```
'''
Created on May 2, 2011

@author: neugebauer
'''
from utilities import abort, endl
import utilities as u
import numpy as np
from structure import CrystalStructure, AtomStructure
from job_queue import JobQueue
import pickle
import matplotlib.pyplot as pltInt
import tables
from scipy.spatial import cKDTree

class atomKMC:
    def __init__(self, control = None,
                 queue = None,
                 dictProject = None):
        if control is None:
            if dictProject is None:
                return
#                u.abort("atomKMC.init", err = "Either arg or dictProject must be given!")
            else:
                control = u.load(u.DumpDir(dictProject["Name"]))
#        print "control; ", dir(control)
#        exit()

        self.control = control
        self.structure = control.paraDict["structure"]
        self.reference_time = 0.
        if queue == None:
            self.queue = JobQueue()
        else:
            self.queue = queue

        self.Version = "1.0"  # TODO: should be connected to repository
        self.Type = "KMC"

        self.__dictProject = dictProject  #{"Name": None, "ID": -1}
        self.read_lattice_needs_update = True
        if self.hasProjectID():
            self.DB_item = self.getDB_Item()

# initialize/define module intern variables
        self.timeList = None
        self.time_search_tree = None


    def writePyTables(self, h5file, group):
        h5file.createGroup(group, "hamilton", "All data related to specific Hamilton")
        self.control.writePyTables(h5file, group.hamilton)
        attrs = group.hamilton._v_attrs
        attrs.Version = self.Version
```

```
        attrs.Type = self.Type
        attrs.ProjectExists = not (self.__dictProject is None)
        if not self.__dictProject is None:
            attrs.ProjectName = self.__dictProject["Name"]
            attrs.ProjectID = self.__dictProject["ID"]

    def loadPyTables(self, h5file, group):
        control = KMC_control()
        control.loadPyTables(h5file, group.hamilton)

        attrs = group.hamilton._v_attrs
        dictProject = None
        if attrs.ProjectExists:
            dictProject = {}
            dictProject["Name"] = attrs.ProjectName
            dictProject["ID"] = attrs.ProjectID

        self.__init__(control = control,
                      queue = None,
                      dictProject = dictProject
                      )

    def hasProjectID(self):
        if self.__dictProject is None:
            return False
        else:
            if "ID" in self.__dictProject.keys():
                return True
            else:
                return False
#        return not self.__dictProject is None

    def getProjectName(self):
        if self.hasProjectID():
            return self.__dictProject["Name"]

    def getProjectID(self):
        if self.hasProjectID():
            return self.__dictProject["ID"]

    def setProjectID(self, projectID):
        self.read_lattice_needs_update = True
        self.__dictProject["ID"] = projectID

    def copy(self):
        import copy
        return copy.copy(self)

    def dump(self,fileName):
        f = open(fileName,"w")
#TODO: pickling of self.queue fails, should be sufficient to provide dns name for
database
        pickle.dump(self.control, f)
        f.close()
```

```
    def getParameters(self):
        return self.control.paraDict.keys()

    def getDB_Item(self):
        from ProjectAdministrator import JobDatabase
        jobDB = JobDatabase(self.getProjectName())
        self.jobDB = jobDB
        self.jobDBTable = jobDB.dbTable
        id = self.getProjectID() + 1  # TODO: start ID number at 0!!
        item = self.jobDBTable.getItemsSQL("id = " + str(id))
        if len(item) == 0:
            print "item_id: ", id, self.getProjectName()
            u.abort("atomKMC.getDB_Item", err = "entry for this id is missing in database")
        return item[0]

    def getItem(self, key):
        return self.DB_item[key]

    def changeParameters(self,newDictPara):
        self.control.paraDict.update(newDictPara)

    def writeLattice(self, structure, fileName = "lattice.inp"):
        is0 = structure.ElementList.getMajorityIntSpecies()
#        print "is0: ",is0
        numberOfMinorityAtoms = len(structure) - structure.NumberOfSpeciesAtoms()[is0]
        print "Majority species: ", is0, numberOfMinorityAtoms, structure.ElementList.
        getMajorityIntSpecies()

        f = open(fileName, 'w')
        f.write('{0:d} {1:d}'.format(len(structure), structure.super_cell_size[0])+endl)
        f.write('{0:d} {1:d}'.format(structure.ElementList.getMajorityIntSpecies(),
        numberOfMinorityAtoms)+endl)
        f.write('referenceTime: {0:f}'.format(self.reference_time)+endl)

        elList = structure.ElementList.getIntIndex()
        for ia,el in enumerate(elList):
            if not(el == is0):
                f.write('{0:d}  {1:d}'.format(ia,el)+ endl)

        f.close()

    def writeLatticeTopology(self,
                             radius = None,
                             fileName = "latticeStructure.inp"):
        """
        Converts atomic structure into network topology that provides
        efficient way to describe any structure and rate using the
        KMC program
        """
        if radius is None:
            #TODO: provide automatic search for shells
            #      (i.e. take e.g. only nearest neighbor shells)
            radius = 2.
