JGromacs v1.0.

documentation

Contents

Introduction 3

Installation Guide 4

Subpackages 6

Multi-level representation 7

Quick Start Guide 11

Example 27

Example codes 28

Classes 29

JGromacs user interface 37

Copyright notice 39

Introduction

What is JGromacs?

JGromacs is a Java library designed to facilitate the development of cross-platform analysis applications for Molecular Dynamics (MD) simulations. The package contains parsers for file formats applied by Gromacs (GROningen MAchine for Chemical Simulations), one of the most widely used MD simulation packages. JGromacs provides a multi-level object-oriented representation of simulation data to integrate and interconvert sequence, structure and dynamics information. A basic analysis toolkit is included in the package. The programmer is also provided with simple tools (e.g. XML-based configuration) to create applications with a user interface resembling the command-line UI of Gromacs applications.

JGromacs is an open-source project released under the GPL (GNU General Public License). It is developed by Márton Münz and Dr. Philip C Biggin at the Structural Bioinformatics and Computational Biochemistry Unit at the Department of Biochemistry in the University of Oxford.

Availability of JGromacs

JGromacs and its detailed documentation can be downloaded from our website: http://sbcb.bioch.ox.ac.uk/jgromacs/. We welcome comments, feature requests and improvements. If you have problem downloading, installing or running JGromacs, contact us by writing email to *marton.munz@dtc.ox.ac.uk*.

JavaDoc and examples

Please note that the automatically generated JavaDoc documentation and a library of example codes are also available on the project's website.

Citing JGromacs

A manuscript is currently under review. In the meantime you can simply refer to the project's website.

Installation Guide

Downloading JGromacs

The entire JGromacs package can be downloaded from our website as a single JAR (Java ARchive) file; *jgromacs_v1_bin.jar*. This file contains all binaries of the library.

(Note that the source code of the package can also be downloaded in the gzipped tar file; *jgromacs_v1_src.tgz*.)

In order to make use of the API in your own Java project or to run the example codes or the JUnit test suite available on our website, you will need to add the *jgromacs_v1_bin.jar* file to the classpath you are using.

First of all, download the jar file and place it in a directory where you can access it later. Here we give instructions for how to use the API in your project, compile and run example codes and the test suite.

Important:

The following instructions are valid only if you are compiling/running your code in Linux or Mac OS X. In case you are working in Windows, you will need to replace colon (:) characters with semicolon (;) characters in the commands below!

Using the API

In order to use the API in your own Java project, you only have to add the *jgromacs_v1_bin.jar* file as an external library to the build path of your project. In the commonly used development environments (IDEs) such as Eclipse, this can easily be done using a GUI.

Alternatively, you may compile your application in the command line using the following command:

```
javac -cp XXX/jgromacs_v1_bin.jar:. MainClass.java
```

Here **XXX** represents the path of location where the *jgromacs_v1_bin.jar* package has been downloaded to. MainClass.java is the class to be compiled.

Once compiled, your application can be run using the command:

```
java -cp XXX/jgromacs_v1_bin.jar:. MainClass
```

Alternatively, you can compile and run your application by rebuilding the JGromacs source code library using the following command:

```
javac -sourve 1.5 -cp YYY:. MainClass.java

java -cp YYY:. MainClass
```

Here **YYY** represents the path of location where the *jgromacs_v1_src.tgz* package has been extracted to.

Note that if you make use of JGromacs classes in your code, you need to import the required classes within your .java file. For example, the following command imports the jgromacs.data.Structure class into your source code:

```
import jgromacs.data.Structure;
```

Alternatively, you can import the entire jgromacs.data subpackage:

```
import jgromacs.data.*;
```

Compiling and running the example codes

The library of example codes can be downloaded as a gzipped tar file; *jgromacs_v1_examples.tgz.* This file also contains an example dataset used by the examples included in the package.

As a first step, unpack the tgz file:

```
tar xvzf jgromacs_v1_examples.tgz
```

This will give you the directory *examples/* containing 7 subdirectories.

