D:\proj\UPB\stud\cod_cercetare\t4l.py **3/11/2022 3:42 PM**

```
from switcher import Switcher
from global_setting import *
from LammpsInputFile import LammpsInputFile
from lists import Lists
from periodic import Periodic
from geometry import Geometry

ERROR = -1
SUCCES = 0

def show_menu():
    f = open("show_menu.in", "r")
    print(f.read())


def read_choice():

    sw = Switcher()

    input_script = input("Enter input script:")
    try:
        script_file = open(input_script)
        while True:
            command = script_file.readline().strip()

            if "EXIT" in command:
                print("Exiting...")
                break

            if command:
                output = sw.indirect(command)
                if output == ERROR:
                    print("Error found while applying operation from input script.Exiting...")
                    return

        script_file.close()
    except FileNotFoundError:
        print("Input file provided does not exist")
        return


if __name__ == "__main__":

    # show_menu()
    # read_choice()
    l = LammpsInputFile("333_LA2.DATA")
    # Lists.list_reorder_random(l)
    # l.write_to_file("test.out")

    atom_list = l.get_atoms()
    OTHER_DEFAULTS.box = l.box
```

D:\proj\UPB\stud\cod_cercetare\modules\switcher.py

```
from config.global_setting import *
import random
from classes.LammpsInputFile import *
from .periodic import Periodic
from .memory_management import Memory_handler


ERROR = -1
SUCCES = 0

class Switcher:

    inputfile_handler = None

    def indirect(self, index):

        command = index.split()
        method_name = 'func_'+ str(command[0])
        method = getattr(self, method_name, lambda :'Invalid')

        return method(command)


    def func_GLOBAL_VAR(self, command):


        #Global_var seed iseed[integer] {reinit|other}
        if command[1] == "SEED" :
            if command[2].isnumeric():
                OTHER_DEFAULTS.iseed = int(command[2])
            else:
                print("Bad syntax in command : ")
                print(command)

            if command[3] == "REINIT":
                # TODO :
```

```
            pass
        else:
            print("Bad syntax in command : ")
            print(command)

    #Global_var unit_style  <lj/real/metal/si/cgs/electron/micro/nano>
    if command[1] == "UNIT_STYLE":
        unit_style = command[2]
        if unit_style in ['LJ', 'REAL', 'METAL', 'SI', 'CGS', 'ELECTRON', 'MICRO', 'NANO']:
            OTHER_DEFAULTS.unit_style = unit_style.upper()
            CONSTANTS.unit_style = unit_style.upper()
        else:
            print("INVALID UNIT STYLE in command:")
            print(command)
            return ERROR


    #Global_var atom_style  integer (2 or 5)
    #1=angle 2=atomic     3=body  4=bond    5=charge    6=dipole 7=electron 8=ellipsoid 9=full 10=line
    # 11=meso 12=molecular 13=peri 14=sphere 15=template 16=tri   17=wavepacket"

    if command[1] == "ATOM_STYLE":
        atom_style = command[2]
        if (int(atom_style) < 1) or (int(atom_style) > 17):
            print("Invalid choice")
            return ERROR
        if (int(atom_style) != 2) and (int(atom_style) != 5):
            print("Currently not implemented:")
            print(command)
        OTHER_DEFAULTS.atomstyle = int(atom_style)

    if command[1] == "NMOLECULAR":
        nmolecular = command[2]
        if (int(nmolecular) != 0) and (int(nmolecular) != 1):
                print("NMOLECULAR should be 0 or 1:")
                print(command)
        else:
            OTHER_DEFAULTS.nmolecular = nmolecular


    #Global_var XBox  xlo xhi (real, xlo<xhi)   ! case (21)
    if command[1] == "XBOX":

        if len(command) != 4:
            print("Invalid choice of xdimensions in command")
            print(command)
            return ERROR
        xlo = float(command[2])
        xhi = float(command[3])

        if xlo > xhi:
            print("Xhi must be greater than xlo")
            print(command)
        else:
            OTHER_DEFAULTS.box["xlo xhi"] = (xlo, xhi)
            Periodic.set_prd()


    if command[1] == "YBOX":
        if len(command) != 4:
            print("Invalid choice of xdimensions in command")
            print(command)
            return ERROR
        ylo = float(command[2])
        yhi = float(command[3])

        if ylo > yhi:
            print("Xhi must be greater than xlo")
            print(command)
        else:
            OTHER_DEFAULTS.box["ylo yhi"] = (ylo, yhi)
            Periodic.set_prd()

    if command[1] == "ZBOX":
        if len(command) != 4:
            print("Invalid choice of xdimensions in command")
            print(command)
            return ERROR
        zlo = float(command[2])
        zhi = float(command[3])

        if zlo > zhi:
            print("Xhi must be greater than xlo")
            print(command)
        else:

            OTHER_DEFAULTS.box["zlo zhi"] = (zlo, zhi)
            Periodic.set_prd()

    return SUCCES


def func_READ(self, command):
```

```
        if command[1] == "DATA_FILE":
            if len(command) < 3:
                print("No data file provided")
                return ERROR
            filename = command[2]
            try:
                input_file = open(filename)
                # Already read a data file?
                if (len(command) == 4) and (command[3] == "DELETE"):
                    if OTHER_DEFAULTS.isalloc:
                        # TODO - delete previous saved list of atoms?
                        # call free_atom_memory()
                        Memory_handler.free_atom_memory()
                        self.inputfile_handler = None
                        OTHER_DEFAULTS.isalloc = False
                self.inputfile_handler = LammpsInputFile(filename)
                self.inputfile_handler.read_file()
                OTHER_DEFAULTS.isalloc = True
                # Update box dimensions with the one read from file
                OTHER_DEFAULTS.box = self.inputfile_handler.box
                Periodic.set_prd()

                input_file.close()

            except FileNotFoundError:
                print("Input file provided does not exist")
                return ERROR

        else:
            # Other types currently not implemented
            pass
        return SUCCES

    def func_WRITE(self, command):
        if command[1] == "DATA_FILE":
            if len(command) < 3:
                print("No data file provided")
                return ERROR
            filename = command[2]
            self.inputfile_handler.write_to_file(filename)
        else:
            # Other types currently not implemented
            pass
        return SUCCES
```

D:\proj\UPB\stud\cod_cercetare\config\global_setting.py

```
# parameters unlikely to need changing
class CONSTANTS:
    mxpartic = 1000000
    lstmembuf = 5
    mxlist = mxpartic * lstmembuf
    mxfirst = mxlist + 1
#    constants
    PI = 3.14

    ir_data = 1
    ir_dump = 2
    iw_data = 3
    db_file = "debug.dbg"
    unit_style = "REAL"


# set units
class UNITS_LJ:
    boltz = 1.0
    hplanck = 0.18292026
    mvv2e = 1.0
    ftm2v = 1.0
    mv2d = 1.0
    nktv2p = 1.0
    qqr2e = 1.0
    qe2f = 1.0
    vxmu2f = 1.0
    xxt2kmu = 1.0
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0
    femtosecond = 1.0
    qelectron = 1.0

    dt = 0.005
    skin = 0.3

class UNITS_REAL:
    boltz = 0.0019872067
    hplanck = 95.306976368
    mvv2e = 48.88821291 * 48.88821291
    ftm2v = 1.0 / 48.88821291 / 48.88821291
    mv2d = 1.0 / 0.602214179
    nktv2p = 68568.415
    qqr2e = 332.06371
    qe2f = 23.060549
    vxmu2f = 1.4393264316e4
    xxt2kmu = 0.1
    e_mass = 1.0/1836.1527556560675
    hhmrr2e = 0.0957018663603261
    mvh2r = 1.5339009481951
    angstrom = 1.0
    femtosecond = 1.0
    qelectron = 1.0

    dt = 1.0
    skin = 2.0

class UNITS_METAL:
    boltz = 8.617343e-5
    hplanck = 4.135667403e-3
    mvv2e = 1.0364269e-4
    ftm2v = 1.0 / 1.0364269e-4
    mv2d = 1.0 / 0.602214179
    nktv2p = 1.6021765e6
    qqr2e = 14.399645
    qe2f = 1.0
    vxmu2f = 0.6241509647
    xxt2kmu = 1.0e-4
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0
    femtosecond = 1.0e-3
    qelectron = 1.0

    dt = 0.001
    skin = 2.0

class UNITS_SI:
    boltz = 1.3806504e-23
    hplanck = 6.62606896e-34
    mvv2e = 1.0
    ftm2v = 1.0
    mv2d = 1.0
    nktv2p = 1.0
    qqr2e = 8.9876e9
```

