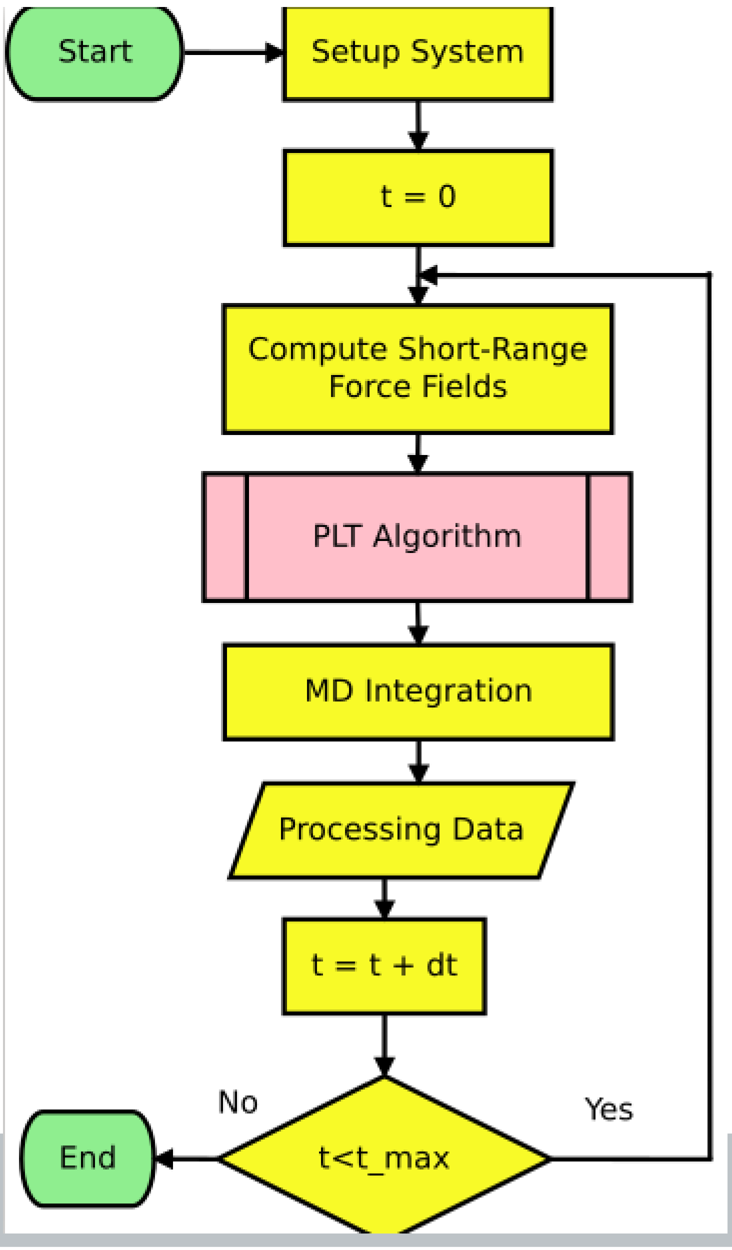
Minimization of the structural energy and calculation of the charge for single docetaxel and excipient molecules
Construction of amorphous cell for pure component and binary blends
Production stage: Extraction of the trajectory, energy minimization and molecular dynamics simulation (800 – 2200 ps)
Equilibrium stage: Performing energy minimization and molecular dynamics simulation (100 ps)
Examination of the homogeneity of the distribution within the cells
Yes
No
Analysis of the equilibrium stage
Yes
No
Calculation of cohesive energy density and solubility parameters
Validation by theoretical analysis and comparison with experimental values
No
Yes
Calculation of Flory-Huggins interaction parameters and evaluation of the solubility of docetaxel in excipients