```

```
        self.structure.refresh_neighbors(radius = radius,
                                         excludeSelf = True)
        nbr_index = self.structure.neighbor_index

        print "min: ", self.structure.min_nbr_number
        print "max: ", self.structure.max_nbr_number
        print "supercell size: ", self.structure.super_cell_size
#         print self.structure.neighbor_distance[0]
#         print self.structure.neighbor_index[0]
#         print self.structure.neighbor_distance_vec[0]

        nx, ny = self.structure.super_cell_size[0:2]
        f = open(fileName, 'w')
        # max. number of neighbbors
        f.write('{0:d} '.format(self.structure.max_nbr_number) + endl)
        # number of atoms
        f.write('{0:d} {1:d} {2:d}'.format(len(self.structure), nx, ny) + endl)

        for i_atom, nbr in enumerate(nbr_index):
            f.write('{0:d} {1:d}'.format(i_atom, len(nbr)) + endl)
            for ia_nbr in nbr:
                prob = 1.0   # a more complex formula might be implemented here
                f.write('{0:d} {1:f}'.format(ia_nbr, prob) + endl)
        f.close()


    def readLattice(self, fileName = None):

        if fileName is None:
            if not self.__dictProject is None:
                fileName = u.WorkDir('lattice.out',
                                     dictProject = self.__dictProject)
            else:  # later: abort()
                fileName = u.srcDir('lattice.out')

        try:
            lines = open(fileName).readlines()
        except ValueError:
            abort("kmc.readLattice:", err = ValueError)

        timeList = []
        structList = []

        count = 0

        while count<len(lines):
#             na,nx =  lines[count].split()
            count += 1

            s =  lines[count].split()
            nAtoms = int(s[1])
            count += 1

            s,t = lines[count].split()
```

```
            if not(s == 'time'):
                abort('kmc.readLattice',err = "wrong file content")
            timeList.append(float(t))
            count += 1

            struct = []
            for iAtom in range(nAtoms):
                s =  lines[count].split()
                struct.append([int(s[0]), int(s[1])]) # ia, is
                count += 1
            structList.append(struct)

        self.timeList = timeList
        self.structList = structList
        self.initTimeSearchTree()

#         print "timeList: ",timeList
#         print "Number of iterations: ", len(structList)


    def run(self,directory = None):
        import os
        cwd = os.getcwd()
        if not(directory == None):
            try:
                os.chdir(directory)
            except ValueError:
                abort('KMC.run', err = ValueError)

        self.writeLattice(self.structure)
        if self.control.paraDict["ReadLatticeStructure"]:
            self.writeLatticeTopology()
        self.control.write()

        print "start"
        if os.path.isfile(self.control.kmcExecutable):
            os.system(self.control.kmcExecutable + ' > logfile.out')
        else:
            abort('KMC.run',err = 'executable not existent:' + self.control.kmcExecutable)

        print "finished"
        os.chdir(cwd)


    def submit(self,directory = None, queue = None, nameTag = None, projectJobID = None):
        import os

        if queue == None: queue = self.queue
        jobID = queue.getJobID()

        if nameTag == None: nameTag = "KMC"

        cwd = os.getcwd()
        if not(directory == None):
```

```
            try:
                os.chdir(directory)
            except ValueError, e:
                abort('KMC.run', err = e)
        if projectJobID==None:
            u.abort("atomKMC.submit", err = "projectJobID is missing!")# projectJobID = ""

        dictProject = {"Name": nameTag, "ID": projectJobID}
        wDir = u.WorkDir("", dictProject = dictProject)
        u.ensure_dir(wDir)
#         exit()
#         if not os.path.exists(wDir):
#             os.mkdir(wDir)

#         self.writeLattice(self.structure, fileName = os.path.join(wDir, 'lattice.inp'))
#         if self.control.paraDict["ReadLatticeStructure"]:
#             self.writeLatticeTopology(fileName = os.path.join(wDir, 'latticeStructure.inp'))
#         self.control.write(fileName = os.path.join(wDir, 'incontrol.dat'))

        self.writeLattice(self.structure, fileName = u.WorkDir('lattice.inp', dictProject =
