**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 t4l

maxbondper = 8

nmolecular = 0

# files in t4l

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)**