```
    qe2f = 1.0
    vxmu2f = 1.0
    xxt2kmu = 1.0
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0e-10
    femtosecond = 1.0e-15
    qelectron = 1.6021765e-19

    dt = 1.0e-8
    skin = 0.001

class UNITS_CGS:
    boltz = 1.3806504e-16
    hplanck = 6.62606896e-27
    mvv2e = 1.0
    ftm2v = 1.0
    mv2d = 1.0
    nktv2p = 1.0
    qqr2e = 1.0
    qe2f = 1.0
    vxmu2f = 1.0
    xxt2kmu = 1.0
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0e-8
    femtosecond = 1.0e-15
    qelectron = 4.8032044e-10

    dt = 1.0e-8
    skin = 0.1

class UNITS_ELECTRON:
    boltz = 3.16681534e-6
    hplanck = 0.1519829846
    mvv2e = 1.06657236
    ftm2v = 0.937582899
    mv2d = 1.0
    nktv2p = 2.94210108e13
    qqr2e = 1.0
    qe2f = 1.94469051e-10
    vxmu2f = 3.39893149e1
    xxt2kmu = 3.13796367e-2
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.88972612
    femtosecond = 0.0241888428
    qelectron = 1.0

    dt = 0.001
    skin = 2.0

class UNITS_MICRO:
    boltz = 1.3806504e-8
    hplanck = 6.62606896e-13
    mvv2e = 1.0
    ftm2v = 1.0
    mv2d = 1.0
    nktv2p = 1.0
    qqr2e = 8.9876e30
    qe2f = 1.0
    vxmu2f = 1.0
    xxt2kmu = 1.0
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0e-4
    femtosecond = 1.0e-9
    qelectron = 1.6021765e-19

    dt = 2.0
    skin = 0.1

class UNITS_NANO:
    boltz = 0.013806503
    hplanck = 6.62606896e-4
    mvv2e = 1.0
    ftm2v = 1.0
    mv2d = 1.0
    nktv2p = 1.0
    # qqr2e = 8.9876e39   # Error: Real constant overflows
its kind
    qe2f = 1.0
    vxmu2f = 1.0
    xxt2kmu = 1.0
    e_mass = 0.0
    hhmrr2e = 0.0
    mvh2r = 0.0
    angstrom = 1.0e-1
    femtosecond = 1.0e-6
    qelectron = 1.6021765e-19

    dt = 0.00045
    skin = 0.1


class OTHER_DEFAULTS:

    unit_style = "REAL"

# global settings and flags
    idimension = 3
    nsteps = 0
    itime = None
    newton_bond = 1
    # dt = None
    isalloc = False

# domain
    xprd = 0
    yprd = 0
    zprd = 0

    xprd_half = 0
    yprd_half = 0
    zprd_half = 0

    box =  {"xlo xhi" : None,
            "ylo yhi": None,
            "zlo zhi": None
            }

    # Define a cell length of 2 Angstrom
    r_cutoff = 2

    perflagx = 0
    perflagy = 0
    perflagz = 0

# Not implemented -> substitute with Atom/Bond class from
atom.py module ?
# - atoms
# - bonds

    # TODO
    # ATOMS
    npartic = 0
    ntypes = 0
    natomtypes = 0
    rvdw = None #[]
    # mass = None # []
    # x = None # [][]
    # v = None # [][]
    # x_unclean = None # [][]
    # q = None # []
    # tag = None #[]
    itype = None # []
    # molecule = None # []
    # true = [] #[]
    ibox = None #[]
    # atomtype = None #[]
    atypes_total = None #[]

    # bond connectivity for each atom

    numbond = None #(:)
    bondtype = None #(:,:)
    bondatom = None #(:,:)

    r_nbonds = 0
    r_nangles = 0
    r_ndihedrals = 0
    r_nimpropers = 0
    r_nbonds_list = 0
    r_nangles_list = 0
    r_ndihedrals_list = 0
    r_nimpropers_list = 0
    r_bonds = None #(:,:)
    r_angles = None #(:,:)
    r_dihedrals = None #(:,:)
    r_impropers = None #(:,:)
    r_nangletypes = 0
    r_nbondtypes = 0
    r_ndihedraltypes = 0
    r_nimpropertypes = 0
    r_bondtype = None #(:)
    r_angletype = None #(:)
```

```
    r_dihedraltype = None #(:)
    r_impropertype = None #(:)

    bindex_list = None #(:)
    aindex_list = None #(:)
    dindex_list = None #(:)
    iindex_list = None #(:)

    r_natypes_list = 0
    r_nbtypes_list = 0
    r_ndtypes_list = 0
    r_nitypes_list = 0

# temperature creation
    rotationflag = None


# input
    trueflag = None
    ndumpatom = None
    ndumpvel = None
    ndumpforce = None
    atomstyle = None

# diagnostics
    idebug = 0
    idbg = 100
    iEtrans = 101
    iErot = 102
    temptest = 103

# fragments
    # list = None # []
    # first = None # []
    # nlist = 0
    # isrange = None #[]

# velocity analysis
    Emax = None #
    vmax = None
    nvbin = None
    nEbin = None
    vbin = None #[]
    Ebin_size = None
    Ebin_sizemol = None
    bin_size = None
    v_tr = None
    v_rot = None
    v_vib = None
    v_rovib = None
    ddisp = None
    ddisp_v = None
    displacements = None
    d_k = None

# random number global seed

    tgvar = 0
    # TODO random number
    iseed = 0
    seed_global = iseed

# read write things
    nm_nmol = None
    ffoffset = 0
    nm_modes = None
    nm_natpermol = None
    nm_ncols = None
    nm_refxyz = None
    nm_modes = None
    nm_paxes = None
    nm_com = None
    nm_masses = None
    nm_paxsorted = None
# read write things
    vcorr0 = None
    # 3-dimensional
    vcorrt = None
    spect = None
    v0 = None



# others defined in t41
    maxbondper = 8
    nmolecular = 0

# files in t41
    etrans = "Etrans.out"
    erot = "Erot.out"
    temptest = "temptest.out"
```

D:\proj\UPB\stud\cod_cercetare\classes\LammpsInputFile.py

```
import re
import datetime
import sys

from .Angle import Angle
from .Bond import Bond
from .Atom import Atom

class LammpsInputFile():

    STYPES = ["Masses", "Nonbond Coeffs", "Bond Coeffs", "Angle Coeffs", "Dihedral Coeffs",
              "Improper Coeffs", "BondBond Coeffs", "BondAngle Coeffs", "MiddleBondTorsion Coeffs",
              "EndBondTorsion Coeffs", "AngleTorsion Coeffs", "AngleAngleTorsion Coeffs", "BondBond13 Coeffs",
              "AngleAngle Coeffs", "Atoms", "Velocities", "Bonds", "Angles", "Dihedrals", "Impropers"]

    def __init__(self, input_fname = ""):
        self.header = { "atoms" : 0,
                        "bonds" : 0,
                        "angles" : 0,
                        "dihedrals" : 0,
                        "impropers" : 0,
                        "atom types" : 0,
                        "bond types" : 0,
                        "angle types" : 0,
                        "dihedral types" : 0,
                        "improper types" : 0,
                      }
        self.box =  {   "xlo xhi" : None,
                        "ylo yhi": None,
                        "zlo zhi": None
                    }

        self.header_keys = self.header.keys()
        self.input_fname = input_fname
        self.output_fname = input_fname.strip() + ".out"
        self.input_file = None
        self.lines = []
        self.current_index = -1

        """!!!Order is important
```

```
          TODO - incomplete list
    """
    self.entries = {
        "Masses"    : [],
        "Atoms"     : [],
        #"Velocities": [],
        "Bonds"     : [],
        "Angles"    : []
    }

    """parse the file"""
    self.read_file()

def get_header(self):
    return self.header

def get_atoms(self):
    return self.entries["Atoms"]

def set_atoms(self, l = []):
    self.entries["Atoms"] = l

def get_masses(self):
    return self.entries["Masses"]

def set_masses(self, l = []):
    self.entries["Masses"] = l

def get_velocities(self):
    return self.entries["Velocities"]

def set_velocities(self, l = []):
    self.entries["Velocities"] = l

def get_bonds(self):
    return self.entries["Bonds"]

def set_bonds(self, l = []):
    self.entries["Bonds"] = l

def get_angles(self):
    return self.entries["Angles"]

def set_angles(self, l = []):
    self.entries["Angles"] = l

def parse_nonbond_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_bond_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_angle_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_dihedral_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_improper_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_bondbond_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_bondangle_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_middlebondtorsion_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_endbondtorsion_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass
```

```
def parse_angletorsion_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_angleangletorsion_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_bondbond13_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_angleangle_coeffs(self):
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_dihedrals(self):
    print("Parsing Dihedrals starting at line #{}".format(self.current_index))
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_impropers(self):
    print("Parsing Impropers starting at line #{}".format(self.current_index))
    self.current_index = len(self.lines)
    print("Not implemented!")
