Software Spec – Cluster class

I want two ways to populate Species – one by explicit specification, another by automatic parsing of cluster. I’m thinking that might be an offline tool used to construct the ClusterXML file that is used to create the Cluster, though. Let’s use that strategy – Species parameters are already determined.

# Public Class Cluster

Public Readonly Property Count()

Returns the number of atoms in the cluster.

Public Overloads Function AddAtom(x as double, y as double, z as double, species as Integer) as Integer

Public Overloads Function AddAtom(position as Vector, species as Integer) as Integer

Adds an atom of the specified species to the cluster at the given position. Returns the (unique) ID of the added atom. If speciesID is not in the collection, throws an InvalidIDException or equivalent.

Possible issue – what happens if two positions overlap or are the same?

Public Property AtomEnabled(atomID as Integer) as Boolean

Gets/Sets the enabled property of the specified atom. If atomID is not in the collection, throws an InvalidIDException or equivalent.

Public Sub RemoveAtom(atomID as integer)

Removes the specified atom from the cluster. If atomID is not in the collection, throws an InvalidIDException or equivalent.

Public Default Readonly Property GetAtom(atomID as integer) as ClusterAtom

Returns a ClusterAtom structure with information about the specified atom. The ClusterAtom structure contains all relevant information about the atom:

AtomID, SpeciesID, Position, Z, Configuration, IMuffinTin, Rmt, PhaseShifts(L), Tmatrix(L)

Public Property Species(atomID as Integer) as Integer

Gets/Sets the species of the specified atom.

Public Overloads Function AddSpecies(Z as Element, configuration as string, Rmt as Double) as Integer

Public Overloads Function AddSpecies(Z as Element, Rmt as Double) as Integer

Adds a Species definition to the cluster, and returns the (unique) ID of the species.

If configuration is invalid, throw an InvalidConfigurationException or equivalent.

If Rmt is <= 0 then throw an exception.

If configuration is omitted, assume the default configuration for the element.

Public Function GetSpecies(speciesID as Integer) as ClusterSpecies

Returns a ClusterSpecies structure containing information about the species of the given ID. This method is used by external objects to create muffin-tin potentials and phase shifts.

The ClusterSpecies structure contains atomic number, Rmt, configuration

Public Sub RemoveSpecies(speciesID as Integer)

Removes the specified species from the collection. If speciesID is invalid, throw an InvalidIDException or equivalent.

Public Property PhaseShifts(speciesID as integer) as Complex()

Gets/Sets the PhaseShift structure, containing phase shift and muffin-tin potential information about the atoms. (A PhaseShiftCalculator class encapsulates the strategy used to compute the ClusterPhaseShift for the species).

Returns an array of Complex values containing the phase shifts for the specified atom. The length of the array is dependent upon the Lmax defined in the PhaseShift object for the associated species.

Public Function GetTmatrix(atomID as Integer) as ComplexMatrix

Returns a reference to the T-matrix of the specified atom. If no phase shifts have been calculated, returns Nothing. If atomID is not in the collection, throws an UnknownIDException or equivalent.

Public Function GetNearestNeighbors(atomid As Integer, rmax as double) As SortedList(Of Integer, Double)

Returns a list of atomIDs and distances, from the given atom for nearest neighbors, sorted by distance. Useful for some implementations of PhaseShiftCalculator, when computing species based on MattheissMuffinTins.

Public Function HalfNeighborRadius(atomid As Integer) As Double

Returns one half of the distance to the first nearest neighbor for the specified atomID. This can be different from the Rmt in the associated species.

Public Function GetAtomIDs() As List(Of Integer)

Returns a list of all member atom IDs in the cluster.

Public Function GetSpeciesIDs() As List(Of Integer)

Returns a list of all member species IDs in the cluster.

Public Function GetPosition(atomid As Integer) As Vector

Returns the position of the specified atom in the cluster.

Public Function GetPathLength(path As MSPath) As Double

Returns the total length of the specified path (list of IDs), in Angstroms (? Or Bohr Radii?)

## Sub New(clusterxml as xmldocument, Emin as Double, Emax as Double)

Internal datatable stores cluster data.

