EASAL User/Developer Manual

Aysegul Ozkan, Rahul Prabhu, Ruijin Wu, Troy Baker,
James Pence, Jorg Peters, Meera Sitharam
University of Florida

January 2, 2018

This is a user/developer guide for the EASAL software described in the accompanying TOMS paper. EASAL generates, describes the topology, and explores the configuration space of two point sets in $\mathbb{R}^3$ that are mutually constrained by distance intervals. Technical concepts and definitions that are required to use this software can be found in the main paper.

1 Introduction

EASAL generates the assembly configuration space of two point sets in $\mathbb{R}^3$ and describes the key aspects of the topology and geometry of of this space [4, 2, 5]. EASAL implements algorithms that use new theoretical results, some of which are presented in [2, 3, 4]. EASAL is opensource and can be downloaded from `https://www.bitbucket.org/geoplexity/EASAL`.

This user guide describes the TOMS version of EASAL. The TOMS version is the backend of EASAL without the GUI and with text input and output. Only this version of EASAL is part of the TOMS submission. The experimental results in Section 4.1 of the paper can be reproduced with this version using the sample input files given in the files directory (see Section 5 for instructions).

This user guide describes in depth the key conceptual functionalities, dependencies and installation, and the major classes and methods in EASAL. A video presenting the theory, applications, and software components of EASAL is available at `http://www.cise.ufl.edu/~rprabhu/EASALvideo.mpg`.

An optional GUI (not part of TOMS submission) has been included for intuitive visual verification of the results. Instructions on how to install, how to use and major functionalities offered by the GUI have been detailed in the 'Complete User Guide'. The source code and the 'Complete User Guide' are present in the EASAL github respository which can be found at `https://www.bitbucket.org/geoplexity/EASAL`.

Organization: The rest of the user guide is organized as follows. Section 2.1 discusses the software dependencies and installation instructions, Section 3 discusses the input and output, Section 4 discusses the functionalities offered by the backend. Section 5 gives an example test driver and Section 6 describes the major classes and methods in EASAL to help developers gain an insight into how EASAL has been implemented.

2 Dependencies and installation

2.1 Dependencies

This section discusses the software dependencies of EASAL and gives installation instructions.

- Operating System: EASAL has been tested only on the following platforms:
  - Ubuntu 12.04 or higher.

  - Fedora 22 or higher.
  - OS X 10.8 Mountain Lion.

  EASAL should work on any UNIX variant platform with little to no modifications.

- We use Version 2.0 of the *Eigen* library for linear algebra computations. All necessary files pertaining to Eigen required by EASAL are provided with the source code in the *include* directory.
- We use *simpleini* to read the settings from the settings.ini file. All necessary files pertaining to simpleini are provided with the source code in the include directory.
- C++ compiler: EASAL requires one of the following compilers
  - g++ Version 4.8 or higher.
  - clang++ Version 3.8 or higher.
- EASAL uses the GNU Make utility to compile the source files. Make Version 4.1 is required.

2.2 Installation

- Install GNU Make
  - On Ubuntu
    - sudo apt-get install make
  - On Fedora
    - yum install make
  - On OSX
    - sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer
- To build the software, run *"make backend"* in the build/root directory.
- To run EASAL run *"bin/EASAL"* in a terminal from the root/build directory.

Before giving the test driver details in Section 5, we first describe the input, output and software functionalities.

3 Input/Output

3.1 Input

Input to EASAL is specified using the settings.ini file. The main input features are the following:

- **Two rigid point sets**: *[PointSet(A/B)]* fields are used to specify the two point sets. The file subfield points to a file that specifies the input data in the pdb format.
- **Constraints**:
  - **Active threshold**: This specifies the range of distances where constraints are considered active. This is given as $\lambda * (r_i + r_j) \pm \delta$. Here, λ is specified by the *activeLowerLambda* subfield and δ is specified by the delta text box in the input window. r_i and r_j are the radii of the spheres in the point sets.
  - **Collision threshold**: This is the minimum distance between the points. This too is given as $\lambda * (r_i + r_j) \pm \delta$. The subfields *collisionLambda* and *collisionDelta* specify these values.
  - **Distance Data**: Only when the constraints between these pairs are active, are the corresponding configuration space regions explored.

