OCpy

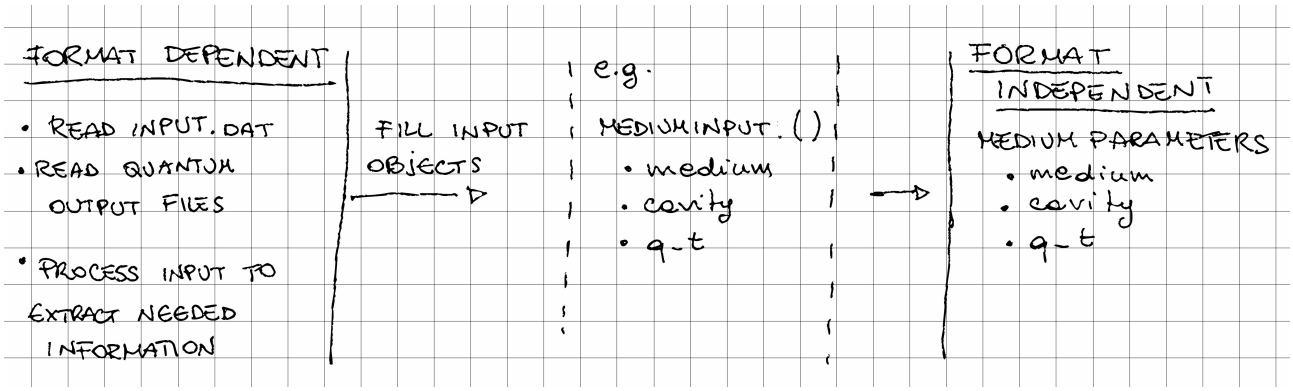
The following document don't have code written. The parst in gray are pseudo-code meant which reproduce approximately the real python code, summarizing and cleaning it, to allow to perform simple example and show the general structure.

General comments

Read interface and optimization process

In OCpy the user input "*read and set", or "read and process",* part and the "optimization" process are completely separated. The first need to know the input format (ocpy input, quantum files input..), the latter does not.

It would be possible to initialize and perform the optimization from a different software skipping the "*read and set",* or to rewrite the entire optimization procedure in a different language to improve the performance and keep the same *input.dat* file and "read and set*"* procedure. The bridge between these two parts are the *Input* classes an objects (see following) which are the format in which the optimization objects need to have the data to initialize the calculations.



Object Oriented

OCpy is object oriented. To understand it, you need to know the basics of object oriented programming. Inheritance is used and, even if it is not mentioned explicitly, composition programming is used. Reading the basic definition of these two mechanism can help in understanding the structure of the code.

Fast zero level concept you need to know to read the following: all the methods belong to classes.

If we want to use *method1* defined in

```
GeneralExampleClass():
          def method1():
                print(“test”)
```

we need to create an instance of *GeneralExampleClass()* and call its method:

```
testInstance = GeneralExampleClass()
testInstance.method1()
>> test
```

Similarity with fortran:

when there is an object made only by attributes, not methods, we are creating what in fortran is a “type”.
It is a collection of informations which characterize something.
e.g.

```
class Cube()
   side
   volume
   diagonal

class MediumParameters():
        medium
        cavity
        q_t
```

In OCpy we have *Input* objects *(FieldInput, MoleculeInput, etc),* the *Parameters* objects and the *SaveFile()* and *SaveRestart()* objects.

Format

Try to have one class per file, with the same name of the python file and of the class inside the file.
Sometimes this is not the case: e.g. *NamelistSections.py* contains several classes named *SectionNAME()* (*SectionSystem*, *SectionField*, etc) which all inherits from the abstract class ABCNamelistSection() in ABCNamelistSection.py. As all these classes are very similar and very short, it makes sense to have them all in the same file. Nevertheless this is an exception, not the rule.

Very general structure

The “read and set” part is done in SystemManager.py file by SystemManager() object.