Atomid, atomicnumber, x, y, z, speciesID, Rmt

SpeciesID is based upon *equivalent sites*, determined by distance and chemical species of nearest neighbors out to (some number) of shells. When constructing MattheisMuffinTins consider out to 2nd nearest neighbor (2nd shell). When constructing PendryMuffinTins, only Rmt and atomicnumber are considered.

# MSPath

A path has a sequence of atom IDs, the distance between those IDs, and the total path distance. Is a member of a parent SortedList which ranks paths by the total path distance, so the IComparable “value” of a ClusterPath is its distance.

ClusterPaths should implement the “+” operator to append.

## Public Function Append(atomid as integer) as ClusterPath

Returns a new ClusterPath instance that has simply appended the given atom to the path instance.

## Public Readonly Property Length() as Double

## Public Function Beta(atomid1 as integer, atomid2 as integer, atomid3 as integer) as double

Returns the (polar) Euler angle between the two bond vectors atom1 – atom2 and atom2 – atom3. Atom2 is considered to be the scattering site.

## Public Readonly Property Order as Integer

Returns the order of scattering (i.e., the number of elements in the path – 1).

# Creating ClusterPaths – Path Enumeration

There are three main modes of calculation:

1. DESD – paths have arbitrary starting points but a single termination point (inverted tree)
2. PED – paths have a single starting point but arbitrary termination points
3. LEED – paths have arbitrary starting and ending points.

This code will be optimized for DESD. PED paths are easy to create by simply inverting the order of atoms in a DESD path.

A calculation should specify a maximum path length (dmax). The routine that creates paths dynamically should compare each new path’s d against the limit and discard any that exceed.

For DESD, the best strategy is to work backwards:

1. Add the terminal atom to the path Path = New ClusterPath(terminalatomid)
2. Add this path to the List (this path represents the direct amplitude, and has Order = 0)
3. For each Path in the List (need to run over keys here, we’ll be modifying the list)
   1. NewPath = Path.Append( the next atomID in the cluster list AtomIDs) (need a Cluster.GetAtomIDs method that returns an array)
   2. If newpath.Length < Dmax then
      1. List.Add newpath
      2. Set “AddedNew” flag to TRUE
   3. End if
4. Recurse until AddedNew = False

Problem with this algorithm is that it will reproduce paths.

Better Idea – build by orders:

First create all paths of order 1, then those of 2, then those of 3, etc.

Can’t I just do this numerically? This is just combinatorics

Paths are integer arrays:

1, 0, 9, 5, 0

We could enumerate the paths and THEN calculate amplitudes, or interleave the two processes. This allows the cutoff to be on amplitude and not total path length, and will help weed out right-angle scattering paths.

For example, appending an atomID to a path triggers computation of the Feff term for the N-1 atom:

1,0 append 2 = 1, 0, 2, triggers computation of Feff for the 1->0->2 leg

If we go that way, then this isn’t a ClusterPath, this is a ScatteringPath.

ScatteringPath has:

* Total distance
* Total Amplitude

I like the interleaving method, because it doesn’t add to a path unless it’s got sufficient amplitude to warrant it.

This means that the MC coordinating the dance is the DESDCalculation class. Come to think of it, perhaps it is the best place to keep the scattering phase shifts as well, and have the Cluster just serve up the coordinates and atomic type of the constituent atoms.

# Scattering Matrices

Rehr/Albers computes Feff for a scattering site as a function of the bond lengths incoming and outgoing, and the angle between them. So each time we calculate an Feff, we store it indexed by the bond lengths and the angle. Heck, it might even be possible to parameterize it, so that for each scatterer type we interpolate over the table:

(Rho1 Rho2 Beta) Feff(6x6)

The “key” in this collection is the Rho1 / Rho2 / Beta combination. So by storing the results of each combination in a lookup, we can avoid repeat computations of the same value. Create a structure for the composite key that implements IComparable and the Equals operator, then we can use LINQ or Collection.ContainsKey to find the element. If it doesn’t exist, then compute it and add it to the collection.

Another strategy is to keep track of parent and child paths, in a tree-like structure, and keep track of composite amplitudes for the whole path. When computing a child path, start with the parent amplitude and work outward. Would be more complicated to code and might not save much.