3.2 Output

The following are the output:

- The *Roadmap*, which stores the atlas, i.e., a topologically stratified set of sample feasible realizations or configurations of the two rigid point sets. This can be found in the 'RoadMap.txt' file in the data folder.
- The *Node* files which contain sampling information, Cayley parameter values, and realizations of the point sets. Each 'Node*.txt' file contains samples for a particular active constraint region.
- The *paths* file which contains the one degree of freedom motion path between all pairs of lowest energy configuration regions. This can be found in the 'paths.txt' file in the data folder.
- The *path matrix*, which contains a path matrix where the rows and columns correspond to 0D and 1D nodes. The $\{ij\}^{th}$ entry indicates the number of paths between nodes i and j. This can be found in the 'path_matrix.txt' file in the data folder.

4 Software Functionalities

4.1 Active Constraint Graph

The active constraint graph for each node in the atlas can be found in the 'RoadMap.txt' file. This active constraint graph shows only the participating points from each set. A 'c' before two points indicates a constraint and a 'p' indicates a parameter between two points belonging to different sets. It has to be noted that the points in the same set already form a clique.

4.2 Finding Boundary Regions

The boundary regions of a particular node of the atlas can be found in the 'Roadmap.txt' file. In terms of the atlas, it depicts the the ancestors and descendants of that node. Since and edge in the graph represents boundary relationship, this feature allows us to inspect the boundary regions of an active constraint region.

4.3 Finding Paths Between Nodes with 6 Active constraints

The atlas output by EASAL can be used to generate all the paths between any two active constraint regions along with their energies. Once the atlas has been generated, finding paths is extremely fast. In the context of molecular assembly, the path topology of the configuration space is crucial for understanding assembly kinetics.

Of particular interest is finding paths between two lowest energy configurations with 6 active constraints or 0D nodes of the atlas with effectively rigid configurations. We are mainly interested in paths through active constraint regions that have 5 or 6 active constraints. One fewer constraint implies one step higher energy and such paths represent a continuous one degree-of-freedom motion.

EASAL finds the shortest paths between all pairs of 0D nodes, if they exist. EASAL writes this path as a series of nodes to the 'paths.txt' file. Once the sampling has been completed, EASAL computes the total number of paths of a particular length between every pair of vertices and writes the entire matrix to the 'path matrix.txt' file. The user can specify lengths by setting the 'path length' parameter in the 'settings.ini' file.

4.4 Sampling Options

There are two options available:

- **Default**: The default mode of sampling is the auto-solve mode, which samples the atlas in a depth-first fashion. EASAL generates, depending on the user input, all possible 4D and 5D root nodes. EASAL then recursively samples these nodes till the atlas generation is complete.
- **BFS**: Setting the *breadthFirst* subfield under '[AtlasBuilding]' to true forces EASAL to explore the atlas in a breadth-first fashion. Otherwise depth-first is used.

4.5 Dimension of the root node

By default, the dimension of the root node of the atlas is 5, which means that the root node has one bond and 5 parameters. The *initialContactGraphs* option under '[RootNodeCreation]' allows the user to change this and make the root node a 4D node. With a 4D root node, there are 2 bonds and 4 parameters in the root node.

4.6 Step Size

EASAL uses Cayley grid sampling to sample the Cayley space. The user can specify this using the *stepSize* option under '[Sampling]' in the settings file. Another option available to the user is to choose dynamic step size. This option can be selected by setting the *dynamicStepSizeAmong* under '[Sampling]' to true. Doing so tells EASAL to run the different dynamic step size variants of EASAL. Setting *dynamicStepSizeWithin* to 0 runs EASAL-1, setting it to 2 runs EASAL-2 and setting it to 1 runs EASAL-3.