Then OCManager() object in OCManager.py initializes the optimization objects and performs the optimization.
The general structure is:

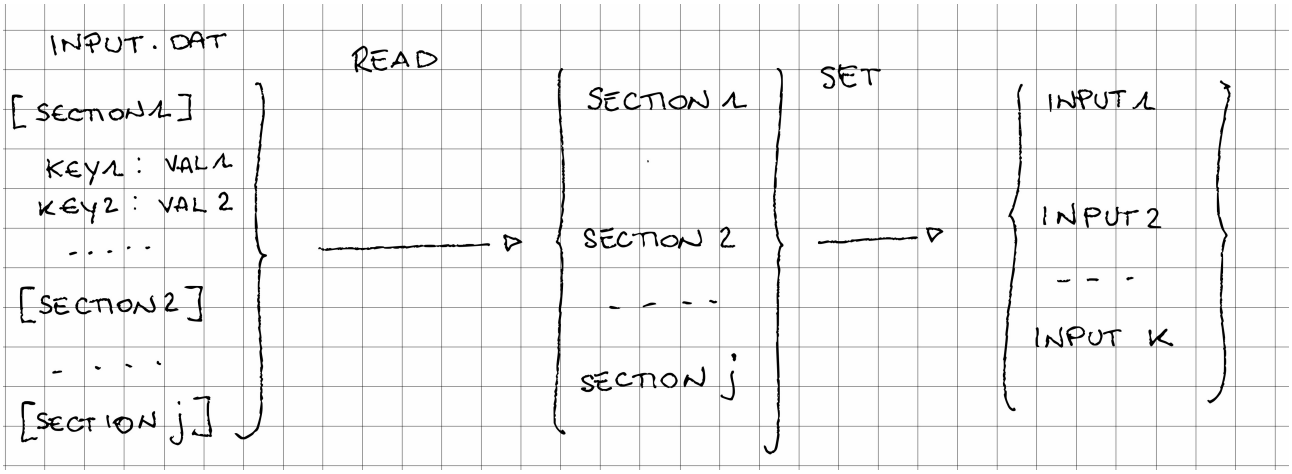
```
- run.py
    OC_system=SystemManager()
    OC_system.init_system(folder, namefile)
    OC_system.oc.iterate()

- SystemManager.py
    SystemManager():
        oc = OCManager()
        def init_system(folder, namefile):
           ...
           oc.init(...)
           ...

 - OCManager.py
     OCManager()
        def init(...)
        ...
        def iterate(...)
        ...
```

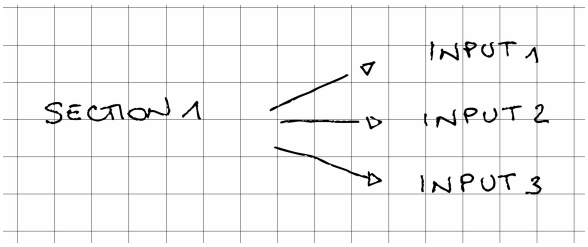
Interface

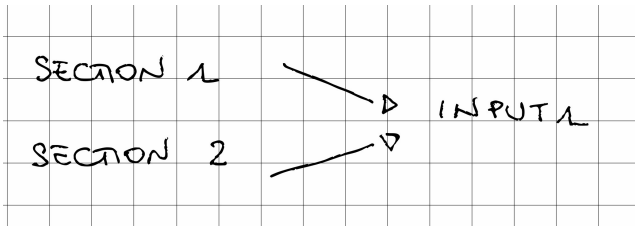
READ and SET procedure inside SystemManager.py



Given the number of Sections *j* in the i*nput.dat* file (SYSTEM, FIELD, WAVEFUNCTION, MEDIUM, OPTIMALC, SAVE), the "read" objects and methods read the file, check if the values are allowed (typos, consistency of the choices within Section and within different Sections) and temporary save them into one *SectionNAMELIST (SectionField, SectionSystem, etc)* object for each section in the *input.dat* file, without modifying them.

Then the *SetNAMEInput* methods (SetFieldInput, SetMoleculeInput, etc) take the values saved in the *SectionNAMELIST* objects and use them to fill a number *k* != *j* NAME*Input (FieldInput, MoleculeInput, etc)* data, which will be then used to initialize the optimization procedure.
The same Section can contain information needed by different NAME*Input* data, different Sections can contain information needed by one *Input*.