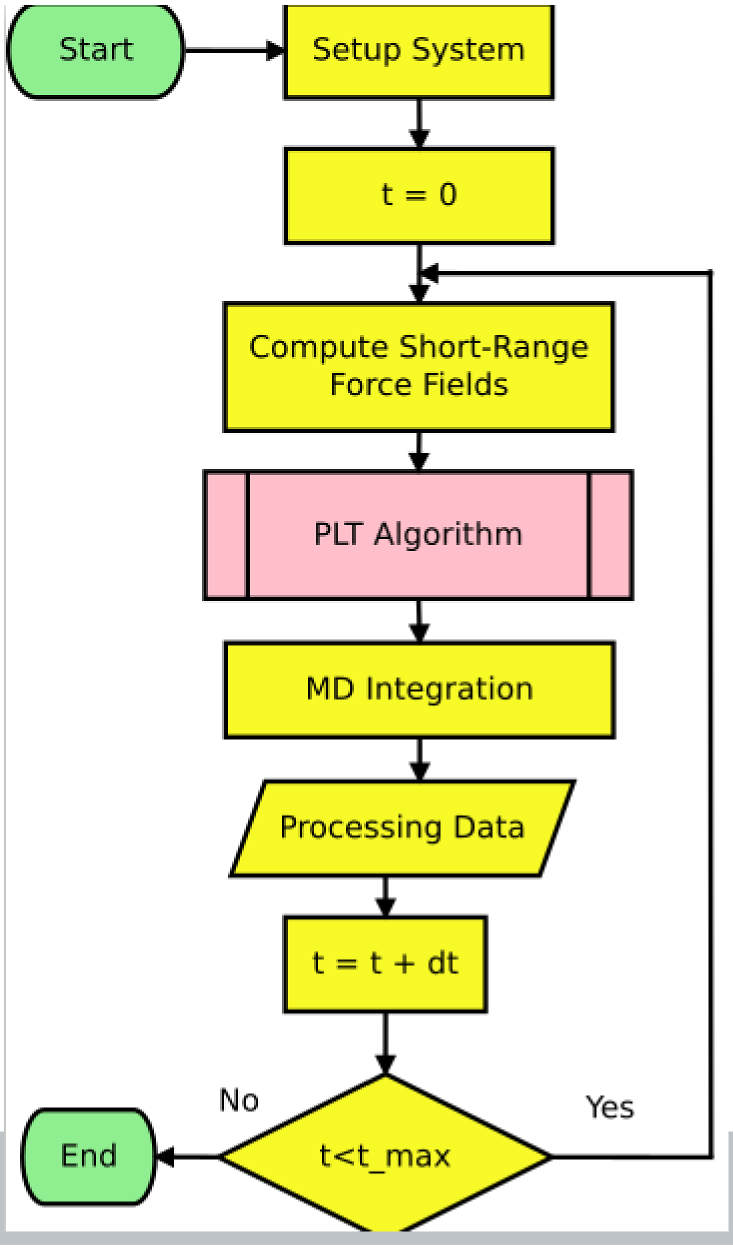
Step 1: Crosslinked structure
Step 2: Define the cutoff
Step 3: NVT Equilibrate the structure
Step 4: If such a pair exists
NO
YES
Step 5: Create a new bond
Step 6: Energy minimization for the structure
Step 7: NVT Equilibrate the structure
Step 8: If the new bond energy is lower than the old bond energy
NO
Break the new bond
YES
Step 9: Break old bond
Step 10: Energy minimization for the structure

There are different MD algorithms and we have one Package. So its software design must support different Algorithms