5 Test Driver

To run the software, run the following command from the top directory: 'bin/EASAL -settings <settings file name>'. Where <settings file name> is the path to the settings.ini file in which the input has been specified. All input to the backend version of EASAL is specified using this settings.ini text file.

We have included two example test drivers and all necessary files for the reviewers to run and test the program. To run these, just run the following commands from the top directory.

'bin/EASAL -settings settings example 1.ini'

'bin/EASAL -settings settings example 2.ini'

Using the test drivers, the results corresponding to 'Atlasing and Paths' (Section 4.1 in the accompanying TOMS paper) can be reproduced. The 'settings example 1.ini' test driver runs the experiment with n = 6 example with step set to 0.5 times the smallest radius and the tolerance set to (1.0 - 0.75) * sum of radii. This corresponds to the third row in Table I in the TOMS paper. The 'settings example 2.ini' runs the experiment with n = 20 with tolerance set to (1.0 - 0.75) * sum of radii and the step size set to 0.25 times the smallest radius. This corresponds to the fourth row in Table I in the TOMS paper.

The output of the run can be found in the 'dataDirectory' specified in the input settings file. For instance, this directory is 'Driver1data' for example 1 and 'Driver2data' for example 2. The results are as explained below.

1. Generating the atlas: The number of samples and the time required for sampling the entire atlas can be found in the 'Samples.txt' file in the data directory.
2. Finding paths between active constraint regions: As mentioned earlier EASAL finds the shortest path between all pairs of 0D nodes, if it exists, and writes this path as a series of nodes to the paths.txt file in the data folder. Once the sampling has been completed, EASAL computes the total number of paths of a particular length between every pair of OD regions and writes the entire matrix to the path

matrix.txt file in the data folder. The user can specify lengths by setting the path length parameter in the settings.ini file. These results correspond to table II in the TOMS paper.

3. Finding Boundary Regions: The boundary regions of any active constraint region can be found using the 'RoadMap.txt file in the data folder. The 'Nodes this node is connected to' filed in this file lists all the boundary regions.

Sample results for these test drivers have been included in the 'SampleOutput' folder in the top directory. They contain the following files:

1. RoadMap.txt - Contains information about the stratification and the boundary regions.
2. paths.txt - Contains the shortest paths between every pair of 0D nodes, if it exists.
3. path matrix.txt - Contains the number of paths between 0D and 1D nodes.
4. Samples.txt - Information about number of samples and time required for sampling.
5. Node0.txt - One typical randomly selected node file containing sampling information along with realizations.

Once the test drivers have been run, the optional GUI (not part of the TOMS submission) can be used to visualize the results. See Section 2.4 of the complete user guide for instructions on how to visualize the results using the GUI. For full use of the GUI, see Section 3 of the 'Complete User Guide' which can be found at `https://www.bitbucket.org/geoplexity/EASAL`.

6 Major Classes of EASAL

Figure 1 gives an overview of the structure of the major classes of EASAL. Each of these classes are explained in the subsections below.

6.1 AtlasBuilder

The AtlasBuilder class populates the ActiveConstraintRegion for each activeConstraintGraph by sampling inside the boundaries of its ConvexChart. It creates and explores only regions that contain at least one Cartesian realization.

Major Attributes:

- **rootGraphs**: The set of all possible 4D or 5D ActiveConstraintGraphs of the root nodes generated before sampling.
- **atlas**: An atlas object that is populated by the AtlasBuilder by sampling. This object is shared between front-end and back-end of the algorithm.