The "*read and set"* procedure ends filing the *NAMEInput* data objects. A different software could exploit the optimal control algorithm filling them directly.

All the objects created and values stored in this "read and set" procedure are <u>temporary.</u> The format of the NAME*Input* objects is the one needed by the permanent part of the code

(the one used in OCManager.py) to initialize. After the initialization, instances of *SectionNAMELIST, SetNAMEInput* and *InputNAME* are destroyed.

The objects where we will permanently store the information during the optimization procedure (*oc_iterator, propagator, field, molecule, etc*) will have them most probably stored in a NAMEParameter object, that is almost identical to the InputNAME object (see example in the following).

This looks like a very repetitive procedure, where the same things are passed and stored many times. The purpose is to completely separate the read and set (i.e. elaborate) part, which need to know the specific format of the input files, and the optimization part, which do not need to know anything about the format and can be initialized by anyone. It is not particularly useful in a full python code, but it would allow to interface OCpy optimization part with anything.

Don't be surprised if you feel you are rewriting many times the same information

Python Files

ReadNamelistOC.py* and *NamelistSections.py

NamelistSections.py contains different classes which inherit from *ABCNamelistSection,* one for each Section in the input.dat file (and other inputs as *genetic_input.dat*)

Each *SectionNAME* (e.g. *SectionSystem, SectionField*, etc) store the keys of the section and their default values, information on which keys values are key sensitive and which are not, which keys have only multiple choices values and witch are free. There are common methods to lower case case unsensitive values and check that only allowed values are selected (i.e. typos).

ReadNamelistOC has the list of the i*nput.dat* Namelists and perform checks on the allowed values of the keys with respect to the other key values within the Section and with respect to values in other sections.

e.g.

possible values for *field_type* key are "pip", "genetic", "constant" ecc
SectionNAME (SectionFiels in this case) methods are able to lower case PiP, PIP, piP ecc and to understand that ppip is an error.
Instead *ReadNamelistOC* methods knows that if *field_type = "constant"* then the value of *omega* key in FIELD section must not be given and that if the optimal control algorithm chosen in the SYSTEM section is "genetic", *fied_type = "constant"* is not an allowed choice.

As OCpy is object oriented we will have to use ReadNamelistOC class in this way:

```
SystemManager():
     oc = OCManager()

     def init_system(folder, namefile):
        user_input =  ReadNamelistOC()
        user_input.read(folder+namefile)
```

Now *user_input* object contains all the information needed to fill the Input objects. As we will not need *user_input* after the “read and set” procedure, this object is temporary and is destroyed after being used.

FieldInput.py, LogInput.py, MediumInput.py, MoleculeInput.py, OCGeneticInput.py, OCInput.py, SaveInput.py
and
SetFieldInput.py, SetLogInput.py, SetMediumInput.py, SetMoleculeInput.py, SetOCGeneticInput.py, SetOCInput.py, SetSaveInput.py

All the *NAMEInput.py* files contains a class with the same name of the file which does not have methods but only attributes, i.e. only stores data. The data stored are the ones needed to initialize and perform the optimization process. Any external software could skip the reading, fill these objects and give them to the *OCManager* (see the following) object which would perform the optimization.

Each of these object has a *SetNAMEInput.py* file and object which receives *user_input = ReadNamelistOC()* instance and processes it to fill the NAME*Input() object*.
E.g. receives the names of the quantum files for the molecule and reads them, filling energies and dipoles in *MoleculeInput()*.
Sometimes the *Set* just takes a value in the *NamelistSection* and copy it in the *Input*, if anything need to be done on that parameter (e.g. dt, nstep, etc…)

Again, OCpy is object oriented, so we will have to create instances for the Set and Input (the Input instance is inside the Set)

```
FieldInput()
    f0
    omega
    (all field parameters, no methods)

SetFieldInput()
    input_parameters = FieldInput()

    def set(..)
        takes a ReadNamelistOC object and fills input_parameters attributes…

SystemManager():
     oc = OCManager()

     def init_system(folder, namefile):
        user_input =  ReadNamelistOC()
        user_input.read(folder+namefile)

        init_field = SetFieldInput()
        init_field.set(user_input)
        … all other sets …
```