        dictProject))
        if self.control.paraDict["ReadLatticeStructure"]:
            self.writeLatticeTopology(fileName = u.WorkDir('latticeStructure.inp', dictProject =
             dictProject))
        self.control.write(fileName = u.WorkDir('incontrol.dat', dictProject = dictProject))

        executable = self.control.kmcExecutable
        self.queue.createRunFile(nameTag, jobID, executable, id = projectJobID, directory = wDir)

        os.chdir(cwd)

        return jobID

    def getStructure(self, iStep, extractSpecies = None):
        if self.read_lattice_needs_update:
            self.readLattice()

        basis0 = self.structure.deepcopy()
        itLen = len(self.structList)
        elList = basis0.ElementList.getFullElementList(self.structList[iStep % itLen])
        basis0.ElementList = elList
        if not extractSpecies is None:
            basis0.ExtractSpecies(extractSpecies)

        return basis0

    def initTimeSearchTree(self):
        print "time tree", self.timeList
        if self.timeList is None:
            u.abort("atomKMC.getIndex", err = "timeList not defined")

        myTimeList = u.r(self.timeList).reshape(len(self.timeList), -1)
        self.time_search_tree = cKDTree(myTimeList)
```

```
    def getIndex(self, time):
        print "getIndex"
        ind = self.time_search_tree.query([time])[1]
        return ind

    def getHistogram(self, iStep, readFile, maxSize = None, bins = 10, normed = False):
        if readFile: self.readLattice()

        basis0 = self.structure.deepcopy()
        itLen = len(self.structList)
        elList = basis0.ElementList.getFullElementList(self.structList[iStep % itLen])
        basis0.ElementList = elList
        basis0.ExtractSpecies(["Al"])
        sizes = basis0.ClusterAnalysis(radius = 2.)
        average = sum(sizes)/float(len(sizes))
        print "cluster: ", sizes
        if maxSize == None: maxSize = max(sizes)
        yHist,xHist = np.histogram(sizes, bins=bins, range=(0,maxSize), normed=normed)
        return xHist,yHist, average, basis0

    def plotHistogram(self, iStep, readFile = True, pltExt = None):
        xHist,yHist, _, basis0 = self.getHistogram(iStep,readFile)

        itLen = len(self.structList)
        if pltExt is None:
            plt = pltInt
            plt.plot(xHist[:-1], yHist)
            plt.show()
        else:
            plt = pltExt.axes
            plt.plot(xHist[:-1], yHist)
            plt.set_title("KMC Step: " + str(iStep%itLen) + " at t = " + str(self.timeList[iStep
            %itLen])+ "s")
            plt.set_xlabel("size")
            plt.set_ylabel("# occurences")

            pltExt.figure.canvas.draw()

    def plot2d(self, iStep, readFile = True, pltExt = None):
        xHist,yHist, _, basis0 = self.getHistogram(iStep,readFile)

        if self.structure.dimension == 3:
            basis0.plot3d(modal = True, scale_radius = 0.1)
            return

        basis0.setRelative()
        x = basis0.coordinates[:,0]
        y = basis0.coordinates[:,1]
        xMin,yMin = [x.min(),y.min()]
        nx,ny = self.structure.super_cell_size[0:2]
        x = (x - xMin) * nx
        y = (y - yMin) * ny
#        print "len: ", len(x)
```

```
        itLen = len(self.structList)
        if pltExt is None:
            plt = pltInt
            plt.figure()
            plt.subplot(121,aspect = 'equal')
            plt.plot(x, y, 'o' ,markersize = 2)

            plt.title("KMC Step: " + str(iStep%itLen) + " at t = " + str(self.timeList[iStep%
            itLen])+ "s")
            plt.xlabel("x")
            plt.ylabel("y")

            plt.xlim(-1,nx+1)
            plt.ylim(-1,ny+1)

            plt.subplot(122)
            plt.plot(xHist[:-1], yHist)
            plt.show()
        else:
            plt = pltExt.axes
            plt.plot(x, y, 'o' ,markersize = 2)

            plt.set_title("KMC Step: " + str(iStep%itLen) + " at t = " + str(self.timeList[iStep
            %itLen])+ "s")
            plt.set_xlabel("x")
            plt.set_ylabel("y")

            plt.set_xlim(-1,nx+1)
            plt.set_ylim(-1,ny+1)
            pltExt.figure.canvas.draw()


    def showGUI(self):
        '''
        use this command not to open a new window from an existing one