    pass

def parse_masses(self):
    N = self.header["atom types"]
    print("Parsing {} Masses starting at line #{}".format(N, self.current_index))

    start = self.current_index + 1
    end = start + N
    for l in self.lines[start:end]:
        tokens = l.split()
        #self.entries["Masses"][int(tokens[0])] = float(tokens[1])
        self.entries["Masses"].append((int(tokens[0]), float(tokens[1])))

    self.current_index = end

def parse_atoms(self):
    N = self.header["atoms"]
    print("Parsing {} Atoms starting at line #{}".format(N, self.current_index))

    start = self.current_index + 1
    end = start + N
    for l in self.lines[start:end]:
        items = l.split()
        if len(items) >= 7:
            idx = int(items[0])
            tag = int(items[1])
            type = int(items[2])
            q = float(items[3])
            x = float(items[4])
            y = float(items[5])
            z = float(items[6])
        # else:
        #     idx = int(items[0])
        #     tag = 1
        #     type = int(items[1])
        #     q = float(items[2])
        #     x = float(items[3])
        #     y = float(items[4])
        #     z = float(items[5])
        # """it means we also have nx ny nz"""
        if len(items) == 10:
            nx = float(items[7])
            ny = float(items[8])
            nz = float(items[9])
            # atom = (idx, tag, type, q, x, y, z, nx, ny, nz)
            atom = Atom(idx, tag, type, q, x, y, z, nx, ny, nz)
        else:
            # atom = (idx, tag, type, q, x, y, z, 0, 0, 0)
            atom = Atom(idx, tag, type, q, x, y, z, 0, 0, 0)
        self.entries["Atoms"].append(atom)

    self.current_index = end

# Added velocity as atom property
def parse_velocities(self):
    N = self.header["atoms"]
    print("Parsing {} Velocities starting at line #{}".format(N, self.current_index))
    start = self.current_index + 1
    end = start + N
```

```
    for l in self.lines[start:end]:
        items = l.split()
        idx = int(items[0])
        vx = float(items[1])
        vy = float(items[2])
        vz = float(items[3])
        # vel = (idx, vx, vy, vz)
        # self.entries["Velocities"].append(vel)
        self.entries["Atoms"][idx - 1].set_velocities((vx, vy, vz))
    self.current_index = end

def parse_bonds(self):
    N = self.header["bonds"]
    print("Parsing {} Bonds starting at line #{}".format(N, self.current_index))
    start = self.current_index + 1
    end = start + N
    for l in self.lines[start:end]:
        items = l.split()
        idx = int(items[0])
        bond_type = int(items[1])
        atom1 = int(items[2])
        atom2 = int(items[3])
        # bond = (idx, bond_type, atom1, atom2)
        # self.entries["Bonds"].append(bond)
        bond = Bond(idx, bond_type, atom1, atom2)
        self.entries["Bonds"].append(bond)

    self.current_index = end

def parse_angles(self):
    N = self.header["angles"]
    print("Parsing {} Angles starting at line #{}".format(N, self.current_index))
    start = self.current_index + 1
    end = start + N
    for l in self.lines[start:end]:
        items = l.split()
        idx = int(items[0])
        angle_type = int(items[1])
        atom1 = int(items[2])
        atom2 = int(items[3])
        atom3 = int(items[4])
        # angle = (idx, angle_type, atom1, atom2, atom3)
        # self.entries["Angles"].append(angle)
        angle = Angle(int(items[0]), int(items[1]), int(items[2]), int(items[3]), int(items[4]))
        self.entries["Angles"].append(angle)

    self.current_index = end

def parse_header(self):
    for i in range(len(self.lines)):
        sline = self.lines[i].strip()
        goto_next_line = False

        """empty line"""
        if sline == "":
            continue
        """comment"""
        if sline.startswith("#"):
            continue

        for key in self.header_keys:
            if sline.endswith(key):
                token = sline.strip("\t {}".format(key))
                self.header[key] = int(token)
                goto_next_line = True
                break

        if goto_next_line:
            continue

        if sline.endswith("xlo xhi"):
            tokens = sline.strip(" xlo xhi").split(" ")
            tokens = [i for i in tokens if i != ""]
            self.box["xlo xhi"] = (float(tokens[0]), float(tokens[1]))
        elif sline.endswith("ylo yhi"):
            tokens = sline.strip(" ylo yhi").split(" ")
            tokens = [i for i in tokens if i != ""]
            self.box["ylo yhi"] = (float(tokens[0]), float(tokens[1]))
        elif sline.endswith("zlo zhi"):
            tokens = sline.strip(" zlo zhi").split(" ")
            tokens = [i for i in tokens if i != ""]
            self.box["zlo zhi"] = (float(tokens[0]), float(tokens[1]))
        else:
            """It means we got out of header"""
            self.current_index = i
            return

"""Returns the line index for the last data point for this type of coeff
    or -1 if type is unknown. If -1 is returned, ignore until next found type"""
def decide_what_comes_next(self):
```

```
    if self.current_index >= len(self.lines):
        return

    coeff_type = self.lines[self.current_index].split("#")[0].strip()
    if coeff_type not in LammpsInputFile.STYPES:
        print ("Something is wrong in the input file. Unknown type: {}".format(coeff_type))
        return

    method_name = "parse_" + coeff_type.lower().replace(" ", "_")
    method_to_call = getattr(self, method_name)
    method_to_call()
    self.decide_what_comes_next()

def read_file(self):
    self.input_file = open(self.input_fname)
    """first two lines are ignored"""
    self.lines = self.input_file.readlines()[2:]
    self.lines = [re.sub(' +', ' ', i.strip()) for i in self.lines if i.strip() != ""]
    """parse the header"""
    self.parse_header()

    """See what is coming next in file"""
    self.decide_what_comes_next()

    print("Reading file: Done")


# def dict_to_list(self, dict):
#     l = []
#     for key, value in dict.items():
#         l.append((int(key), float(value)))
#     return l

# def class_dict_to_list(self, dict):
#     l = []
#     for key, value in dict.items():
#         l.append((key, value))
#     return l

def write_to_file(self, output_file_name):
    if output_file_name is None:
        output_file = open(self.output_fname, "w+")
    else:
        output_file = open(output_file_name, "w+")

    out_string = ""
    """First two lines are ignored"""
    #output_file.write("LAMMPS Description ({})\n\n".format(datetime.datetime.now()))
    out_string += "LAMMPS Description ({})\n\n".format(datetime.datetime.now())
    empty_line = True

    for k in self.header:
        if k.endswith("types") and empty_line:
            #output_file.write("\n")
            out_string += "\n"
            empty_line = False

        if self.header[k] != 0:
            #output_file.write("{} {}\n".format(self.header[k], k))
            out_string += "{} {}\n".format(self.header[k], k)

    """Simulation box"""
    #output_file.write("\n")
    out_string += "\n"
    for k in self.box:
        #output_file.write("{} {} {}\n".format(self.box[k][0], self.box[k][0], k))
        out_string += "{} {} {}\n".format(self.box[k][0], self.box[k][0], k)
    #output_file.write("\n")
    out_string += "\n"
    output_file.write(out_string)

    for entry in self.entries:
        elist = self.entries[entry]

        if elist:
            # output_file.write("{}\n\n".format(entry))
            if entry == "Masses":
                masses_string = "{}\n\n".format(entry)
                for item in elist:
                    masses_string += " {}\t{:.12f}\n".format(item[0], item[1]).expandtabs()
                output_file.write(masses_string + "\n")

            elif entry == "Atoms":
                atoms_string = "{}\n\n".format(entry)
                velocities_string = "{}\n\n".format("Velocities")
                for item in elist:
                    atoms_string += item.print_atom()
                    velocities_string += item.print_velocity()
                output_file.write(atoms_string + "\n")
                output_file.write(velocities_string + "\n")
```

```
                elif entry == "Bonds":
                    bonds_string = "{}\n\n".format(entry)
                    for item in elist:
                        bonds_string += item.print_bond()
                    output_file.write(bonds_string + "\n")
                elif entry == "Angles":
                    angles_string = "{}\n\n".format(entry)
                    for item in elist:
                        angles_string += item.print_angle()
                    output_file.write(angles_string + "\n")

        output_file.close()

    def __del__(self):
        self.input_file.close()
        print("All cleared!")