ReadOutputQuantumCalc.py

Contains the subroutine to read Gamess output files.
Beware:
read_V() method reads ci_pot.inp files which are printed in gamess in a format different than what is expected and correcr an error on the sign.
#when gamess print will be debugged, read_V_wavet should become the new read_V.
WaveT does not know of the bug so read the ci_pot.inp file
#as it shoudl be. If we want to compare ocpy results with wavet results we have to use read_V_wavet()

What to do if you want to add a keyword in an input Section e.g. Field:

Files to modify: NamelistSections.py, ReadNamelistOc.py, InputNAME.py, SetNAMEInput.py, (NAMEParameters.py)

NamelistSections.py:
In the desired namelist object (e.g. *SectionField*) add in *section_default_dictionary* as attribute the new key and its default value. If the new key has only some allowed values, write them in allowed_val and if it is case unsensitive add the key in the case_unsensitive_keys

ReadNamelistOc.py:
Insert any check needed for the new keyword and insert it in the proper method (e.g. *check_field_nml_consistency*)

InputNAME.py
Identify the NAME*Input* file(s) in which you want to save the value of the new keyword. Can be more than one: e.g. often information on the filed belongs both to *FieldInput.py* and to *LogInput.py*, which is the input which store things that must be saved as header in the output.

Add a variable to store the new keywoord in the Input file(s). Keep nomenclature simple. If you want to add key “gamma” please call the variable “gamma” in all the input files, and everywhere you want to save it. No need to use creative or different names.

SetNAMEInput.py
Add the needed information in *SetNAMEInput* object corresponding to the *NAMEInput* object(s) you want to store the information into.
If you simply need to copy the information in the *InputNAME*, you will have to write inside *SetNAMEInput.set* something like:

```
SetFieldInput()

    def set(user_input)
      input_parameters.key_name=
                  user_input.field.section_dictionary[‘key_name’]
```

and the value will be stored. If the value of the key was needed to do something, you will have to add a more complex function:

e.g.

```
SetMediumInput()

    def set(user_input)
      input_parameters.cavity=

read_output.read_cavity_tessere(user_input.medium.section_dictionary
['name_field_cavity'])
```

Finally, as this information is in the *NAMEInput* object, it means it will be needed by something to be stored, I will need to add the variable somewhere (most probably in a *NAMEParameters.py* file) and modify an object init method which will use the *NAMEInit* file to initialize this new variable.

What to do if you want to add a Section:

Files to modify: NamelistSections.py, ReadNamelistOc.py, InputNAME.py, SetNAMEInput.py, (NAMEParameters.py)

ReadNamelistOc.py:
In addition to what is in the previous section "What to do if you want to add a keyword in an input Section", you need to add the name of the section in ReadNamelistOC() in ReadNamelistOc.py, a check_new_namelist_nml_consistency() method and then all the new keywords as explained earlier.

What to do if you want to add a new Namelist for a specific calculation e.g. NamelistGenetic:

Files to create: ReadNamelistNEWNAMELIST.py
Files to modify: NamelistSections.py, InputNAME.py, SetNAMEInput.py, (NAMEParameters.py)

ReadNamelistNEWNAMELIST.py:
You have to create a new class with the same structure as ReadNamelistOc(), with the names of the different Sections and all the cheks, and then add the NamelistSections and keywords as explained earlier.

OPTIMIZATION

OPTIMIZATION inside OCManager.py

OCManager.py is the only object which needs to be initialized and then runs the optimization.

In principle its initialization method should take as argument all and only FieldInput(), LogInput(), MediumInput(), MoleculeInput(), OCGeneticInput(), OCInput(), SaveInput().

Within the software many other objects are needed to perform different tasks, and all of them are instantiated and initialized inside OCManager().
Within them there are Molecule(), Field() and Medium().
In principle OCManager should instantiate and initialize the three of them:

```
OCManager()
   molecule = Molecule()
   filed = Field()
   medium = Medium()
   … all the other objects …


   def init_oc(MoleculeInput, FieldInput,MediumInput, …)
       mol.init(MoleculeInput)
       field.init(FieldInput)
       medium.init(MediumInput)
```

In practice we prefer to initialize in SystemManager() these three objects as it is easier to keep track of the initial conditions, for developing and debugging reasons, and then the OCManager() initialization is done by copy. External Molecule(), Field() and Medium() istances are then never changed, while the ones inside OCManager() are modified as needed. Can be changed at any time.

