```
import numpy as np
from databaseUtilities import databaseAccess
from utilities import r, abort
import copy
#from ChemicalElementData import ChemicalElementTable
import pyspglib._spglib as spg
from ElementList import elementList as ElementList
from scipy.spatial import cKDTree
import matplotlib.pyplot as plt

import sys
from PyQt4 import QtGui
from StructureInspectorGUI import MainWindow

import tables

class AtomStructure:
    """Structure class to efficiently set up atomic structure."""

    def __init__(self, elementList = None,
                       coordinates = None,
                       amat = None,
                       tag = None,
                       rel = True,               # coordinates are given in relative lattice
                       units
                       pbc = [True, True, True]  # periodic boundary conditions
                       ):

        if elementList is None:
            return

        self.ElementList = elementList

        self.plot3d_initialized = False
        self.dbTable = None
        self.colList = [["id", "int"],["x","float"],["y","float"],["z","float"],["el",
        "varchar(2)"]]

        self.dimension = len(amat)

        self.amat = amat
        self.coordinates = coordinates
        self.rel_coordinates = rel
        self.pbc = pbc
        self.units = "Angstrom"
        self.tag = tag

# TODO: check if needed
        self.super_cell_size = r(self.dimension * [1])

        self.neighbor_distance = [] # neighbors[0]
        self.neighbor_distance_vec = []
        self.neighbor_index = []
        self.min_nbr_number = None
```

```
        self.max_nbr_number = None

    def writePyTables(self, h5file, group):
        h5file.createGroup(group, "structure", "All data of atomic structure")
        self.ElementList.writePyTables(h5file, group.structure)

        h5file.createArray(group.structure, "amat", self.amat, "Lattice vectors")

        attrs = group.structure.amat.attrs
        attrs.dimension = self.dimension
        attrs.rel_coordinates = self.rel_coordinates
        attrs.pbc = self.pbc
        attrs.tag = self.tag
        attrs.units = self.units

        h5file.createArray(group.structure,
                           "coordinates",
                           self.coordinates,
                           "Atomic coordinates")

    def loadPyTables(self, h5file, group):
        elList = ElementList()
        print "group: ",group
        elList.loadPyTables(h5file, group.structure)

        attrs = group.structure.amat.attrs
        dimension = attrs.dimension
        rel = attrs.rel_coordinates
        pbc = attrs.pbc
        units = attrs.units
        tag = attrs.tag

        amat = group.structure.amat.read()
        coordinates = group.structure.coordinates.read()

        self.__init__(elementList = elList,
                      coordinates = coordinates,
                      amat = amat,
                      tag = tag,
                      rel = rel,
                      pbc = pbc)

        if not self.dimension == dimension:
            abort("structure.loadPyTables", err = "Inconsistency in alat dimension!")
        self.units = units




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

    def deepcopy(self):
        return (copy.deepcopy(self))
```

```
    def reloadDBTable(self):
        self.createDBTable(self.dbTable.table_name)

    def createDBTable(self,dbTableName,dbName = 'mdb'):
        self.dbTable = databaseAccess(dbName)
        self.dbTable.obj = self
        colList = self.colList
        coordDic = {}
        elList = self.ElementList.getSymIndex()
#        print "length1: ",len(elList)
#        print "length2: ",len(self.coordinates)
#        print "elList: ",elList
        for i,el in enumerate(elList):
            x,y,z = self.coordinates[i]
            coordDic[i] = (x,y,z,el)

        self.dbTable.deleteTable(dbTableName)
        self.dbTable.createTable(tableName = dbTableName,
                                 colList = colList,
                                 data = coordDic,
                                 unique = "id")

        colNames, _ = zip(*colList)
        self.dbTable.setColumnSequence(zip(colNames,[col.lower() for col in colNames]))

#TODO: should be replaced by a more general Extract function that applies also to tags,
coordinates etc.