    def __str__(self):
    res = "Header: " + str(self.header) + "\n"
    res += "Box: " + str(self.box) + "\n"
        if self.entries["Masses"]:
    res += "Masses: " + str(self.entries["Masses"]) + "\n"
        if self.entries["Atoms"]:
    res += "Atoms: [" + str(self.entries["Atoms"][0]) + "\n\t\t. . .\n\t" + str(self.entries["Atoms"][-1]) + "]\n"
        if self.entries["Velocities"]:
    res += "Velocities: [" + str(self.entries["Velocities"][0]) + "\n\t\t. . .\n\t" + str(self.entries["Velocities"][-1]) + "]\n"
        if self.entries["Bonds"]:
    res += "Bonds: [" + str(self.entries["Bonds"][0]) + "\n\t\t. . .\n\t" + str(self.entries["Bonds"][-1]) + "]\n"
        if self.entries["Angles"]:
    res += "Angles: [" + str(self.entries["Angles"][0]) + "\n\t\t. . .\n\t" + str(self.entries["Angles"][-1]) + "]\n"
        return res
```

D:\proj\UPB\stud\cod_cercetare\modules\lists.py

```
import random
import numpy as np
import math
from hilbert import decode, encode
from config.global_setting import OTHER_DEFAULTS
from classes.LammpsInputFile import *

class Lists:

    @staticmethod
    def list_reorder_random(lif_object):
        atoms_number = lif_object.get_header()["atoms"]
        l = np.arange(1, atoms_number+1, 1)
        random.shuffle(l)

        atoms_list = lif_object.get_atoms()

        for atom in atoms_list:
            atom.set_new_id(l[atoms_list.index(atom)])
        atoms_list.sort(key=lambda atom: atom.get_new_id())

    # take region as
    # region =  {
    # "xlo xhi" : None,
    # "ylo yhi": None,
    # "zlo zhi": None
    # }
    @staticmethod
    def divide_cells(atom_list, region, box):

        atoms_indexed = None

        if (region["xlo xhi"][0] > region["xlo xhi"][1]) or
           (region["ylo yhi"][0] > region["ylo yhi"][1]) or (region["zlo zhi"][0] > region["zlo zhi"][1]):
            print("ERROR x|y|z hi must be greater than x|y|z lo")
            return atoms_indexed

        if (not all(box["xlo xhi"][0] <= a <= box["xlo xhi"][1] for a in [region["xlo xhi"][0],region["xlo xhi"][1]])) or
           (not all(box["ylo yhi"][0] <= a <= box["ylo yhi"][1] for a in [region["ylo yhi"][0],region["ylo yhi"][1]])) or
           (not all(box["zlo zhi"][0] <= a <= box["zlo zhi"][1] for a in [region["zlo zhi"][0],region["zlo zhi"][1]])):
            print("Region for x not in box")
            return atoms_indexed

        atoms_indexed = {}

        nx = math.ceil(abs(region["xlo xhi"][1] - region["xlo xhi"][0]))
        ny = math.ceil(abs(region["ylo yhi"][1] - region["ylo yhi"][0]))
        nz = math.ceil(abs(region["zlo zhi"][1] - region["zlo zhi"][0]))

        keys_list = []
        for atom in atom_list:
            dx = (atom.x - region["xlo xhi"][0]) / OTHER_DEFAULTS.r_cutoff
            dy = (atom.y - region["ylo yhi"][0]) / OTHER_DEFAULTS.r_cutoff
            dz = (atom.z - region["zlo zhi"][0]) / OTHER_DEFAULTS.r_cutoff

            if ((nx > dx >= 0) and (ny > dy >= 0) and (nz > dz >= 0)):
```

```
                # Cell position
                ix = int(dx)
                iy = int(dy)
                iz = int(dz)
                #ilist = iz + nz * iy + nz * ny * ix
                ilist = ix + nx * iy + nx * ny * iz
                keys_list.append(ilist)
                if ilist in atoms_indexed:
                    atoms_indexed[ilist].append(atom)
                else:
                    atoms_indexed[ilist] = [atom]


        # print(len(keys_list))
        # print(max(keys_list))
        return atoms_indexed

    @staticmethod
    def get_hilbert_curve_crossing(lammps_input):

        region = lammps_input.box
        nx = math.ceil(abs(region["xlo xhi"][1] - region["xlo xhi"][0]))
        ny = math.ceil(abs(region["ylo yhi"][1] - region["ylo yhi"][0]))
        nz = math.ceil(abs(region["zlo zhi"][1] - region["zlo zhi"][0]))

        num_cels = nx*ny*nz

        # print("nx = " + str(nx))
        # print("ny = " + str(ny))
        # print("nz = " + str(nz))
        print(num_cels)

        maximum_cells = max(nx, ny, nz)

        print(maximum_cells)

        bit_resolution = 0
        power_of_2 = 1
        while (power_of_2 < maximum_cells):
            power_of_2 = power_of_2 << 1
            bit_resolution += 1

        print(power_of_2)
        print(bit_resolution)
        cells = np.arange(0, num_cels, 1)
        print(cells)
        print(np.array(cells))
        hilbert_curve = decode(np.array(cells), 3, bit_resolution)
        print(hilbert_curve)

        crossing_order = [e[0] + nx * e[1] + nx * ny * e[2] for e in hilbert_curve]

        # print(crossing_order)
        print(len(crossing_order))
        print(max(crossing_order))
```

D:\proj\UPB\stud\cod_cercetare\modules\periodic.py

```
from config.global_setting import OTHER_DEFAULTS


class Periodic:


    # Remap the point (xx,yy,zz) into the periodic box,
    # no matter how far away it is. Adjust true flag accordingly.
    @staticmethod
    def remap(xx, yy, zz, itrue):

        if OTHER_DEFAULTS.perflagx  == 0:
            while xx < OTHER_DEFAULTS.box["xlo xhi"][0] :
                xx = xx + OTHER_DEFAULTS.xprd
                itrue = itrue - 1
            while xx >= OTHER_DEFAULTS.box["xlo xhi"][1] :
                xx = xx - OTHER_DEFAULTS.xprd
                itrue = itrue + 1

        if OTHER_DEFAULTS.perflagy == 0:
            while yy < OTHER_DEFAULTS.box["ylo yhi"][0] :
                yy = yy + yy + OTHER_DEFAULTS.yprd
                itrue = itrue - 1000
            while yy >= OTHER_DEFAULTS.box["ylo yhi"][1] :
                yy = yy - OTHER_DEFAULTS.yprd
                itrue = itrue + 1000

        if OTHER_DEFAULTS.perflagz == 0:
            while zz < OTHER_DEFAULTS.box["zlo zhi"][0] :
                zz = zz + OTHER_DEFAULTS.zprd
```

```
            itrue = itrue - 1000000
        while zz >= OTHER_DEFAULTS.box["zlo zhi"][1] :
            zz = zz - OTHER_DEFAULTS.zprd
            itrue = itrue + 1000000

    return xx, yy, zz, itrue


#  enforce PBC on appropriate dims, no matter which box image the particles are in
@staticmethod
def pbc(atom_list):

    for atom in atom_list:
        atom.x, atom.y, atom.z, atom.true =  Periodic.remap(atom.x, atom.y, atom.z, atom.true)

    print(atom.x, atom.y, atom.z, atom.true)


@staticmethod
def minimg(dx, dy, dz):

    if OTHER_DEFAULTS.perflagx == 0:
        if abs(dx) > OTHER_DEFAULTS.xprd_half:
            if dx < 0.0:
                dx = dx + OTHER_DEFAULTS.xprd
            else:
                dx = dx - OTHER_DEFAULTS.xprd

    if OTHER_DEFAULTS.perflagy == 0:
        if abs(dy) > OTHER_DEFAULTS.yprd_half:
            if dy < 0.0:
                dy = dy + OTHER_DEFAULTS.yprd
            else:
                dy = dy - OTHER_DEFAULTS.yprd

    if OTHER_DEFAULTS.perflagz == 0:
        if abs(dz) > OTHER_DEFAULTS.zprd_half:
            if dz < 0.0:
                dz = dz + OTHER_DEFAULTS.zprd
            else:
                dz = dz -OTHER_DEFAULTS.zprd

    return dx, dy, dz

# Returns the image indices ix,iy,iz of the box, according to the true-flag
@staticmethod
def get_image_index2(iat):
    ix = 0
    iy = 0
    iz = 0
    return ix, iy, iz
@staticmethod
def get_image_index(itrue):

    d = itrue
    rem = d % 1000
    ix = rem - 500
    d = d / 1000
    rem = d % 1000
    iy = rem - 500
    d = d / 1000
    rem = d % 1000
    iz =  rem - 500
    return ix, iy, iz

# Returns the true-flag, according to the image indices ix,iy,iz of the box
@staticmethod
def get_itrue(ix, iy, iz):

    itrue = (500 + iz) * 1000000 + + (500 + iy) * 1000 + (500 + ix)
    return itrue

# Sets *prd and *prd_half - call every time the box size changes
@staticmethod
def set_prd():

    OTHER_DEFAULTS.xprd = OTHER_DEFAULTS.box["xlo xhi"][1] - OTHER_DEFAULTS.box["xlo xhi"][0]
    OTHER_DEFAULTS.xprd_half = OTHER_DEFAULTS.xprd * 0.5
    OTHER_DEFAULTS.yprd = OTHER_DEFAULTS.box["ylo yhi"][1] - OTHER_DEFAULTS.box["ylo yhi"][0]
    OTHER_DEFAULTS.yprd_half = OTHER_DEFAULTS.yprd * 0.5
    OTHER_DEFAULTS.zprd = OTHER_DEFAULTS.box["zlo zhi"][1] - OTHER_DEFAULTS.box["zlo zhi"][0]
    OTHER_DEFAULTS.zprd_half = OTHER_DEFAULTS.zprd * 0.5

    @staticmethod
    def clean_edges():
        pass

    @staticmethod
    def x_unclean_edges():
        pass
```

D:\proj\UPB\stud\cod_cercetare\modules\geometry.py

```
from config.global_setting import OTHER_DEFAULTS
from classes.Atom import Atom
from math import sqrt

class Geometry:

    @staticmethod
    def translate(atom_list, disp):
        for atom in atom_list:
            x = atom.x + disp[0]
            y = atom.y + disp[1]
            z = atom.z + disp[2]
            # velocity = atom.get_velocities()
            # new_vel = tuple(item1 + item2 for item1, item2 in zip(velocity, disp))
            # atom.set_velocities(new_vel)
            atom.x = x;
            atom.y = y;
            atom.z = z;

        return atom_list

    @staticmethod
    def find_molecules(atom_list, bonds_list):
        nats = len(atom_list)
        mol = 0

        #assign molecule id 0 to all atoms
        for atom in atom_list:
            atom.molecule_tag = 0
        stack = []

        #dfs
        for atom in atom_list:
            if atom.molecule_tag == 0:
                mol = mol + 1
                atom.molecule_tag = mol
                # nstack =1
                stack.append(atom)
                while (len(stack) != 0):
                    atom_j = stack.pop()

                    # nstack = nstack - 1
                    for bond in bonds_list:
                        atoms = bond.get_atoms_id()
                        if atom_j.id in atoms:
                            if atoms.index(atom_j.id) == 0:
                                atom_k_id = atoms[1]
                            else:
                                atom_k_id = atoms[0]
                            if atom_list[atom_k_id - 1].molecule_tag == 0:
                                atom_list[atom_k_id - 1].molecule_tag = mol
                                # nstack = nstack + 1
                                stack.append(atom_list[atom_k_id - 1])


        # return atom_list


    @staticmethod
    def displace_atoms(atom_list, ts):

        for atom in atom_list:
            atom.x = atom.x + atom.get_velocities()[0] * ts
            atom.y = atom.y + atom.get_velocities()[1] * ts
            atom.z = atom.z + atom.get_velocities()[2] * ts

    # return atom_list

    @staticmethod
    def get_atoms_within_sphere(atom_list, point, radius, result_atom_list):

        for atom in atom_list:
            dx = point[0] - atom.x
            dy = point[1] - atom.y
            dz = point[2] - atom.z
            d = sqrt(dx **2 + dy **2 + dz **2)
            if d < radius:
                result_atom_list.append(atom)



    # @staticmethod
    # def make_supercell(atom_list, cell_thick, idimension, npartic):

    #     nats = len(atom_list)
    #     if nats == 0:
    #         print("Empty atom list...Read data file first")
```

```
    #         return

    #     if cell_thick < 1:
    #         print("Invalid choice! [cell thick < 1]")
    #         return
    #     # Total number of cells
    #     ncell1d = cell_thick * 2 + 1
    #     ncells = ncell1d ** idimension

    #     # Skip timeshift options - no dump file handling for now

    #     npartic_new = npartic + nats * ncells
```

D:\proj\UPB\stud\cod_cercetare\modules\kinetics.py

```
from config.global_setting import *
from config.global_setting import *
from classes.Atom import Atom
from classes.LammpsInputFile import *
from math import sqrt

class Kinetics:

    # HELPER function
    #  method used to set up default values depending on the unit style
    @staticmethod
    def determine_default_unit_values(unit_style):

        if OTHER_DEFAULTS.unit_style == 'REAL':
           return UNITS_REAL()
        elif OTHER_DEFAULTS.unit_style == 'LJ':
            return UNITS_LJ()
        elif OTHER_DEFAULTS.unit_style == 'METAL':
           return UNITS_METAL()
        elif OTHER_DEFAULTS.unit_style == 'SI':
            return UNITS_SI()
        elif OTHER_DEFAULTS.unit_style == 'CGS':
            return UNITS_CGS()
        elif OTHER_DEFAULTS.unit_style == 'ELECTRON':
            return UNITS_ELECTRON()
        elif OTHER_DEFAULTS.unit_style == 'MICRO':
            return UNITS_MICRO()
        elif OTHER_DEFAULTS.unit_style == 'NANO':
            return UNITS_NANO()

    @staticmethod
    def calc_mc_pos(atom_list, lif_object, point):
        masses = lif_object.get_masses()
        xcm = 0
        ycm = 0
        zcm = 0
        wgrp = 0
        if len(atom_list) == 0:
            return
        for atom in atom_list:
            res =  [item for item in masses if item[0] == atom.atom_type]