CAVIAR MD Flowchart

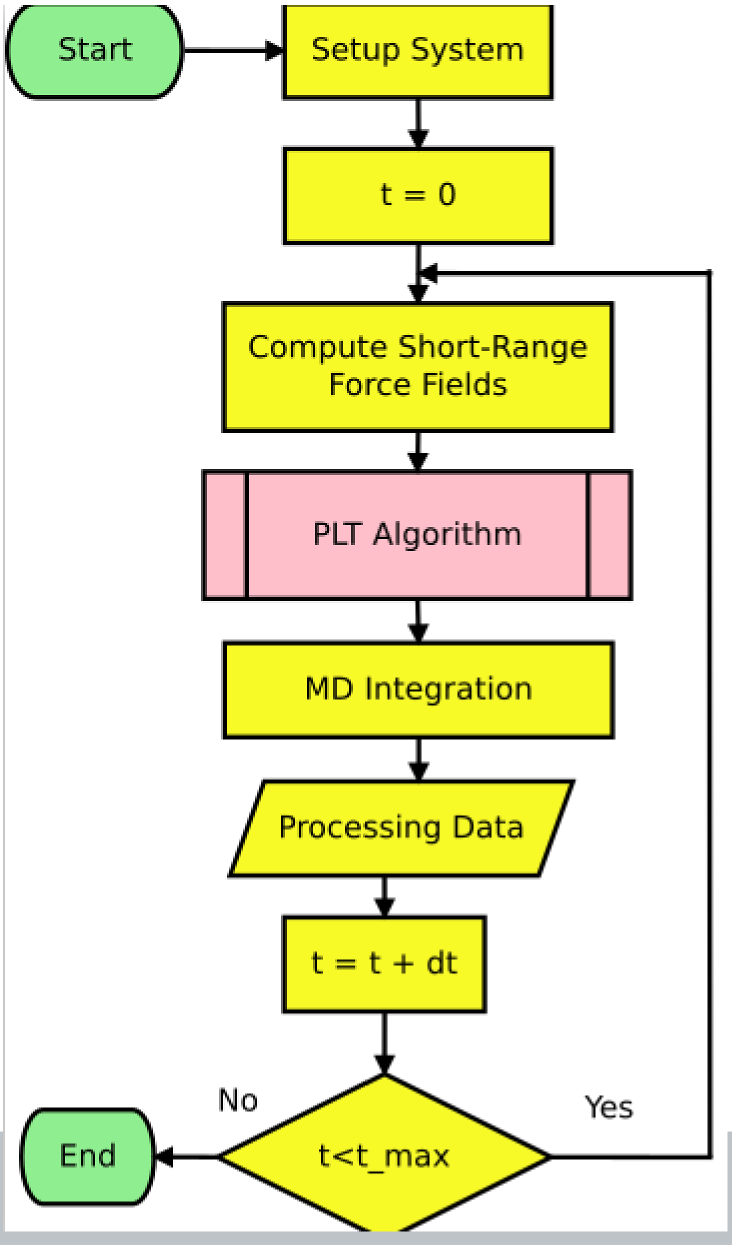


CAVIAR MD Flowchart



Where is the main MD time loop in CAVIAR package?

CAVIAR MD Flowchart



Where is the main MD time loop in CAVIAR package?

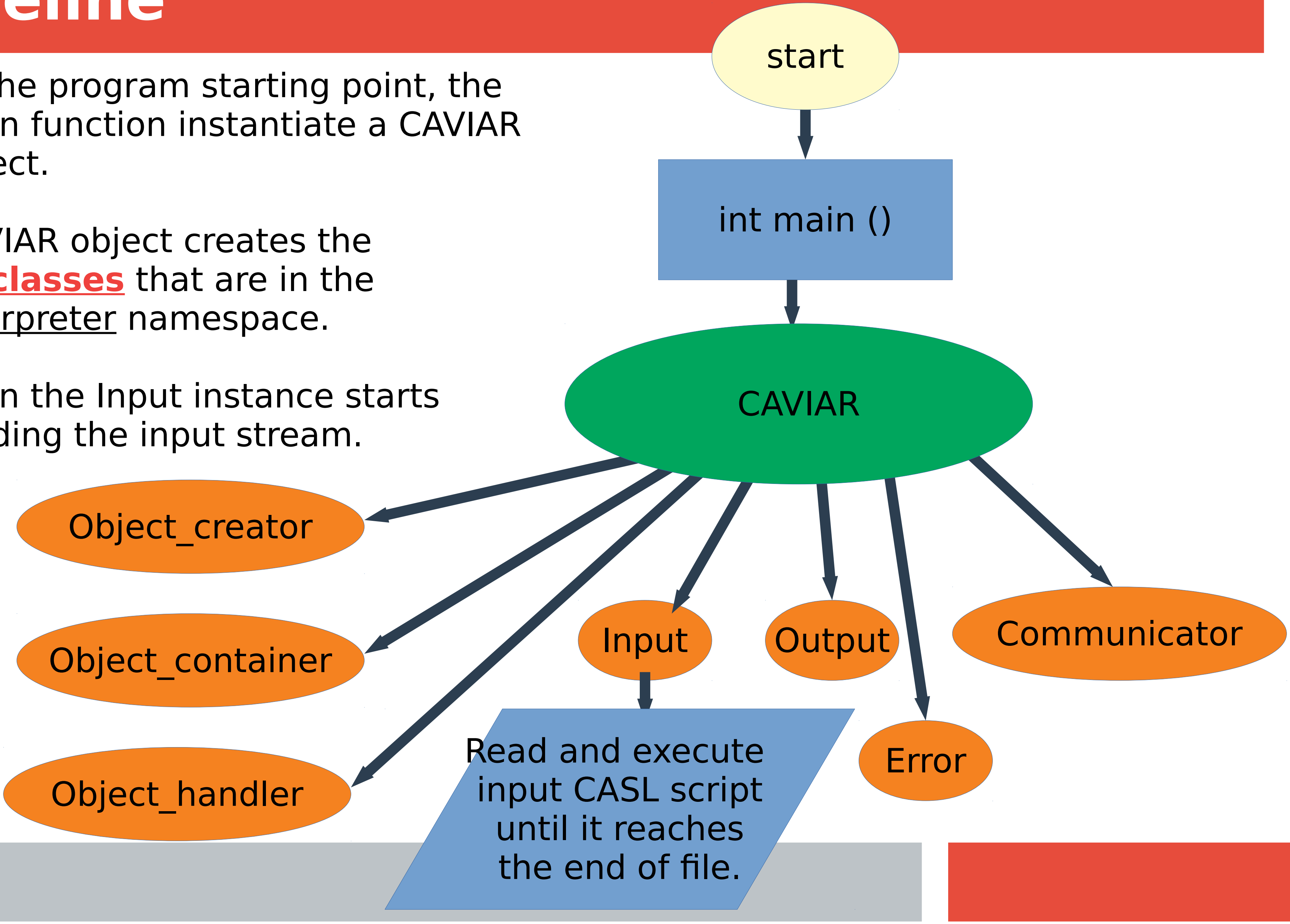
There's no main loop. One can instantiate an Md_simulator class to have such functionality. Also one can develop his or hers own custom MD loop.