    def ExtractSpecies(self,elList):
        selElementList, selIndex = self.ElementList.select(elList)
        self.coordinates = self.coordinates[selIndex]
        self.ElementList = selElementList

    def deleteAtom(self, index):
        if not(type(index)==type(int())):
            print "deleteAtom: index must be of type integer"
            exit()
        elif index >= len(self) | index < 0:
            print "deleteAtom: index out of range"
            exit()

        self.ElementList.delete(index)
        self.coordinates = np.delete(self.coordinates, index, axis=0)

    def substituteAtom(self, index, el):
        self.ElementList.substitute([[el,index]])

    def addAtoms(self, el, coords, rel = False ):
        coordsRel = self.relCoord(coords,rel)
#        self.setElementDict(self.elementDict.values() + el)
        self.coordinates = np.append(self.coordinates,coordsRel, axis=0)
        self.ElementList = self.ElementList + ElementList[el]

    def addAtom(self, el, coords, rel = False):
```

```
        self.addAtoms([el],[coords],rel)

    def relCoord(self,coordList,source,target = None):
        if target == None:
            target = self.rel_coordinates

        if source == target:
            return coordList
        elif source:      # relative to absolute coordinates
            return r([np.dot(self.amat,coord) for coord in coordList])
        else:
            bmat = np.linalg.inv(self.amat)
            return r([np.dot(bmat,coord) for coord in coordList])

    def IdenticalUnitCell(self, myBasis):
        return (abs(np.linalg.det(self.amat - myBasis.amat)) < 1e-10)



#TODO: check whether to replace by __xor__ to free __add__ for coordinates
    def __add__(self, addBasis):
        if not(self.IdenticalUnitCell(addBasis)):
            print "Add new basis failed: Unit cells are not identical!"
            exit()
        newBasis = copy.copy(self)
        newBasis.coordinates = np.append(self.coordinates,addBasis.coordinates, axis=0)
        newBasis.ElementList = self.ElementList + addBasis.ElementList
        return (newBasis)

    def transformCell(self,val):
        if type(val) == type (list()):
            print "transformCell: not implemented"
            exit()
        else:
            rel_coordinates = self.rel_coordinates
            self.setRelative()
            self.amat = val*self.amat
            self.coordinates = val * self.coordinates
            if not(rel_coordinates): # return original state
                self.setAbsolute()


    def shift (self,vec,rel = True):
        relVec = self.relCoord([vec],rel)[0]
        self.coordinates = [coord + relVec for coord in self.coordinates]


    def print_short(self, DropMajoritySpecies = False):
        for el in self.ElementList.listSpecies:
            print "Element:                " + el

        print "Supercell size         " + str(self.super_cell_size)
        print "Number of basis atoms: " + str(len(self.coordinates))
        if self.rel_coordinates:
```

```
            print "Coordinates given in relative units"
        else:
            print "Coordinates given in absolute units"
#         print "elements: " + str(self.elementList)
        el0 = self.ElementList.getMajoritySymSpecies()

        elList = self.ElementList.getSymIndex()
        for i,coord in enumerate(self.coordinates):
            el = elList[i]
            if not(DropMajoritySpecies & (el == el0)):
                print "coord("+str(i)+"): " + str(coord) + "  " + el
        print "a1, a2, a3:            " + str(self.amat[0])
        if self.dimension > 1:
            print "                       " + str(self.amat[1])
        if self.dimension > 2:
            print "                       " + str(self.amat[2])


    def repeat(self, repeatVec):

        if self.rel_coordinates:
            self.setAbsolute()

        dim = self.dimension
        intVec = [repeatVec[0],1,1]
        a1 = self.amat[0]
        a2,a3 = 0,0
        if dim > 1:
            a2=self.amat[1]
            intVec[1] = repeatVec[1]
        if dim > 2:
            a3=self.amat[2]
            intVec[2] = repeatVec[2]

        i_count = 0
        number_of_atoms = intVec[0]*intVec[1]*intVec[2]*len(self.coordinates)
        new_coordinates = np.zeros((number_of_atoms,dim))
        el_list = self.ElementList.getIntIndex()
        new_el_list = range(number_of_atoms)
        for ix in range(0,intVec[0]):
            for iy in range(0,intVec[1]):