            wi = res[0][1]
            xcm = xcm + wi * atom.x
            ycm = ycm + wi * atom.y
            zcm = zcm + wi * atom.z
            wgrp = wgrp + wi

        if (wgrp == 0):
            print('Wrgp is zero, division by zero not possible')
            return
        point.append(xcm / wgrp)
        point.append(ycm / wgrp)
        point.append(zcm / wgrp)

        print("Mass center of the list is : " + str(point[0]) + " " + str(point[1]) + " " + str(point[2]))

    # Mass center for given set of coordinates
    @staticmethod
    def calc_mc_pos2(np, xyz, masses, point):
        xcm = 0
        ycm = 0
        zcm = 0
        wgrp = 0
        if(np == 0):
            return
        for i in range(0, np):
            wi = masses[i]
            xcm = xcm + wi * xyz[i][0]
            ycm = ycm + wi * xyz[i][1]
            zcm = zcm + wi * xyz[i][2]
            wgrp = wgrp + wi
            if (wgrp == 0):
                print('Wrgp is zero, division by zero not possible')
                return
```

```
        point.append(xcm / wgrp)
        point.append(ycm / wgrp)
        point.append(zcm / wgrp)

        print("Mass center is: " + str(point[0]) + " " + str(point[1]) + " " + str(point[2]))


    # Calculate linear momentum of a set of aprticles
    @staticmethod
    def calc_lin_mom(atom_list, lif_object):

        px = 0
        py = 0
        pz = 0

        if len(atom_list) == 0:
            return

        masses = lif_object.get_masses()

        for atom in atom_list:
            res =  [item for item in masses if item[0] == atom.atom_type]
            wi = res[0][1]
            px = px + wi * atom.get_velocities()[0]
            py = py + wi * atom.get_velocities()[1]
            pz = pz + wi * atom.get_velocities()[2]

        print(" Px, Py, Pz are: " + str(px) + " " + str(py) + " " + str(pz))
        return (px, py, pz)

    @staticmethod
    def calc_mc_vel(atom_list, lif_object):

        if len(atom_list) == 0:
            return

        masses = lif_object.get_masses()
        wgrp = 0
        for atom in atom_list:
            wgrp = wgrp + [item for item in masses if item[0] == atom.atom_type][0][1]

        if wgrp <= (1e-10):
            return

        (px, py, pz) = Kinetics.calc_lin_mom(atom_list, lif_object)

        vxcm = px / wgrp
        vycm = py / wgrp
        vzcm = pz / wgrp

        print("Vxcm, Vyxm, Vzcm are: " + str(vxcm) + " " + str(vycm) + " " + str(vzcm))
        return (vxcm, vycm, vzcm)

    @staticmethod
    def calc_ang_mom(atom_list, lif_object, point):
        angm = [0, 0, 0]

        if len(atom_list) == 0:
            return
        masses = lif_object.get_masses()
        for atom in atom_list:
            res =  [item for item in masses if item[0] == atom.atom_type]
            wi = res[0][1]
            xi = atom.x - point[0]
            yi = atom.y - point[1]
            zi = atom.z - point[2]
            vels = atom.get_velocities()

            angm[0] = angm[0] + wi * ( yi * vels[2] - zi * vels[1])
            angm[1] = angm[1] + wi * ( zi * vels[0] - xi *vels[2])
            angm[2] = angm[2] + wi * ( xi * vels[1] - yi * vels[0])

        print("Angm0, Angm1, Angm2 are: " + str(angm[0]) + " " + str(angm[1]) + " " + str(angm[2]))
        return angm

    # calculate the inertia tensor of a set of particles, relative to a point (point = [p1, p2, p3])
    @staticmethod
    def calc_inertia_mom(atom_list, lif_object, point):

        eps = 0.000001
        imom = [[0,0,0],[0,0,0],[0,0,0]]
        size_atom_list = len(atom_list)

        if  size_atom_list == 0:
            return
        xx = 0
        xy = 0
        xz = 0
        yy = 0
        yz = 0
```

```
    zz = 0

    masses = lif_object.get_masses()
    for atom in atom_list:
        xi = atom.x - point[0]
        yi = atom.y - point[1]
        zi = atom.z - point[2]
        res =  [item for item in masses if item[0] == atom.atom_type]
        wi = res[0][1]
        xx = xx + wi * xi * xi
        xy = xy + wi * xi * yi
        xz = xz + wi * xi * zi
        yy = yy + wi * yi * yi
        yz = yz + wi * yi * zi
        zz = zz + wi * zi * zi

        imom[0][0] = yy + zz
        imom[0][1] = -xy
        imom[0][2] = -xz
        imom[1][0] = -xy
        imom[1][1] = xx + zz
        imom[1][2] = -yz
        imom[2][0] = -xz
        imom[2][1] = -yz
        imom[2][2] = xx + yy

        # Avoid division by zero
        if size_atom_list <= 2:
            imom[0][0] = imom[0][0] + eps
            imom[1][1] = imom[1][1] + eps
            imom[2][2] = imom[2][2] + eps

        print('Inertia momentum is:')
        print(imom)
        return imom

# compute current temperature
@staticmethod
def calc_temperature(atom_list, lif_object, frozendof):

    temp = 0
    size_atom_list = len(atom_list)
    mvv2e = 0
    boltz = 0

    if  size_atom_list == 0:
        return

    masses = lif_object.get_masses()
    for atom in atom_list:
        vel = atom.get_velocities()
        atom_type_mass =  [item for item in masses if item[0] == atom.atom_type][0][1]
        temp = temp + (vel[0] * vel[0] + vel[1] * vel[1] + vel[2] * vel[2]) * atom_type_mass


    Unit_style = Kinetics.determine_default_unit_values(OTHER_DEFAULTS.unit_style)
    mvv2e = Unit_style.mvv2e
    boltz = Unit_style.boltz
    ndof = OTHER_DEFAULTS.idimension * size_atom_list - frozendof
    temp = temp * mvv2e / (ndof * boltz)

    print('Temperature of fragment:')
    print(temp)
    return temp


# rescale temperature to target temp T0
@staticmethod
def temp_rescale(atom_list, lif_object, T0, Temp):

    factor = sqrt(T0/ Temp)
    size_atom_list = len(atom_list)

    if size_atom_list == 0:
        return

    for atom in atom_list:
        vels = atom.get_velocities()
        atom.set_velocities((vels[0] *factor, vels[1] * factor,  vels[2] * factor))

    temp = Kinetics.calc_temperature(atom_list, lif_object, 6)
    print(temp)
    return temp



# compute temperature contributions from trans,vib,rotation
# vt, vv, vr = [[vx, vy, vz]...]
@staticmethod
def calc_tvr_temperature(atom_list, lif_object, nat, vt, vv, vr):
```

```
        j = 0
        Tt = 0
        Tr = 0
        Tv = 0

        masses = lif_object.get_masses()
        for atom in atom_list:
            mass =  [item for item in masses if item[0] == atom.atom_type][0][1]
            vt1 = vt[j]
            vr1 = vr[j]
            vv1 = vv[j]
            Tt = Tt + (vt1[0] * vt1[0] + vt1[1] * vt1[1] + vt1[2] * vt1[2]) * mass
            Tr = Tr + (vr1[0] * vr1[0] + vr1[1] * vr1[1] + vr1[2] * vr1[2]) * mass
            Tv = Tv + (vv1[0] * vv1[0] + vv1[1] * vv1[1] + vv1[2] * vv1[2]) * mass
            j = j + 1

            ndof = OTHER_DEFAULTS.idimension * nat

            unit_style_class = Kinetics.determine_default_unit_values(OTHER_DEFAULTS.unit_style)
            mvv2e = unit_style_class.mvv2e
            boltz = unit_style_class.boltz

            Tt = Tt * mvv2e / ( ndof * boltz)
            Tr = Tr * mvv2e / ( ndof * boltz)
            Tv = Tv * mvv2e / ( ndof * boltz)
```

D:\proj\UPB\stud\cod_cercetare\classes\Atom.py

```
class Atom:
    id = None
    new_id = None
    molecule_tag = None
    atom_type = None
    q = None
    x = None
    y = None
    z = None
    nx = None
    ny = None
    nz = None
    #  True flag for each atom to replace true from OTHER DEFAULTS
    true = None

    # Velocities Vx    Vy    Vz
    velocity = (None, None, None)

    def __init__(self, id, molecule_tag, atom_type, q, x, y, z, nx, ny, nz):
        self.id = id
        self.molecule_tag = molecule_tag
        self.atom_type = atom_type
        self.q = q
        self.x = x
        self.y = y
        self.z = z
        self.nx = nx
        self.ny = ny
        self.nz = nz
        # True flag for each atom
        self.true = 500500500

    def get_velocities(self):
        return self.velocity

    def set_velocities(self, velocity):