Subdirectories *examples/data*, *examples/input, examples/output, examples/analysis, examples/advanced* and *examples/ui* contain example java files organized by topic.

In addition, the subdirectory *examples/dataset* contains the sample dataset required for running the example codes.

In order to compile and run an example code (for instance, input/ReadStructures.java), use the following commands in the *examples* directory:

```
javac -source 1.5 -cp XXX/jgromacs_v1_bin.jar:. input/ReadStructures.java

java -cp XXX/jgromacs_v1_bin.jar:. input/ReadStructures
```

Here XXX represents the path of location where the *jgromacs_v1_bin.jar* package has been downloaded to.

(Note that compiling example codes may give a list of warnings about variables that are never read. These warnings can be ignored as these codes are not complete programs but only illustrations of the exact usage of JGromacs methods.)

Running the JUnit Test Suite

A comprehensive JUnit Test Suite can be dowloaded as a gzipped tar file; *jgromacs_v1_test.tgz*. The suite contains 21 test classes with 342 test methods.

As a first step, extract the gzipped tar file:

```
tar xvzf jgromacs_v1_test.tgz
```

This will give you the directory *test/* which contains binaries (.class filess), source codes (.java files) and an example dataset (in the subdirectory *test\dataset*) required for running the test suite.

To run the tests with the junit.textui.TestRunner tool, use the following command in the directory *test/* :

```
java -cp XXX/jgromacs_v1_bin.jar:junit-4.10.jar:. junit.textui.TestRunner AllTests
```

Here XXX represents the path of location where *jgromacs_v1_bin.jar* has been downloaded to. (There should be a space after the period and before “junit.textui...”.)

Note that 2 of the 342 test methods depend on Gromacs installation as they have been designed to test those methods of the package that are dependent on Gromacs. This means that two tests will fail unless you set up your Gromacs environment. The remaining 340 test methods can be executed independently of Gromacs.

Subpackages

The JGromacs API comprises 5 subpackages, each of which is a collection of Java classes sharing a distinct function.

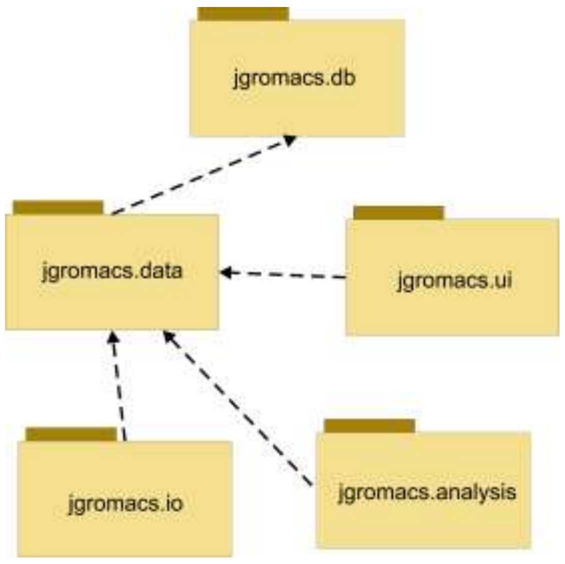
The largest subpackage, **jgromacs.data** contains 13 classes representing different levels of structural information from single atoms and residues to protein structures and molecular dynamics trajectories. Objects of these classes are the basic building blocks of JGromacs applications.

The subpackage **jgromacs.io** contains 2 classes that provide static methods for reading and writing Gromacs file formats (pdb, gro, ndx, xtc, trr) as well as sequence files (fasta) and mathematical structures (matrices and vectors).

The subpackage **jgromacs.db** contains 2 classes that represent atom and residue types.

The subpackage **jgromacs.analysis** is a collection of 6 classes that provide static methods for a series of basic data analysis tasks from calculating dihedral angles to extracting contact matrices to weighted superposition of structures. Routines of the analysis toolbox operate on the classes of jgromacs.data subpackage.