        '''
        import sys
        from PyQt4 import QtGui
        from kmcHamiltonViewerGUI import MainWindow

        app = QtGui.QApplication(sys.argv)
        dialog = MainWindow(self)

        dialog.show()
        app.exec_()

        return


class KMC_control:
    def __init__(self,
                 structure = None,
                 temperature = 1500,
                 energies = None,
```

```
                barriers = None,
                fluxes =  None,
                stopTime = 1.,
                NumberOfSteps = 100,
                DrawLatticeSteps = 1000,
                ReadLatticeStructure = False,
                Restart = True,
                kmcExecutable = u.PATH_KMC_EXE
                ):

        if structure is None:
            return

        if energies is None: energies = [[1.,0.1],[0.1,2]]
        if barriers is None: barriers = [1.] * len(energies)
        if fluxes is None: fluxes = [0.] * len(energies)

#TODO: use the dictionary formulation as template for all other parameter sets
# highly efficient, allows easy change of individual parameters and python like
(**kwargs)
        self.paraDict={"structure":structure,
                       "temperature":temperature,
                       "stopTime":stopTime,
                       "NumberOfSteps":NumberOfSteps,
                       "DrawLatticeSteps":DrawLatticeSteps,
                       "ReadLatticeStructure":ReadLatticeStructure,
                       "Restart":Restart,
                       "NumberOfSpecies":len(energies),
                       "energies":energies,
                       "barriers":barriers,
                       "fluxes":fluxes}

        self.structure = structure
        self.temperature = temperature
        self.stopTime = stopTime
        self.NumberOfSteps = NumberOfSteps
        self.DrawLatticeSteps = DrawLatticeSteps
        self.ReadLatticeStructure = ReadLatticeStructure
        self.Restart = Restart
        self.energies = energies
        self.barriers = barriers
        self.fluxes = fluxes

        self.kmcExecutable = kmcExecutable

    def writePyTables(self, h5file, group):
        h5file.createGroup(group, "KMC_control", "data to initialize KMC Hamilton")
        self.structure.writePyTables(h5file, group)

        h5file.createArray(group.KMC_control, "energies", self.energies, "binding energy matrix
        E_ij")
        h5file.createArray(group.KMC_control, "barriers", self.barriers, "diffusion barriers")
        h5file.createArray(group.KMC_control, "fluxes", self.fluxes, "fluxes for grand
        canonical systems")
```

```
        attrs = group.KMC_control.energies.attrs
        attrs.temperature = self.temperature
        attrs.stopTime = self.stopTime
        attrs.NumberOfSteps = self.NumberOfSteps
        attrs.DrawLatticeSteps = self.DrawLatticeSteps
        attrs.ReadLatticeStructure = self.ReadLatticeStructure
        attrs.Restart = self.Restart

    def loadPyTables(self, h5file, group):
        myStructure = AtomStructure()
        myStructure.loadPyTables(h5file, group)

        attrs = group.KMC_control.energies.attrs
        energies = group.KMC_control.energies.read()
        barriers = group.KMC_control.barriers.read()
        fluxes = group.KMC_control.fluxes.read()

        self.__init__(
                      structure = myStructure,
                      temperature = attrs.temperature ,
                      energies = energies,
                      barriers = barriers,
                      fluxes =  fluxes,
                      stopTime = attrs.stopTime,
                      NumberOfSteps = attrs.NumberOfSteps,
                      DrawLatticeSteps = attrs.DrawLatticeSteps,
                      ReadLatticeStructure = attrs.ReadLatticeStructure,
                      Restart = attrs.Restart,
                      kmcExecutable = u.PATH_KMC_EXE
                     )

    def bool_to_c(self,bool):
        if bool:
            return 1
        else:
            return 0


    def write(self, fileName = "incontrol.dat"):
        dict = self.paraDict
        f = open(fileName, 'w')
        f.write('Restart: {0:4d}'.format(dict["Restart"])+endl)
        f.write('ReadLatticeStructure: {0:1d}'.format(dict["ReadLatticeStructure"])+endl)
        f.write('Temperature: {0:f}'.format(dict["temperature"])+endl)