                for iz in range(0,intVec[2]):
                    for ia in range(0,len(self.coordinates)):
                        new_coordinates[i_count] = ix*a1 +iy*a2+iz*a3+self.coordinates[ia]
#                         new_el_list[i_count] = self.elementList[ia]
                        new_el_list[i_count] = el_list[ia]
                        i_count = i_count+1

        self.coordinates = new_coordinates
        listSpecies = self.ElementList.listSpecies
#         print "listSpecies: ", listSpecies
#         print "elList: ", new_el_list
        self.ElementList = ElementList(new_el_list, listSpecies = listSpecies)
```

```
        self.amat=self.amat * intVec[:dim]
        self.super_cell_size = self.super_cell_size * intVec[:dim]

#TODO: determine automatically nearest neighbor shell to avoid explicit radius
    def ClusterAnalysis(self, radius, speciesList = None, max_num_neighbors = 14):
        self.refresh_neighbors(radius, max_num_neighbors)
        self.cluster = [0] * len(self)
        cCount = 1
        myElementList = self.ElementList.getIntIndex()
        for ia in range(len(self)):
            el0 = myElementList[ia]
            nbrs = self.neighbor_index[ia]
            if self.cluster[ia]==0:
                self.cluster[ia] = cCount
                self.__TestCluster(cCount, el0, nbrs)
                cCount += 1

        sizes = [self.cluster.count(ic+1) for ic in range(cCount-1)]
        return sizes

    def __TestCluster(self,cCount,el0,neighbors):
        myElementList = self.ElementList.getIntIndex()
        for nbr in neighbors:
            if self.cluster[nbr] == 0:
                if el0 == myElementList[nbr]: # TODO: check also for ordered structures
                    self.cluster[nbr] = cCount
                    nbrs = self.neighbor_index[nbr]
                    self.__TestCluster(cCount, el0, nbrs)

    def __selectSlice(self, iDim, iFlag, dist):
        if not self.rel_coordinates:
            abort("__selectSlice works only for relative coordinates")
        if iDim + 1 > self.dimension:
            return True
        if iFlag == 1:
            return (self.coordinates[:,iDim] < dist)
        elif iFlag == 0:
            return True
        elif iFlag == -1:
            return (self.coordinates[:,iDim] > 1-dist)

    def getBoundaryRegion(self, dist):
        '''
        get all atoms in the boundary around the supercell which have a distance
        to the supercell boundary of less than dist
        '''
        self.setRelative()
        rel_coordinates = self.coordinates

        dim = self.dimension
        a1 = self.amat[0]
        a2,a3 = 0,0
        iy,iz = 1,1
        iyl,izl = 0,0
```

```
        if dim > 1:
            a2=self.amat[1]
            iyl,iy = -1,2
        if dim > 2:
            a3=self.amat[2]
            izl,iz = -1,2
#        a1,a2,a3 = self.amat
        index = r(range(len(self)))
        newCoordinates = r([dim*[0]])
        pbcVec = r([dim*[0]])
        iaList = r([[0]])
        for i0 in range(-1,2):
            for i1 in range(iyl,iy):
                for i2 in range(izl,iz):
                    rVecAbs = i0*a1 + i1*a2 + i2*a3
                    rVec = r([i0,i1,i2])[:dim]
                    select = self.__selectSlice(0,i0,dist) & self.__selectSlice(1,i1,dist) &
                    self.__selectSlice(2,i2,dist)
                    if i0*i0 + i1*i1 + i2*i2 > 0:
#                        print "select: ", select, i0,i1,i2
                       if len(select) > 0:
#                            selCoordinates = abs_coordinates[select] - rVec
                           selCoordinates = rel_coordinates[select] + rVec

                           newCoordinates = np.append(newCoordinates, selCoordinates, axis=0)
                           if len(selCoordinates) > 0:
                               rVecs = len(selCoordinates) * [rVecAbs]
                               pbcVec = np.append(pbcVec, r(rVecs), axis = 0)
                               iaList = np.append(iaList, index[select])
#                                print "rVec: ", i0,i1,i2,rVecs[0],index[select],select

        return newCoordinates[1:], pbcVec[1:], iaList[1:]


    def refresh_neighbors(self, radius,
                          max_num_neighbors = 20,
                          tVec = True,
                          include_boundary = True,
                          excludeSelf = False):
        """
        arguments::
        radius:
             distance up to which nearest neighbors are searched for
             (in absolute units)
        tVec = True:
             compute distance vectors
                (pbc are automatically taken into account)
        include_boundary = True:
             search for neighbors assuming periodic boundary conditions
             False is needed e.g. in plot routines to avoid showing
             incorrect bonds
        excludeSelf = False:
             include central atom (i.e. distance = 0)
        """
```

```
        if (excludeSelf):
            def f_ind(x): return (x > 0.001) & (x < len(self))
        else:
            def f_ind(x): return x < len(self)

        if not include_boundary:  # periodic boundaries are NOT included
            tree = cKDTree(self.coordinates)
            neighbors = tree.query(self.coordinates, k = max_num_neighbors, distance_upper_bound
             = radius)

            self.neighbor_distance = neighbors[0]
            self.neighbor_index = map(lambda x:filter(f_ind,x),neighbors[1])
            return



#         print "periodicity included"
        # include periodic boundaries
        # translate radius in boundary layer with relative coordinates
        rel_width = [radius/np.sqrt(np.dot(ai,ai)) for ai in self.amat]
        rel_width_scalar = np.max(rel_width)

        # construct cell with additional atoms bounding original cell
        boundaryAtoms,bound_rVecs, ia0 = self.getBoundaryRegion(rel_width_scalar)
        elementList = self.ElementList.getIntIndex()
#         boundaryElementList = elementList[ia0]
        boundaryElementList = [elementList[ia] for ia in ia0]
        extCoordinates = np.append(self.coordinates, boundaryAtoms, axis=0)
        listSpecies = self.ElementList.listSpecies
#         dictSpecies = self.ElementList.dictSpecies
        extElementList = ElementList(np.append(elementList, boundaryElementList),
                                     listSpecies = listSpecies) #,
#                                      dictSpecies = dictSpecies)
        extendedCell = AtomStructure(extElementList,
                                     extCoordinates,
                                     self.amat,
                                     rel = True)

        # build index to map boundary atoms back to original cell
        map_to_cell = np.append(r(range(len(self))), ia0)

        self.setAbsolute()
        extendedCell.setAbsolute()

        tree = cKDTree(extendedCell.coordinates)
        neighbors = tree.query(self.coordinates,
                               k = max_num_neighbors,
                               distance_upper_bound = radius)

        self.neighbor_distance = [] # neighbors[0]
        self.neighbor_distance_vec = []
        self.neighbor_index = []
```

```
        i_start = 0
        if (excludeSelf):
            i_start = 1

        def f_ind_ext(x): return x < len(extendedCell)

        neighbor_index = map(lambda x:filter(f_ind_ext,x),neighbors[1])
        numNeighbors = []
        for i,index in enumerate(neighbor_index):
            self.neighbor_distance.append(neighbors[0][i][i_start:len(index)])
            self.neighbor_index.append(map_to_cell[index][i_start:])
            if tVec:
                vec0 = self.coordinates[index[0]]
                nbr_dist = []
                for i_nbr,ind in enumerate(index[1:]):
                    ind0 = map_to_cell[ind]
                    if ind0 != ind:
                        vecRij = self.coordinates[ind0] + bound_rVecs[ind-len(self)] - vec0

                    else:
                        vecRij = self.coordinates[ind0] - vec0

                    dd0 = neighbors[0][i][i_nbr+1]
                    dd = np.sqrt(np.dot(vecRij,vecRij))
#                    print "nbr: ", dd0,dd
                    if (dd - dd0 > 0.001):
                        print "wrong: ", vecRij, dd,dd0,i_nbr,ind,ind0,i
                        print self.coordinates[ind0], bound_rVecs[ind-len(self)], vec0
                    nbr_dist.append(vecRij)
                self.neighbor_distance_vec.append(nbr_dist)
            numNeighbors.append(len(index) - i_start)
#            print index
        min_nbr,max_nbr = min(numNeighbors), max(numNeighbors)
        if max_nbr == max_num_neighbors:
            print self.neighbor_distance
            abort("structure.refresh_neighbors" + str(max_nbr) + " " + str(max_num_neighbors),
                  err = "Increase max. number of neighbors!")