CAVIAR 1.0 (without python) Pipeline

At the program starting point, the main function instantiate a CAVIAR object.

CAVIAR object creates the **IO classes** that are in the interpreter namespace.

Then the Input instance starts reading the input stream.

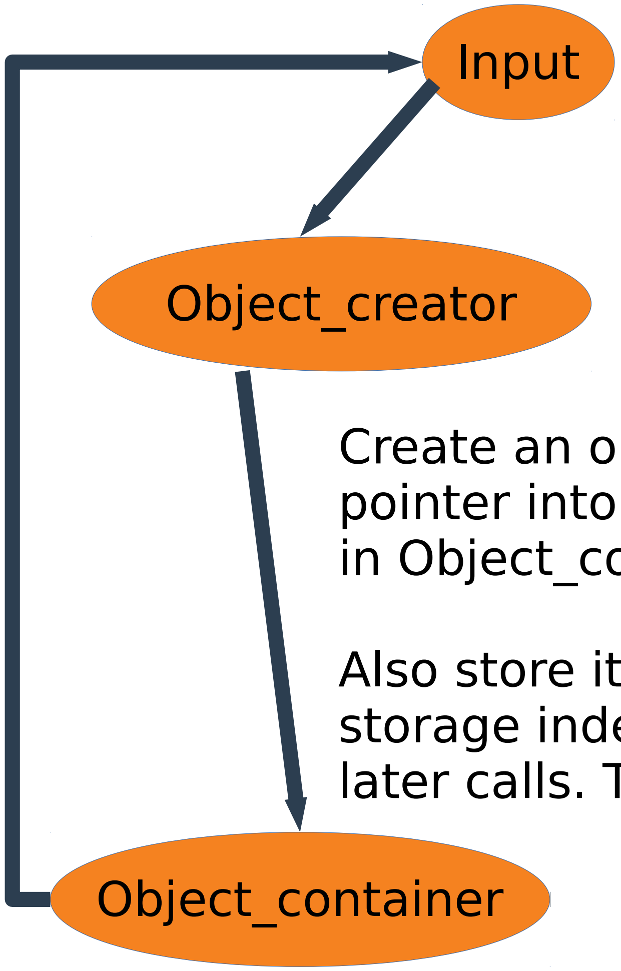


CAVIAR 1.0 (without python) Pipeline

There are three different type of commands:

I) Object Creation

```
force_field Lj my_force
```



Create an object and put its pointer into the correct container in Object_container.

Also store its name, type and storage index in a dictionary for later calls. Then return to input.