Major Methods:

- **startAtlasBuilding()**: For each of the generated root graph, creates an atlasNode labeled with a contact graph G_F where F is the set of contacts. Then calls the recursive sampleTheNode method for each of the root atlas nodes.
- **sampleTheNode(atlasNode)**: The exploration of the atlas is done by the recursive **sampleAtlasNode** algorithm (see Algorithm 1) using one of the generated atlas root nodes as input. This algorithm is implemented by the sampleTheNode method. Using depth first search this algorithm samples the atlas node and all its descendants.

  Base case of recursion: If active constraint graph G_H of the node is minimally rigid i.e., the active constraint region is 0-dimensional, we have no more sampling to do, return.

```
sampleAtlasNode
input  : atlasNode: node
output: Complete sampling of the atlasNode and all its children
H = node.activeConstraints
G_H = node.activeConstraintGraph
if G_H is minimally rigid then
| stop;
end
F = complete3Tree(G_H)
C = computeConvexChart(G_H, F)
for each cayleyPoint p within convexChart C do
| R = computeRealizations(p)
| for each realization r in R do
| | if !aPosterioriConstraintViolated(r) then
| | | if isBoundaryPoint(r) && hasNewActiveConstraint(r, G_H) then
| | | | e = newActiveConstraint(r, G_H);
| | | | G' := G_H ∪ {e} ;
| | | | if G' is not already present in the current atlas then
| | | | | childNode = new atlasNode(G')
| | | | | sampleAtlasNode(childNode);
| | | | end
| | | | else
| | | | | childNode = findNode(G');
| | | | end
| | | | node.setChildNode(childNode);
| | | end
| | end
| end
end
```

Algorithm 1: High level EASAL pseudocode

The recursion step: If G_H is not minimally rigid, we use the **complete3Tree** algorithm to find find a set of parameters F so as to form a maximal 3-tree to leverage the convex parametrization theory [3]. This also ensures that $H \cup F$ is minimally rigid and easily realizable.

The method computeConvexChart shown in the pseudocode finds the convex chart for the parameters F is done by the ConvexChart class explained later. Method ComputeRealizations computes the realization for a Cayley point and is done by the findRealizations method in the software. The aPosterioriConstraintViolated method which checks for angle and steric violations is implemented in the ConstraintCheck class explained later. Next we make a call to the findBoundary to detect boundaries and newly active constraints.

- **determineStepSizeDynamically()**: Finds out the step size s given T, the total number of samples. Each 5D atlas node has its own s computed by using the volume of the Cayley parameter space of the node over total number of samples per node. The volume of the Cayley parameter space of the node is approximately computed by exhaustive sampling within the exact chart without considering any constraints. The number of samples per node roughly can be computed by T over total number of root(starter) atlas nodes, m. The number of samples in child nodes are negligible since the volume of regions in low dimensional nodes are negligible compared to the regions of high dimensional nodes.

- **findBoundary()**: Boundary detection ensures that sampling stays in the feasible region and minimizes discarded samples. The findBoundary method which detects boundary points, checks for newly formed active constraints and makes function calls which in turn call sampleTheNode for a child region.

6.2 Atlas

The 'Atlas' class stores the directed acyclic graph that represents the relationship between active constraint regions.
Major Attributes:

- **nodes**: A vector of all AtlasNodes.

- **rootIndices**: The indices of all the root nodes in the atlas.

Major Methods:

- **search(node)**: Uses depth first search on the atlas to check whether the node exists in the atlas or not. It is used to avoid repeated sampling of the same region. The time complexity of the search is $O((\text{depth of the tree})) = O(6(k-1))$ which in our case is $O(1)$ since we fix k to be 2.

6.3 AtlasNode

AtlasNodes make up the Atlas. Each AtlasNode represents an active constraint region reperesented by an ActiveConstraintGraph.
Major Attributes:

- **acg**: The active constraint graph corresponding to the node.

- **region**: The set of Cayley points in the active region.

- **connection**: The id of the nodes in the atlas that represent the boundary of this node's region.