        f.write('stopTime: {0:f}'.format(dict["stopTime"])+endl)
        f.write('NumberOfSteps: {0:d}'.format(dict["NumberOfSteps"])+endl)
        f.write('DrawLatticeSteps: {0:d}'.format(dict["DrawLatticeSteps"])+endl)
        f.write('NumberOfSpecies: {0:2d}'.format(dict["NumberOfSpecies"])+endl)
        for i_s in range(dict["NumberOfSpecies"]):
            f.write('fluxes: {0:f}'.format(dict["fluxes"][i_s])+endl)
        for i_s in range(dict["NumberOfSpecies"]):
            f.write('DiffusionBarriers: {0:f}'.format(dict["barriers"][i_s])+endl)
        for i_s in range(dict["NumberOfSpecies"]):
```

```
            for j_s in range(dict["NumberOfSpecies"]):
                f.write('BindingEnergy: {0:f}'.format(dict["energies"][i_s][j_s])+endl)

        if dict["structure"].dimension == 3:
            nx, ny, nz = dict["structure"].super_cell_size
        else:
            nx, ny = dict["structure"].super_cell_size[0:2]
            nz = 1

        f.write('SuperLattice: {0:d} {1:d} {2:d}'.format(nx,ny,nz)+endl)

        f.close()

if __name__ == '__main__':
    import random
    dim = 3
    if dim == 2:
        alat = 1.9
        basis = CrystalStructure(element = "Fe",
                                 BravaisBasis = 'primitive',
                                 Dimension = dim,
                                 LatticeConstants = [alat]).getAtomStructure()

#        basis.amat[2][2]=20.
        basis.repeat([25,25,1])
        na = len(basis)
        iRandom = random.sample(range(na),20)
        for i in iRandom:
            basis.substituteAtom(i, "Al")

        basis.print_short(DropMajoritySpecies=True)
        basis0 = basis.copy()

        kmcControl = KMC_control(basis,
                                 stopTime = 1000,
                                 temperature = 1000,
                                 ReadLatticeStructure = True,
                                 DrawLatticeSteps = 10000,
                                 NumberOfSteps = 100)

        kmc = atomKMC(kmcControl)
        kmc.changeParameters({"temperature":1500,"stopTime":10})

#        kmc.__dictProject = {"Name": "test", "ID": 2}
#        kmc.setProjectID(3)

        file = tables.openFile("test.h5", mode = "w", title = "KMC Hamilton")
        root = file.root
        kmc.writePyTables(file, root)
        file.close()

        file = tables.openFile("test.h5", mode = "r")
        root = file.root
        kmcNew = atomKMC()
```

```
        kmcNew.loadPyTables(file, root)
        file.close()

        kmc.writeLatticeTopology(radius = 2.)
        print "finished"

        u.initDirectoryStructure()
        kmc.dump(u.DumpDir("kmcTest.dat"))
        print "finished"

        kmc.run()
        kmc.readLattice()
        for timeStep in range(0,10):
            print "index: ", timeStep, kmc.getIndex(timeStep/10.), max(kmc.timeList), len(kmc.
            timeList)
        #exit()
        kmc.plot2d(-1)
    elif dim == 3:
        alat = 1.9
        basis = CrystalStructure(element = "Fe",
                                 BravaisBasis = 'fcc',
                                 LatticeConstants = [alat]).getAtomStructure()

        basis.repeat([15, 15, 15])
        na = len(basis)
        iRandom = random.sample(range(na), 200)
        for i in iRandom:
            basis.substituteAtom(i, "Al")

        basis.print_short(DropMajoritySpecies = True)
        basis0 = basis.copy()

        kmcControl = KMC_control(basis,
                                 stopTime = 1000,
                                 temperature = 1000,
                                 ReadLatticeStructure = True,
                                 DrawLatticeSteps = 10000,
                                 NumberOfSteps = 100)

        kmc = atomKMC(kmcControl)
        kmc.changeParameters({"temperature":1500, "stopTime":10})

        kmc.writeLatticeTopology(radius = 1.6)
        print "finished"

        u.initDirectoryStructure()
        kmc.dump(u.DumpDir("kmcTest.dat"))
        print "finished"

        kmc.run()
        kmc.readLattice()
        kmc.plot2d(-1)
    #exit()
    basis0.ElementList.setSparseElementList(kmc.structList[10])
```

```
    basis0.print_short(DropMajoritySpecies = True)
    basis0.substituteAtom([["N",100]])
    basis0.ExtractSpecies(["Al","N"])
    basis0.showGUI()
```