        self.min_nbr_number = min_nbr
        self.max_nbr_number = max_nbr


    def __len__(self):
        return len(self.coordinates)

    def NumberOfSpecies(self):
        return self.ElementList.getNumberOfSpecies()

    def NumberOfSpeciesAtoms(self):
        return self.ElementList.getNumberOfSpeciesAtoms()

    def setAbsolute(self):
        if self.rel_coordinates:
            self.coordinates = self.relCoord(self.coordinates,True,False)
            self.rel_coordinates = False
```

```
    def setRelative(self):
        if not(self.rel_coordinates):
            self.coordinates = self.relCoord(self.coordinates,False,True)
            self.rel_coordinates = True

    def getSpacegroup(self, symprec=1e-5):
        """
        Return space group in international table symbol and number
        as a string.
        """
        # Atomic positions have to be specified by scaled positions for spglib.
        self.setRelative()
#        elementList = r([1] * len(self.elementList))
        return spg.spacegroup(self.amat.copy(),
                              self.coordinates.copy(),
                              r(self.ElementList.getIntIndex()),
                              symprec)

    def getSymmetry(self,symprec=1e-5):

        multi = 48 * len(self)
        rotation = np.zeros((multi, 3, 3), dtype=int)
        translation = np.zeros((multi, 3))

        # Get symmetry operations
        self.setRelative()
        num_sym = spg.symmetry( rotation,
                                translation,
                                self.amat.copy(),
                                self.coordinates.copy(),
                                r(self.ElementList.getIntIndex()),
                                symprec )

        return {'rotation': rotation[:num_sym], 'translation': translation[:num_sym]}

    def getRefinedCell(self, symprec=1e-5):
        """
        Return refined cell
        """
        # Atomic positions have to be specified by scaled positions for spglib.
        self.setRelative()

        amat = self.amat.copy()
        coordinates = self.coordinates.copy()
        speciesList = self.ElementList.getIntIndex()
        #TODO: check whether lattice is transposed with respect to our definition
        num_atom_bravais = spg.refine_cell(amat,
                                           coordinates,
                                           r(speciesList),
                                           len(self),
                                           symprec)

        listSpecies = self.ElementList.listSpecies
```

```
        elList = ElementList(speciesList[:num_atom_bravais],
                             listSpecies = listSpecies)
        return AtomStructure(elList, coordinates[:num_atom_bravais],amat)


    def getPrimitiveCell(self, symprec=1e-5):
        """
        Find primitive cell in the input cell
        return primitive cell as AtomStructucture class.
        If no primitive cell is found, None is returned.