6.4 ActiveConstraintGraph

The ActiveConstraintGraph class is used to store the set of active constraints.
Major Attributes:

- **activeConstraints**: The set of point index pairs that represent contacts.
- **verticesA**: Participating points from first point set.
- **verticesB**: Participating points from second point set.
- **parameters**: A vector of point index pairs that represent parameters.

Major Methods:

- **completeTo3by3Graph()**: Adds points to make sure there are at least 3 points from each point set so that the graph is realizable. While choosing additional points, it has 2 options, choosing the points closest to each other or the points that lead to a user specified angle.

6.5 ActiveConstraintRegion

The ActiveConstraintRegion class contains the set of feasible Cayley points generated by sampling.
Major Attributes:

- **space**: The set of feasible Cayley points.
- **witspace**: The set of feasible witness Cayley points obtained from an ancestor node.

Major Methods:

- **convertSpace(activeConstraintRegion)**: Re-parametrizes a region using an input regions parameters. This method converts each Cayley point in the input activeConstraintRegion to the Cayley point parametrized by the input region's parametrization.

6.6 CayleyPoint

The CayleyPoint class represents a multi-dimensional point in the Cayley parameter space and stores the corresponding Cartesian space orientations of the point set.
Major Attributes:

- **data**: Values of the Cayley parameters (non-edge lengths).
- **orients**: The set of Cartesian space Orientations of the point set that were computed by realizing the active constraint graph with the given length of the edges and non-edges.

6.7 Orientation

The Orientation class is the Euclidean transformation of point set. The Orientation class stores only the information necessary to compute the transformation matrix that will yield a Cartesian realization for the entire point set.
Major Attributes:

- **FromB**: Cartesian coordinates of three points from the first point set before the transformation.
- **ToB**: Cartesian coordinates of three points from the second point set after the transformation.
- **connections**: The set of node indices that this orientation belongs to. An orientation be on the boundary of multiple regions.

6.8 CayleyParameterization

The CayleyParameterization class chooses non-edges in an ActiveConstraintGraph that convert the graph into complete 3-tree. Those non-edges are called the parameters. The complexity of the sampling algorithm varies based on the choice of non-edges and the order in which they are fixed.
Major Attributes:

- **partial3tree**: A boolean variable indicating whether an ActiveConstraintGraph is partial 3-tree or not.
- **parameters**: The set of points pairs that represent non-edges.
- **tetrahedra**: The ordered tetrahedron set that helps in defining the order of parameters that is required during the sampling procedure. This data is later passed to ConvexChart and CartesianRealizer to help in their computations.
- **updateList**: Adjacency map containing the dependency of parameters. It provides the set of parameters whose range will be updated when one of the parameters is fixed.
- **boundaryComputationWay**: Inequalities that express the range of a parameter can be classified into either a linear or non-linear class. This variable is the characterization of the parameter that tells what inequality is needed to compute the parameter range i.e., triangular or tetrahedral inequality.
- **complete3trees**: The set of complete 3 trees.

Major Methods:

- **defineParameters()**: The parameters of an active constraint graph are selected as maximal 3-realizable (3-tree) extensions by leveraging the convex parametrization theory. [3]. It creates a look-up table containing all possible complete 3-trees. We find a graph in the look-up table so that active constraint graph is a proper subset of either the graph or one of its isomorphisms.
- **parameterMinDeviation()**: An alternate way to pick the parameters for 5D regions is by ensuring that the range of each parameter is similar. The aim here is to sample more uniformly in the Cartesian space.
- **built3tree()**: The 3-tree formed by starting with a 4-vertex complete graph and repeatedly adding vertices in such a way that each added vertex is edge-connected to the face of a tetrahedron. Store the tetrahedrons in the order they are created in the attribute **tetrahedra**.