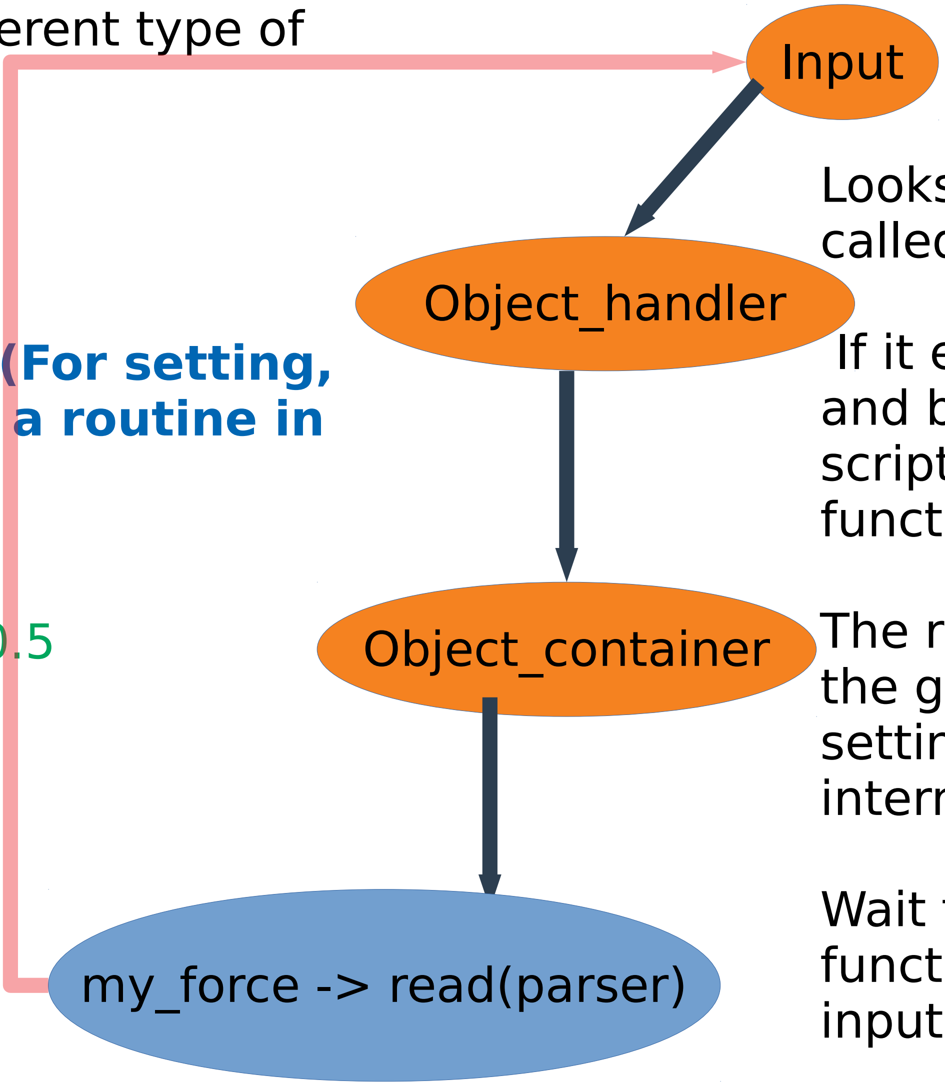
CAVIAR 1.0 (without python) Pipeline

There are three different type of commands:

I) Object Creation

II) Object Calling (For setting, getting or calling a routine in that object)

my_force cutoff 0.5



Looks for an object with the called name.

If it exists, find its pointer and by using it, pass the script parser to its read() function.

The read function interprets the given command such as setting or calling another internal function.

Wait for the return from the function. Then return to input to get the next line.

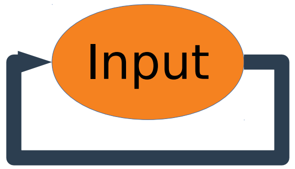
CAVIAR 1.0 (without python) Pipeline

There are three different type of commands:

I) Object Creation

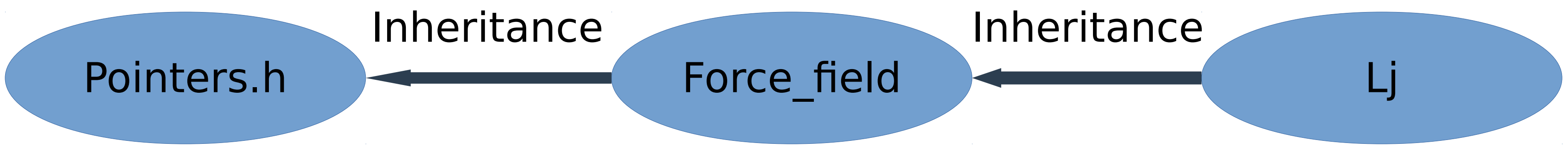
II) Object Calling (For setting, getting or calling a routine in that object)

- CASL commands (If Condition, Loop,...)



The input class handles the CASL commands by calling its internal functions.

CAVIAR 1.0
objects interface:



Pointer class has :
pointer to Error
pointer to Output
pointer to Input
pointer to Communicator
pointer to Obj. Container