        """
        self.setRelative()
        amat = self.amat.copy()
        coordinates = self.coordinates.copy()
        speciesList = r(self.ElementList.getIntIndex())
        # lattice is transposed with respect to the definition of Atoms class
        num_atom_prim = spg.primitive(amat,
                                      coordinates,
                                      speciesList,
                                      symprec)
        print "num_atom_prim: ", num_atom_prim
        if num_atom_prim > 0:
            listSpecies = self.ElementList.listSpecies
            dictSpecies = self.ElementList.dictSpecies
            elList = ElementList(speciesList[:num_atom_prim],
                                 listSpecies = listSpecies) #,
#                                 dictSpecies = dictSpecies)
            return AtomStructure(elList, coordinates[:num_atom_prim],amat)
        else:
            return None, None, None

    def AtomicMassDOF(self):
        dim = self.dimension
        elDic = {}
        for el in self.ElementList.dictSpecies.values():
            elDic[el]=[self.ElementList.elementObjDict[el].AtomicMass for _ in range(dim)]
        mass = r([elDic[el]  for el in self.ElementList.list])
        return mass.reshape(-1)

    def plot1d(self, **kwargs):
        # define default parameters
        options = {
                   "vec"      : None,
                   "scaleVec" : 1,
                   "atomSize" : 5
                   }

        options.update(kwargs)
        if (len(options.keys())) < len(kwargs.keys()):
            abort("plot1d", err = "argument does not exist!")
        if not self.dimension == 1:
            abort("structure.plot1d",
                  err = "implemented only for dimension = 1 but dim = " +
                        str(self.dimension))
```

```
        vec = options["vec"]
        scaleVec = options["scaleVec"]
        atomSize = options["atomSize"]

        x = self.coordinates[:,0]
        y = [0. for _ in x]
        xMin = x.min()
        rad = self.ElementList.CovalentRadius()
#         nx = self.super_cell_size[0:1]

        plt.figure()
        plt.subplot(111,aspect = 'equal')
        colorTable = ['red', 'blue', 'green', 'black', 'yellow']
        myDict = self.ElementList.getDict()
        for iEl, atoms in enumerate(myDict.values()):
            xEl = [x[i] for i in atoms]
#             print "xEl: ", xEl
            xEl = (xEl - xMin) #* nx
            yEl = [0. for _ in xEl]
            myAtomSize = rad[i] * atomSize

            plt.plot(xEl, yEl, 'o' , color = colorTable[iEl], markersize = myAtomSize)

        if not vec is None:
            newVec = vec.copy()
            newVec = newVec.reshape(-1,self.dimension)
#             print np.shape(newVec), np.max(newVec)
            u = newVec[:,0]
            v = [0 for _ in u]

            plt.quiver(x,y,u,v,scale = scaleVec)

#         itLen = len(self.structList)
        plt.xlabel("x [A]")
        plt.ylabel("")
        plt.yticks([0])

        plt.xlim(np.min(x)-5,np.max(x)+5)
        plt.ylim(np.min(y)-10,np.max(y)+10)

        plt.show()

    def plot2d(self, **kwargs):
        # define default parameters
        options = {
                    "vec"      : None,
                    "scaleVec" : 1,
                    "atomSize" : 20
                    }
#         print "kwargs2: ", kwargs
        options.update(kwargs)
#         print "args:", options
        if (len(options.keys())) < len(kwargs.keys()):
```

```
            abort("plot2d", err = "argument does not exist!")