6.9 ConvexChart

The ConvexChart class is used to determine the chart that parameterize the regions i.e., it computes the range of parameters of ActiveConstraintGraph. An exact convex chart yields feasible Cayley points for the current active constraint region. The resulting Cayley configuration space is convex, before collisions or other (e.g. angle) constraints are introduced. The range of parameters are computed by triangle and tetrahedral inequalities.
Major Attributes:

- **param_lengthUpper**: The upper bound of the parameters' range
- **param_lengthLower**: The lower bound of the parameters' range
- **param_length**: current value of parameters

Major Methods:

- **initializeChart()**: Initalizes the boundaries of convex chart. Tighter bounds are given in [1].

- **computeRange(v1, v2)**: Computes the range of the parameter $v1-v2$ in order to eliminate sampling infeasible grid points. Range computation is required in every iteration for dependent parameters.
- **setRangeByTriangleInequality(v1, v2)**: Computes the range of the non-edge $v1 - v2$ through triangular inequalities.
- **setRangeByTetrahedralInequality(v1, v2, tetrahedron)**: Computes the range of the non-edge $v1 - v2$ through tetrahedral inequality.
- **stepGrid**: Sets parameter point to the next grid point within the computed range.
- **stepNeighbour()**: Sets the parameter point to the neighbor grid point in all dimensions consecutively.
- **stepGridBinary()**: Sets the parameter point to somewhere between current point and neighbor grid point according to binary search procedure in findBoundary.

6.10 CartesianRealizer

The CartesianRealizer class contains routines that compute orientations that represent transformations of rigid helices relative to each other. It computes Cartesian realization of an active constraint graph with the parameter lengths taken from cayleyPoint and active constraint lengths for a specific flip. **Major Attributes:**

- **positions**: Cartesian coordinates of vertices in ActiveConstraintGraph.
- **edge_length**: Contains all fixed distances plus current distance values of non-edges of ActiveConstraintGraph.

Major Methods:

- **computeRealization(activeConstraintGraph, convexChart, flipno)**: Computes the Orientation by leveraging partial 3-tree techniques. activeConstraintGraph which is a complete 3-tree is built up from a base tethedra by adding, at each step, a new vertex edge-connected to the face of a tetrahedron.
- **setBaseTetra(tetrahedron)**: Finds Cartesian coordinates of the vertices of tetrahedron by known edge lengths.
- **locateVertex(vertex, face)**: Finds Cartesian coordinates of the vertex that is connected to the face of a tetrahedron.

6.11 ConstraintCheck

The ConstraintCheck class is designed to check whether any non-active constraints become active. Users have the option to define a set of constraints of interest. In which case, the new constraint activation check is done only for these. For an input Orientation, ConstraintCheck first computes the Cartesian realization for the entire molecular composite then passes it to other subroutines to perform user specified constraint check such as steric constraints or angle constraints.

References

[1] Ugandhar Reddy Chittamuru. Efficient bounds for 3d cayley configuration space of partial 2-trees. Master's thesis, University of Florida, 2010.

[2] Aysegul Ozkan, Ruijin Wu, Jörg Peters, and Meera Sitharam. Efficient atlasing and sampling of assembly free energy landscapes using EASAL: Stratification and convexification via customized Cayley parametrization. (on arxiv), 2014.

[3] Meera Sitharam and Heping Gao. Characterizing graphs with convex and connected Cayley configuration spaces. *Discrete & Computational Geometry*, 43(3):594–625, 2010.

[4] Meera Sitharam, Aysegul Ozkan, James Pence, and Jörg Peters. EASAL: Efficient atlasing, analysis and search of molecular assembly landscapes. *CoRR*, abs/1203.3811, 2012.

[5] Ruijin Wu, Aysegul Ozkan, Antonette Bennett, Mavis Agbandje-McKenna, and Meera Sitharam. Prediction of crucial interactions for icosahedral capsid self-assembly by configuration space atlasing using EASAL. (on arxiv), 2014.