        self.velocity = velocity

    def set_new_id(self, new_id):
        self.new_id = new_id

    def get_new_id(self):
        return self.new_id

    def print_atom(self):
        str = "\t{}\t{}\t{}\t{:.18f}\t{:.18f}\t{:.18f}\t{:.18f}\t{}\t{}\t{}\n".format(self.id,  self.molecule_tag,
self.atom_type, self.q, self.x, self.y, self.z, self.nx, self.ny, self.nz).expandtabs()
        return str

    def print_velocity(self):
        str = "\t{}\t{:.18f}\t{:.18f}\t{:.18f}\n".format(self.id, self.velocity[0], self.velocity[1],
self.velocity[2]).expandtabs()
        return str
```

D:\proj\UPB\stud\cod_cercetare\classes\Bond.py

```
class Bond:
    id = None
    bond_type = None
```

```
    atom_id1 = None
    atom_id2 = None

    def __init__(self, id, bond_type, atom_id1, atom_id2):
        self.id = id
        self.bond_type = bond_type
        self.atom_id1 = atom_id1
        self.atom_id2 = atom_id2

    def get_id(self):
        return self.id

    def get_bond_type(self):
        return self.bond_type

    def get_atoms_id(self):
        ids = (self.atom_id1, self.atom_id2)
        return ids

    def set_bond_type(self, bond_type):
        self.bond_type = bond_type

    def set_atoms_id(self, atom_id1, atom_id2):
        self.atom_id1 = atom_id1
        self.atom_id2 = atom_id2

    def print_bond(self):
        bond_id = self.id
        bond_type = self.bond_type
        atom_id1 = self.atom_id1
        atom_id2 = self.atom_id2
        str = " {}\t{}\t{}\t{}\n".format(bond_id, bond_type, atom_id1, atom_id2).expandtabs()
        return str
```

D:\proj\UPB\stud\cod_cercetare\classes\Angle.py

```
class Angle:
    id = None
    angle_type = None
    atom_id1 = None
    atom_id2 = None
    atom_id3 = None

    def __init__(self, id, angle_type, atom_id1, atom_id2, atom_id3):
        self.id = id
        self.angle_type = angle_type
        self.atom_id1 = atom_id1
        self.atom_id2 = atom_id2
        self.atom_id3 = atom_id3

    def get_id(self):
        return self.id

    def get_angle_type(self):
        return self.angle_type

    def get_atoms(self):
        ids = (self.atom_id1, self.atom_id2, self.atom_id3)
        return ids

    def set_angle_type(self, angle_type):
        self.angle_type = angle_type

    def set_atom_ids(self, atom_id1, atom_id2, atom_id3):
        self.atom_id1 = atom_id1
        self.atom_id2 = atom_id2
        self.atom_id3 = atom_id3

    def print_angle(self):
        id = self.id
        angle_type = self.angle_type
        atom_id1 = self.atom_id1
        atom_id2 = self.atom_id2
        atom_id3 = self.atom_id3
        str = " {}\t{}\t{}\t{}\t{}\n".format(id, angle_type, atom_id1, atom_id2, atom_id3).expandtabs()
        return str
```

D:\proj\UPB\stud\cod_cercetare\modules\memory_management.py

```
from config.global_setting import OTHER_DEFAULTS


class Memory_handler:
    @staticmethod
    def free_atom_memory():
        pass
        print("Memory deallocated - erased")
```

D:\proj\UPB\stud\cod_cercetare\TestModule.py

```
from typing import List
from classes.LammpsInputFile import LammpsInputFile
from modules.periodic import Periodic
from modules.geometry import Geometry
from config.global_setting import *
from modules.lists import Lists
from modules.kinetics import Kinetics


#   Checked periodic functions from PERIODIC
def periodic_test_check(l, atom_list):
    OTHER_DEFAULTS.box = l.box
    print('PBC function check')
    Periodic.pbc(atom_list)

    print('MINIMG function check')
    atom_list[1].x, atom_list[1].y, atom_list[1].z = Periodic.minimg(atom_list[1].x, atom_list[1].y, atom_list[1].z)
    print(atom_list[1].x, atom_list[1].y, atom_list[1].z)

    print('GET_IMAGE_INDEX function check')
    atom_list[1].x, atom_list[1].y, atom_list[1].z = Periodic.get_image_index(atom_list[1].true)
    print(atom_list[1].x, atom_list[1].y, atom_list[1].z)

    print('GET_ITRUE function check')
    print(atom_list[1].x, atom_list[1].y, atom_list[1].z, atom_list[1].true)
    atom_list[1].true = Periodic.get_itrue(atom_list[1].x, atom_list[1].y, atom_list[1].z)
    print(atom_list[1].x, atom_list[1].y, atom_list[1].z, atom_list[1].true)

    print('SET_PRD function check')
    print(OTHER_DEFAULTS.xprd)
    Periodic.set_prd()
    print(OTHER_DEFAULTS.xprd)

# Check translate fucntion from GEOMETRY
def geometry_test_check_translate(l, atom_list):

    OTHER_DEFAULTS.box = l.box
    Geometry.translate(atom_list, (1, 2, 3))
    l.write_to_file("out/TEST_translate_velocity.out")

# Check find_molecules function from GEOMETRY
def geometry_test_check_find_molecules(l, atom_list):

    OTHER_DEFAULTS.box = l.box

    with open('out/TEST_original_molecule.out', 'w') as the_file2:
        for a in atom_list:
            the_file2.write(str(a.molecule_tag) + "\n")
    # Test for find molecules
    Geometry.find_molecules(atom_list, l.get_bonds())
    with open('out/TEST_modified_molecule.out', 'w') as the_file2:
        for a in atom_list:
            the_file2.write(str(a.id) + " " + str(a.molecule_tag) + "\n")

    with open('out/TEST_bonds.out', 'w') as the_file2:
        for b in l.get_bonds():
            the_file2.write(str(b.get_atoms_id()) + "\n")

# Check displace_atoms fucntion from GEOMETRY
def geometry_test_check_displace_atoms(l, atom_list):

    OTHER_DEFAULTS.box = l.box
    with open('out/TEST_displace_atoms.out', 'w') as the_file2:
        Geometry.displace_atoms(atom_list, 2)
        for a in atom_list:
            v = a.get_velocities()
            s = str(a.x) + " " + str(a.y) + " " + str(a.x) + " "
            vv = str(v[0]) + " " +  str(v[1]) + " " +  str(v[2])
            the_file2.write( s + vv + "\n")

# Check atoms_within_sphere funciton form GEOMETRY
def geometry_test_check_atoms_within_sphere(l, atom_list):
    OTHER_DEFAULTS.box = l.box

    with open('out/TEST_atom_within_sphere.out', 'w') as the_file2:
        first_atoms = atom_list[:10]
        result_list = []
        Geometry.get_atoms_within_sphere(first_atoms, (-9, -7, -9), 5, result_list)

        for a in first_atoms:
            s = str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
            the_file2.write( s + " " + "\n")

        the_file2.write( "\n\n\n")
        for a in result_list:
            s = str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
```

```
            the_file2.write( s + " " + "\n")

# Check calc_mc_pos and calc_mc_pos2 from KINETICS
def geometry_test_check_calc_mc_pos(l, atom_list):

    OTHER_DEFAULTS.box = l.box

    # Check calc_mc_pos function from kinetics module
    print('\nTest calc_mc_pos')
    first_atoms = atom_list[:10]
    point_res = []
    Kinetics.calc_mc_pos(first_atoms, l, point_res)

    # Check calc_mc_pos2
    print('\nTest calc_mc_pos2')
    xyz = [(1, 1, 2), (2, 3, 4), (1, 1.5, 1.3)]
    masses = [2, 3, 3.5]
    point_res2 = []
    Kinetics.calc_mc_pos2(3, xyz, masses, point_res2)

# Check calc_lin_mom, calc_ang_mom, calc_inertia_mom, calc_mc_vel from KINETICS
def geometry_test_check_calc_mom(l, atom_list):

    OTHER_DEFAULTS.box = l.box

    # Check calc_lin_mom
    print('\nTest calc_lin_mom')
    first_atoms = atom_list[:10]
    (px, py, pz) = Kinetics.calc_lin_mom(first_atoms, l)

    # Check calc_mc_vel
    print('\nTest calc_mc_vel')
    first_atoms = atom_list[:10]
    (vxcm, vycm, vzcm) = Kinetics.calc_mc_vel(first_atoms, l)
    # Check calc_ang_mom
    print('\nTest calc_ang_mom')
    first_atoms = atom_list[:10]
    angm = Kinetics.calc_ang_mom(first_atoms, l, [1, 1, 1])
    # Check calc_inertia_mom
    print('\n Test calc_inertia_mom')
    first_atoms = atom_list[:10]
    inm = Kinetics.calc_inertia_mom(first_atoms, l, [1,1, 1])

# Check calc_temperature, temp_rescale, calc_tvr_temperature from KINETICS
def geometry_test_check_temperature(l, atom_list):
    OTHER_DEFAULTS.box = l.box

    print('\n Test calc_temperature')
    first_atoms = atom_list[:10]