As it is now:

```
SystemManager():
     oc = OCManager()
     molecule = Molecule()
     filed = Field()
     medium = Medium()

     def init_system(folder, namefile):
       user_input =  ReadNamelistOC()
       user_input.read(folder+namefile)

       init_field = SetFieldInput()
       init_field.set(user_input)
        … all other sets …

       mol.init(init_molecule.MoleculeInput)
       field.init(init_field.FieldInput)
       medium.init(init_medium.MediumInput)
      oc.init(LogInput,  OCGenetiInput,  OCInput,  SaveInput,  mol,
field, medium)
```

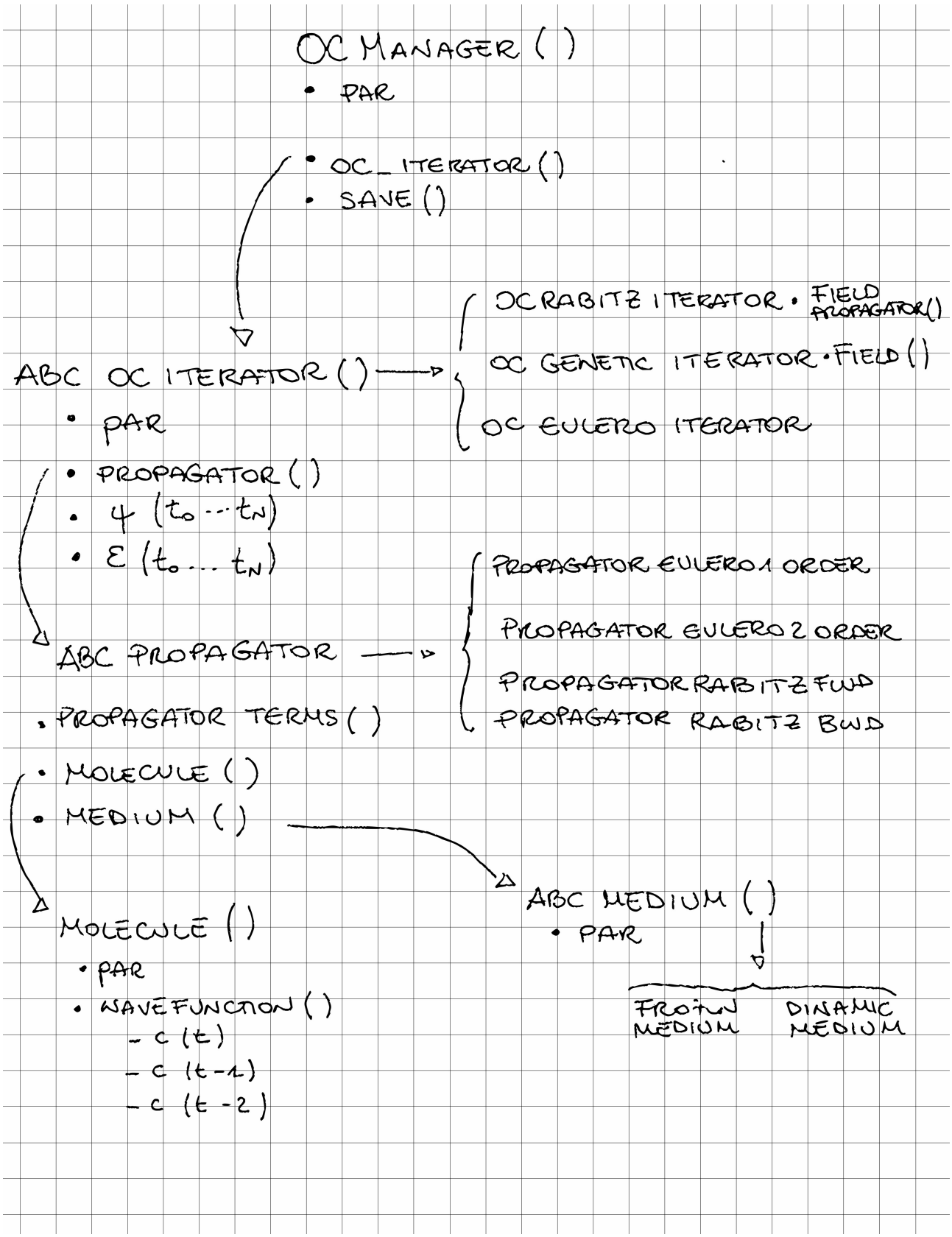
As it should be:

```
SystemManager():
     oc = OCManager()


     def init_system(folder, namefile):
       user_input =  ReadNamelistOC()
       user_input.read(folder+namefile)

       init_field = SetFieldInput()
       init_field.set(user_input)
        … all other sets …
            oc.init(LogInput,  OCGenetiInput,  OCInput,  SaveInput,
MoleculeInput, FieldInput, MediumInput)
```