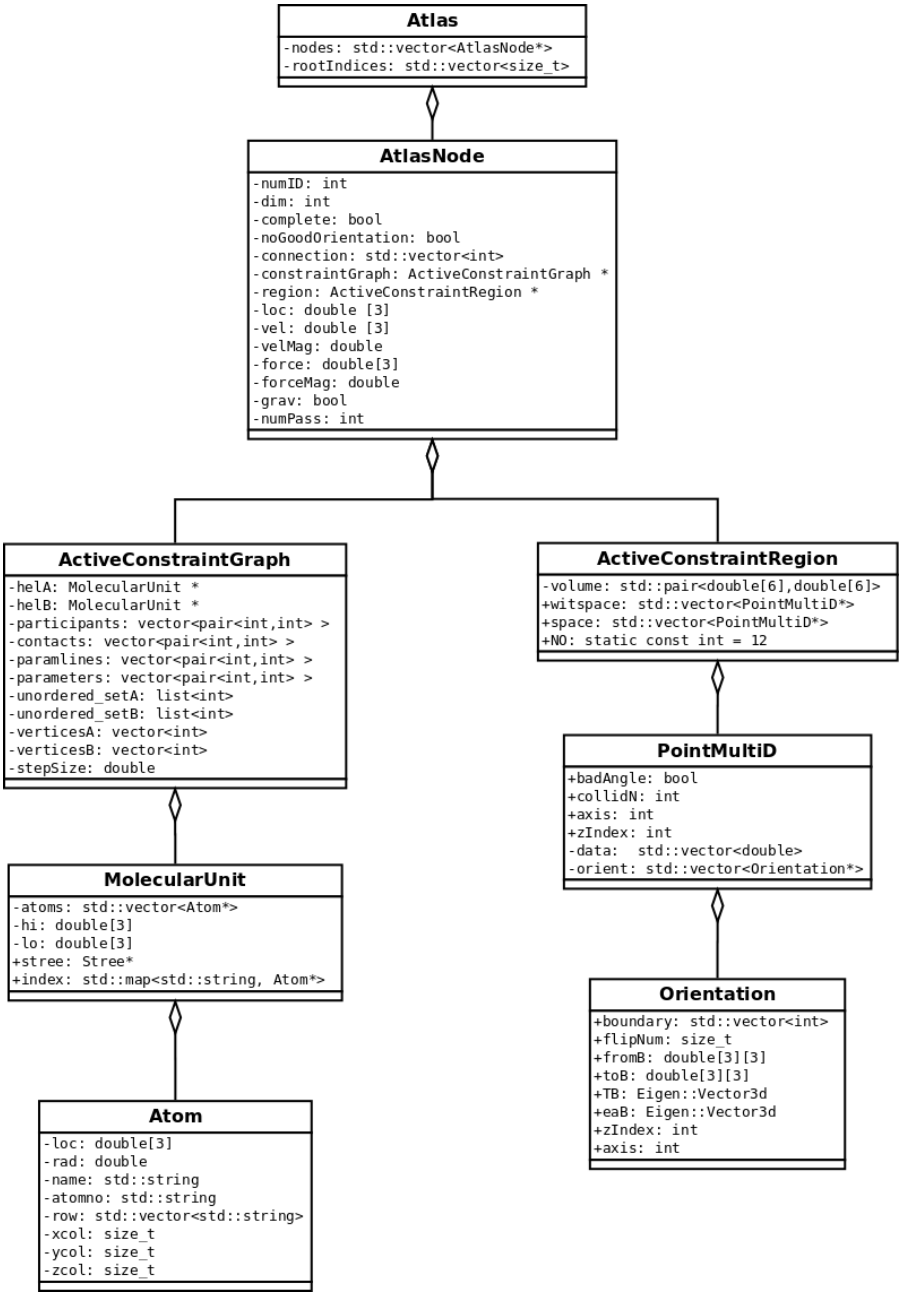

Figure 1: EASAL UML Diagram