    temp = Kinetics.calc_temperature(first_atoms, l,1)

    print('\n Test temp_rescale')
    first_atoms = atom_list[:10]
    temp = Kinetics.temp_rescale(first_atoms, l, 20, 3)

    print('\n Test calc_tvr_temperature')
    first_atoms = atom_list[:10]
    vv=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]
    vt=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]
    vr=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]
    temp = Kinetics.calc_tvr_temperature(first_atoms, l, 20, vt, vv, vr)



# ################################################################################
# Check divide_cells function from Lists
def divide_cells(lammps_input):
    with open('TEST_Divide_cells.out', 'w') as fil:
        # first_atoms = atom_list[:10]
        # region =  {
        # "xlo xhi" : (-10, 3),
        # "ylo yhi": (-10, 3),
        # "zlo zhi": (-10, 3)
        # }
        # box =  {
        # "xlo xhi" : (-18, 4),
        # "ylo yhi": (-15, 6),
        # "zlo zhi": (-15, 7)
        # }
        # nx = 2
        # ny = 5
        # nz = 4

        atoms = Lists.divide_cells(l.get_atoms(), l.box, l.box)
        # for a in first_atoms:
        #     s = str(a.id) + " " + str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
        #     fil.write( s + " " + "\n")

        sum = 0
        for key in atoms:
```

```
            write_string = str(key) + " -> "
            for v in atoms[key]:
                write_string = write_string + str(v.id) + " "
            fil.write(write_string + "\n")
            sum += len(atoms[key])
        fil.write("Total # of Atoms = " + str(sum) + "\n")
        return atoms

if __name__ == "__main__":

    # Data
    l = LammpsInputFile("in/333_LA2.DATA")
    atom_list = l.get_atoms()
    OTHER_DEFAULTS.box = l.box

    # Tests
    periodic_test_check(l, atom_list)
    geometry_test_check_translate(l, atom_list)
    geometry_test_check_find_molecules(l, atom_list)
    geometry_test_check_atoms_within_sphere(l, atom_list)
    geometry_test_check_calc_mc_pos(l, atom_list)
    geometry_test_check_calc_mom(l, atom_list)
    geometry_test_check_temperature(l, atom_list)



    # Testing divide_cells
    atoms_indexed = divide_cells(l)
    Lists.get_hilbert_curve_crossing(l)
```

D:\proj\UPB\stud\cod_cercetare\TestCases.py

```
from typing import List
from classes.LammpsInputFile import LammpsInputFile
from modules.periodic import Periodic
from modules.geometry import Geometry
from config.global_setting import *
from modules.lists import Lists
from modules.kinetics import Kinetics

def divide_cells(lammps_input):
    with open('TEST_Divide_cells.out', 'w') as fil:
        # first_atoms = atom_list[:10]
        # region =  {
        # "xlo xhi" : (-10, 3),
        # "ylo yhi": (-10, 3),
        # "zlo zhi": (-10, 3)
        # }
        # box =  {
        # "xlo xhi" : (-18, 4),
        # "ylo yhi": (-15, 6),
        # "zlo zhi": (-15, 7)
        # }
        # nx = 2
        # ny = 5
        # nz = 4

        atoms = Lists.divide_cells(l.get_atoms(), l.box, l.box)
        # for a in first_atoms:
        #     s = str(a.id) + " " + str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
        #     fil.write( s + " " + "\n")

        sum = 0
        for key in atoms:
            write_string = str(key) + " -> "
            for v in atoms[key]:
                write_string = write_string + str(v.id) + " "
            fil.write(write_string + "\n")
            sum += len(atoms[key])
        fil.write("Total # of Atoms = " + str(sum) + "\n")
        return atoms

if __name__ == "__main__":

    l = LammpsInputFile("in/333_LA2.DATA")
    atom_list = l.get_atoms()
    OTHER_DEFAULTS.box = l.box

    # Testing divide_cells
    atoms_indexed = divide_cells(l)
    #print(atoms_indexed)
    Lists.get_hilbert_curve_crossing(l)

    #   Checked periodic functions from Periodic
    # Periodic.pbc(atom_list)
    # atom_list[1].x, atom_list[1].y, atom_list[1].z = Periodic.minimg(atom_list[1].x, atom_list[1].y, atom_list[1].z)
    # print(atom_list[1].x, atom_list[1].y, atom_list[1].z)
    # atom_list[1].x, atom_list[1].y, atom_list[1].z = Periodic.get_image_index(atom_list[1].true)
    # print(atom_list[1].x, atom_list[1].y, atom_list[1].z)
```

```
    # print(atom_list[1].x, atom_list[1].y, atom_list[1].z, atom_list[1].true)
    # atom_list[1].true= Periodic.get_itrue(atom_list[1].x, atom_list[1].y, atom_list[1].z)
    # print(atom_list[1].x, atom_list[1].y, atom_list[1].z, atom_list[1].true)

    # print(OTHER_DEFAULTS.xprd)
    # Periodic.set_prd()
    # print(OTHER_DEFAULTS.xprd)

    # #Checked translate function from velocity
    # Geometry.translate(atom_list, (1, 2, 3))
    # l.write_to_file("TEST_translate_velocity.out")

    # # Check find_molecules function
    # with open('TEST_original_molecule.out', 'w') as the_file2:
    #     for a in atom_list:
    #         the_file2.write(str(a.molecule_tag) + "\n")
    # # Test for find molecules
    # Geometry.find_molecules(atom_list, l.get_bonds())
    # with open('TEST_modified_molecule.out', 'w') as the_file2:
    #     for a in atom_list:
    #         the_file2.write(str(a.id) + " " + str(a.molecule_tag) + "\n")

    # with open('TEST_bonds.out', 'w') as the_file2:
    #     for b in l.get_bonds():
    #         the_file2.write(str(b.get_atoms_id()) + "\n")

    # #Test for displace_atoms

    # with open('TEST_displace_atoms.out', 'w') as the_file2:
    #     Geometry.displace_atoms(atom_list, 2)
    #     for a in atom_list:
    #         v = a.get_velocities()
    #         s = str(a.x) + " " + str(a.y) + " " + str(a.x) + " "
    #         vv = str(v[0]) + " " +  str(v[1]) + " " +  str(v[2])
    #         the_file2.write( s + vv + "\n")

    # #Test for atoms within sphere function

    # with open('TEST_atom_within_sphere.out', 'w') as the_file2:
    #     first_atoms = atom_list[:10]
    #     result_list = []
    #     Geometry.get_atoms_within_sphere(first_atoms, (-9, -7, -9), 5, result_list)
    #     for a in first_atoms:
    #         s = str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
    #         the_file2.write( s + " " + "\n")

    #     the_file2.write( "\n\n\n")
    #     for a in result_list:
    #         s = str(a.x) + " " + str(a.y) + " " + str(a.z) + " "
    #         the_file2.write( s + " " + "\n")



# Check calc_mc_pos function from kinetics module
    # print('\nTest calc_mc_pos')
    # first_atoms = atom_list[:10]
    # point_res = []
    # Kinetics.calc_mc_pos(first_atoms, l, point_res)


    # # Check calc_mc_pos2
    # print('\nTest calc_mc_pos2')
    # xyz=[(1, 1, 2), (2, 3, 4), (1, 1.5, 1.3)]
    # masses = [2, 3, 3.5]
    # point_res2 = []
    # Kinetics.calc_mc_pos2(3, xyz, masses, point_res2)

    # # Check calc_lin_mom
    # print('\nTest calc_lin_mom')
    # first_atoms = atom_list[:10]
    # (px, py, pz) = Kinetics.calc_lin_mom(first_atoms, l)

    # # Check calc_mc_vel
    # print('\nTest calc_mc_vel')
    # first_atoms = atom_list[:10]
    # (vxcm, vycm, vzcm) = Kinetics.calc_mc_vel(first_atoms, l)

    # print('\nTest calc_ang_mom')
    # first_atoms = atom_list[:10]
    # angm = Kinetics.calc_ang_mom(first_atoms, l, [1, 1, 1])

    # print('\n Test calc_inertia_mom')
    # first_atoms = atom_list[:10]
    # inm = Kinetics.calc_inertia_mom(first_atoms, l, [1,1, 1])

    # print('\n Test calc_temperature')
    # first_atoms = atom_list[:10]
    # temp = Kinetics.calc_temperature(first_atoms, l,1)

    # print('\n Test temp_rescale')
    # first_atoms = atom_list[:10]
    # temp = Kinetics.temp_rescale(first_atoms, l, 20, 3)

    print('\n Test calc_tvr_temperature')
    first_atoms = atom_list[:10]
    vv=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]
    vt=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]
    vr=[[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3],[1, 2, 3], ]


     temp = Kinetics.calc_tvr_temperature(first_atoms, l, 20, vt, vv, vr)
```