General Structure



The Figure shows a schematic view of the mail part of the code (without the saving that will be discussed later.

The majority of the objects have the *par* attribute which contains the parameters initialized from the *Input* files, as we already anticipated.

e.g.

```
MoleculeInput()
    muT
    en_ci
    Vijn


MoleculeParameters()
    muT
    en_ci
    Vijn


Molecule()
   par = MoleculeParameters()

   def init(MoleculeParameters)
       par.muT =  MoleculeParameters.muT
       ...
```

As this is an optimal control software, the main action it can accomplish is *iterate() (*method of OCManager()). OCEuleroIterator iterate only once, so practically performs a propagation, which is the simplest case.

Objects

Field()

- creates the field at each time t given the desired shape and parameters. This object is not needed in the OCRabitz and OCEulero, where only a matrix containing the value of the field a different t is needed and modified. Instead is used in OCGenetic to create fieds at each iteration.
- It is also able to read a field from file

Molecule()

- It also contains Wavefunction(), which stores the coefficients during the propagation **at the current time t** and previous times t-1 and t-2.

ABCMedium()
(VacMedium, FrozenSolventMedium, DinamicMedium)

- contains the parameters about the molecule: states, energy values, transition dipoles, potentials.
- contains the parameters about the medium: cavity, charges, kind of medium
- performs the propagation of the charges internally (Frozen) or interfacing with TDPlas (Dinamic)

ABCPropagator()
(Eulero1Order, Eulero2Order, RabitzFwd, RabitzBwd)

- contains Medium() and Molecule()
- contains PropagatorTerms()
- contains a list with the propagator terms (energy, field, medium, normalization…) needed for the specific propagation selected
- sets the list for the specific propagation selected (Eulero1Order, Eulero2Order, RabitzFwd, RabitzBwd)
- propagates

PropagationTerms()

- contains the propagation terms: energy, field, medium… written as
  c_t = H_i * c_t-1
  e. g.

```
eulero_energy_term(mol, order, dt):
   ci += -order * 1j * dt * (en_ci * ci_prev[0])
```

ABCIterator()
(OCRabitz, OCGenetic, OCEulero)

- at the end of each iteration saves matrices containing psi and field
- performs the optimization
- calculate J and convergence
- return quantities to be saved on file to be given to Save()

ABCSave()
(SaveEulero, SaveOCRabitz, SaveOCGenetic)

- creates headers and print them on the save files
- save the desired data on the files

Non obvious programming choices details

ABCPropagator() and PropagatorTerms():

Given the equation:

*c(t) = E*c(t-1) + mu*field(t-1)*c(t-1) + q(t-1)*V*c(t-1) + …*

which gives the t+1 value of the coefficients, we create a different method in PropagatorTerms() which perform the single term of the equation

```
def eulero_energy_term(order, dt):
    ci += -order * 1j * dt * (en_ci * ci_prev[0])

def eulero_field_term(order, dt, field_dt_vector):
    ci += -order * 1j * dt * (-np.dot(np.dot(ci_prev[0], mol.par.muT),
field_dt_vector))
```

Here we define also the Rabitz terms, and every term we may need.
In ABCPropagator we have an empty list *propagator = [ ],* a *set_propagator( )* abstract method and a *propagate_one_step()* abstract method.