        if not self.dimension == 2:
            abort("structure.plot2d",
                  err = "implemented only for dimension = 2 but dim = " +
                        str(self.dimension))

        vec = options["vec"]
        scaleVec = options["scaleVec"]
        atomSize = options["atomSize"]

        x = self.coordinates[:,0]
        y = self.coordinates[:,1]
        xMin,yMin = [x.min(),y.min()]
        nx,ny = self.super_cell_size[0:2]
        x = (x - xMin) * nx
        y = (y - yMin) * ny

        plt.figure()
        plt.subplot(111,aspect = 'equal')
        plt.plot(x, y, 'o' ,markersize = atomSize)
        if not vec is None:
            newVec = vec.copy()
            newVec = newVec.reshape(-1,self.dimension)
#            print np.shape(newVec), np.max(newVec)
            u = newVec[:,0]
            v = newVec[:,1]
#            print "u: ", u
#            print "v: ", v
            plt.quiver(x,y,u,v,scale = scaleVec)

#        itLen = len(self.structList)
        plt.xlabel("x")
        plt.ylabel("y")

        plt.xlim(np.min(x)-5,np.max(x)+5)
        plt.ylim(np.min(y)-5,np.max(y)+5)

        plt.show()


    def plot3d_redraw(self,scale_radius = None, show_bonds = None, bond_radius = None ):
        if not(self.plot3d_initialized):
            print "new plot3d initialization"
            self.plot3d(scale_radius, show_bonds, bond_radius)

        if scale_radius == None:
            scale_radius = self.scale_radius
        if show_bonds == None:
            show_bonds = self.show_bonds
        if bond_radius == None:
            bond_radius = self.bond_radius

        self.setAbsolute()
        x = self.coordinates[:,0]
```

```
        y = self.coordinates[:,1]
        z = self.coordinates[:,2]

        rad = self.ElementList.CovalentRadius()
        radScale = self.scale_radius * rad

        elList = self.ElementList.getIntIndex()

        self.ms.reset(x=x,y=y,z=z,u=radScale,v=radScale,w=radScale,scalars = elList)


    def plot3d(self, scale_radius = 1, show_bonds = True, bond_radius = 0.1, modal = False):
        from enthought.mayavi import mlab

        self.scale_radius = scale_radius
        self.show_bonds = show_bonds
        self.bond_radius = bond_radius

        mlab.figure(1, bgcolor=(0, 0, 0))
        mlab.clf()
        self.setAbsolute()

        x = self.coordinates[:,0]
        if self.dimension == 1:
            y = 0. * x
        else:
            y = self.coordinates[:,1]
        if self.dimension <= 2:
            z = 0. * x
        else:
            z = self.coordinates[:,2]

        rad = self.ElementList.CovalentRadius()
        radScale = self.scale_radius * rad

        elList = self.ElementList.getIntIndex()

        pts = mlab.quiver3d(x, y, z, radScale, radScale, radScale, scalars=elList, resolution =
        10, mode = 'sphere')
        pts.glyph.color_mode = 'color_by_scalar'
        pts.glyph.glyph_source.glyph_source.center = [0, 0, 0]

        if show_bonds:
            self.refresh_neighbors(radius=2, include_boundary = False)
            connections = list()

            for i in range(len(self)):
                nbr = self.neighbor_index[i]
                rad_i = rad[i]
                for j, j_nbr in enumerate(nbr):
                    if not(i==j_nbr):
                        max_dist = rad_i + rad[j_nbr]
                        dist = self.neighbor_distance[i][j]
                        if dist < max_dist:
```

```
                            connections.append((i, j_nbr))
            pts.mlab_source.dataset.lines = np.array(connections)

            # Turn of clamping: the size of the glyph becomes absolute
            pts.glyph.glyph.clamping = False

            tube = mlab.pipeline.tube(pts, tube_radius = bond_radius)
            tube.filter.radius_factor = 1.
            tube.filter.vary_radius = 'vary_radius_by_scalar'
            mlab.pipeline.surface(tube, color=(0.8, 0.8, 0))

        self.ms = pts.mlab_source
        self.plot3d_initialized = True
        if modal: mlab.show()

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

        dialog.show()
        print "test show"
#        sys.exit(app.exec_())
        app.exec_()
        print "test sys"
        return



class CrystalStructure:
    def __init__(self,
                 element = "Fe",
                 BravaisLattice = 'cubic',
                 BravaisBasis = 'primitive',
                 LatticeConstants = [2.],  # depending on symmetry length and angles
                 Dimension = 3
                 ):
        self.BravaisLattice = BravaisLattice
        self.BravaisBasis = BravaisBasis
        self.LatticeConstants = LatticeConstants
        self.Dimension = Dimension

        bravais_lattice_types = ["cubic"]
        if not(BravaisLattice in bravais_lattice_types):
            abort("Bravais lattice: " + BravaisLattice + "not defined!")
            bravais_basis_types = ["primitive", "fcc", "bcc"]
            if not(BravaisBasis in bravais_basis_types):
                abort("Bravais basis: " + BravaisBasis + "not defined!")