When we instantiate a specific child of ABCPropagator (PropagatorEulero2Order, PropagatorEulero1Order, PropagatorOCfwd, PropagatorOCbwd), *set_propagator()* add in the list the specific terms which create the propagator chosen.
Each method in the list takes c(t) already calculated by the previous elements of the list and and sum to it the c(t) just calculated.
So, no matter which methods are in the list, each child of ABCPropagator can run

```
for func in self.propagator:
    func(self.mol, order, dt, field_dt_vector, self.medium)
```

SaveProcedure:

In each child of *ABCOCIterator* there is a dictionary (*out_dict*). Each entry of the dictionary is initialized by *init_output_dictionary()* with methods which return matrices to be saved on file.

e.g.

```
def init_output_dictionary(self):
    dict_out['pop_t'] = get_pop_t

def get_pop_t(self):
    pop_t = np.real(af.population_from_wf_matrix(psi_coeff_t_matrix))
    return pop_t
```

The saving on file is done inside OCManager() by ABCSave and its children.

SaveFile()

Object SaveFile() has only attributes and, as we said, it is what in fortran is called a “type”. We will create an instance of SaveFile for each output file we want to create, and it will correspond to a dictionary entry in *out_dict* inside the children of *ABCOCIterator.*

```
class SaveFile():
    def __init__(self, name, header, save_step, dict_flag,
oc_iterator):
        self.name = name
        self.header = header
        self.save_step = save_step
        self.dict_flag = dict_flag
        self.out = oc_iterator.dict_out[self.dict_flag]
```

when we create an instance of SaveFile() we will provide

- the appendix of the output file name. “_pop_t.dat” means the output file will be called *NAME_pop_t.dat* with NAME given in *input.dat*
- header
- how often I want this file to be saved, which is a parameter given in *input.dat*
- name of the flag in *out_dict* in the *OCIterator* we have initialized
- **the instance** of the OCIterator we have initialized

```
OCManager()
   oc_iterator
   save

   def init_oc(...)
      oc_iterator = OCEuleroIterator()
      oc_iterator.init(...)
      ##inside the init out_dict is initialized and “pop_t” key return
get_pop_t method ###
      save = SaveEulero()
      save.init(..., oc_iterator, ...)
```

in red to show that I am passing the **instance** that will be used to initialize the SaveFiles inside *save.*

Same is done for the *SaveRestart()* object and *get_restart* method inside *OCIterator*

Genetic Iterator

OCGeneticIterator()

The genetic optimization is done exploiting DEAP library. In principle it could do a lot of different optimization strategies in all the steps (optimization, mutation, selection), in practice only one choice is possoble right now. The dictionary at the beginning of OCGeneticIterator.py should allow to easily add more possibilities

The optimization exploit a second input file (i.e. configuration file), which is is *genetic.conf*

Contrary to the other two iterators, here the *Propagator()* objects (which are as many as the replicas) are inside the *chromosomes* array.

Here we define a Deap object called Chromosome(). It works this way:

```
creator.create("Chromosome", list, J=creator.J, field = Field(),
prop_psi = prop.PropagatorEulero2Order())
```

means that when we create an istance myChromosome of Cromosome() (which we have to create through Deap in any case), myChromosome has associated a J value, a Field() object, a PropagatorEulero2Order() **and a list of elements.**

The optimization procedure (.mate, .select, .mutate in Deap library) acts on the list, so at every iteration

- the list values are converted to amplitudes of the Field() object,
- the field is initialized and the PropagatorEulero2Order() propagates the wf with the created field
- the J is calculated
- the best individuals are selected, elements of the list are mated and mutated

Initialization

All the Field() objects in the Chromosomes are initialized copying starting_field = Field() object, which provides the omegas. amplitudes are all equal to the default value 0.01 and are then copied in the list and mutated before starting the first optimization iteration.
For each chromosome the propagator() and the Medium() objects are then initialized.

Right now, the only way of selecting the omegas is calculating the fourier frequencies of the interval and the only method to initialize the frequencies is random. To obtain

reproducibility in the calculations the seed must be fixed, but it is anyway difficult to understand what Deap is doing