        self.amat = r([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])
        if self.Dimension == 3:
            if self.BravaisLattice == "cubic":
                alat = self.LatticeConstants[0]
                self.amat =  alat * r([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])
```

```
            if self.BravaisBasis == "primitive":
                self.coordinates = r([[0.,0.,0.]])
            elif self.BravaisBasis == "bcc":
                self.coordinates = r([[0.,0.,0.],[0.5,0.5,0.5]])
            elif self.BravaisBasis == "fcc":
                self.coordinates = r([[0.,0.,0.],[0.5,0.5,0],[0.,0.5,0.5],[0.5,0.,0.5]])
        elif self.Dimension == 2:
            if self.BravaisLattice == "cubic":
                alat = self.LatticeConstants[0]
                self.amat =  alat * r([[1.,0.],[0.,1.]])
            if self.BravaisBasis == "primitive":
                self.coordinates = r([[0.,0.]])
        elif self.Dimension == 1:
            alat = self.LatticeConstants[0]
            self.coordinates = r([[0.]])
            self.amat =  alat * r([[1.]])

        elList = len(self.coordinates) * [element]
        self.ElementList = ElementList(elList)


    def getCell(self):
        return self.amat

    def getCoordinates(self):
        return self.coordinates

    def getAtomStructure(self):
        return AtomStructure(elementList = self.ElementList,
                             coordinates = self.coordinates,
                             amat = self.amat,
                             tag = "Crystal",
                             rel = True ,               # coordinates are given in relative
                             lattice units
                             pbc = [True, True, True][0:self.Dimension]
                             )


if __name__ == '__main__':

    els1 = ElementList(["Ga","As","Ga"])
    els1.addTags({"rel":[0,1]})
    els2 = ElementList(["Si","As"])

    els = els1 + els2
    print "speciesDict: ", els.dictSpecies
    print "len: ", len(els)
    print "intIndex: ", els.getIntIndex()
    print "Number of species: ", els.getNumberOfSpeciesAtoms()
#    elsSelect = els.copy()
    print "sel: ", els1.selectIndex(["Si", "rel"])
#    print els.dictIndex

#    exit()
```

```
    alat = 4.
    basis1 = CrystalStructure(element = "Ga",
                              BravaisBasis = 'fcc',
                              LatticeConstants = [alat]).getAtomStructure()
    basis2 = CrystalStructure(element = "As",
                              BravaisBasis = 'fcc',
                              LatticeConstants = [alat]).getAtomStructure()


    basis2.shift([1./4,1./4,1./4],rel=True)
    basis = basis1+basis2
    print basis.ElementList.getSymIndex()
    print basis.ElementList.selectIndex("Ga")
    print basis.ElementList.selectIndex("As")
    basis.ExtractSpecies(["As"])
    print basis.ElementList.getIntIndex()
#    basis.print_short()
#    exit()

    basis.repeat([2,2,2])
    basis1.print_short()
    print "mass: ", basis1.AtomicMassDOF()

    file = tables.openFile("test.h5", mode = "w", title = "structure test")
    root = file.root
    basis.writePyTables(file, root)
    file.close()

    file = tables.openFile("test.h5", mode = "r")
    mygroup = file.root
    newBasis = AtomStructure()
    newBasis.loadPyTables(file, mygroup)
    file.close()

    newBasis.print_short()
#    exit()

    print "spacegroup:", basis.getSpacegroup()
    sym_ops = basis1.getSymmetry()
    print "Number of symmetry operations: ", len(sym_ops['translation'])
#    for i,trans in enumerate(sym_ops['translation']):
#        print 'operation: ',i
#        print trans
#        print sym_ops['rotation'][i]

    primitiveCell = basis.getPrimitiveCell()
#    basis.showGUI()
    primitiveCell.print_short()

    refinedCell = basis.getRefinedCell()
    print "refined cell: "
    refinedCell.print_short()
```