Finally, the subpackage **jgromacs.ui** containing 2 classes is designed to help the development of analysis applications of user-friendly UIs. The JGromacs user interface incorporates many aspects of the UI of Gromacs applications.



UML package diagram representing the inner structure of the JGromacs library. Dependencies between the five subpackages are shown by arrows (an arrow is pointing from package A to package B if package A uses the classes of package B)..

Multi-level representation

The subpackage *jgromacs.data* contains 13 classes that represent sequence, structure and trajectory data. The following section aims to give you a general overview on how these hierarchically related classes represent multiple levels of structural information.

Figure 1 presents the hierarchy of the different levels of coordinate (i.e. structural and trajectory) data and the way they are mapped on JGromacs classes. The parsers in the subpackage *jgromacs.io* provide a simple way to read single structures/models from coordinate (PDB and GRO) files. A single protein structure may be further decomposed to a set of polypeptide chains, each chain is a collection of residues and each residue is composed of a set of atoms. The position of each atom can be described by 3-dimensional atomic coordinates.

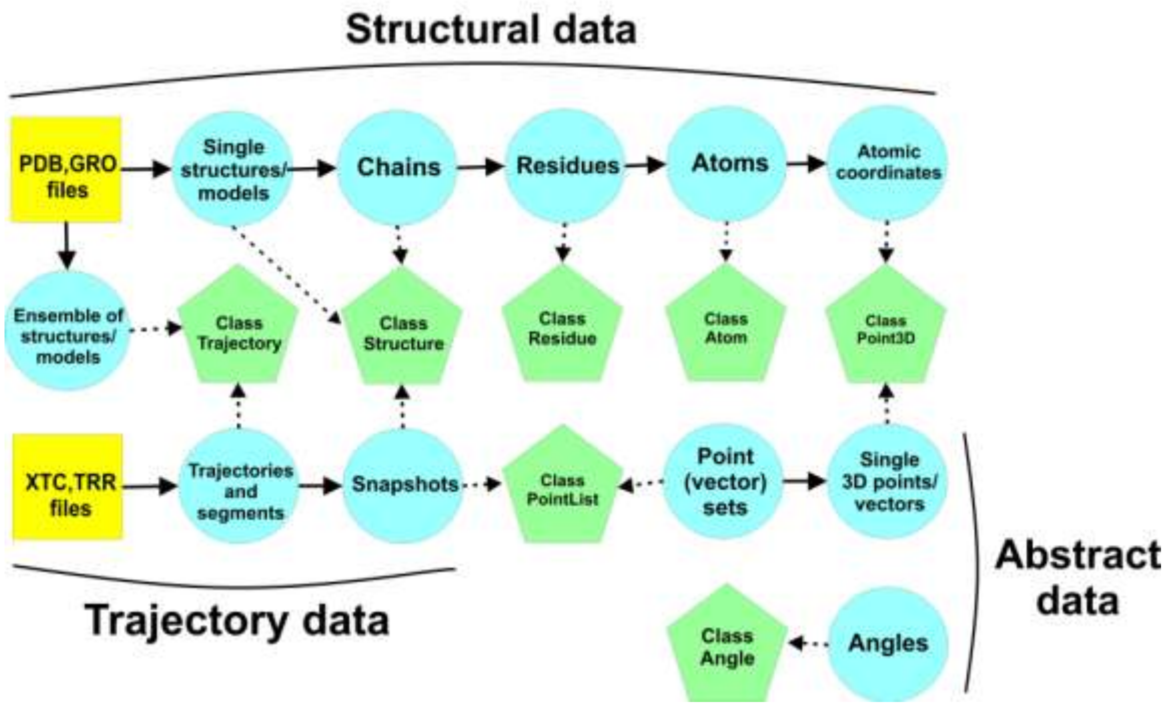


Figure 1: Multiple levels of structural and trajectory data. Blue circles represent different levels of data, green pentagons represent Java classes. An arrow between two blue circles means a hierarchical relationship, while an arrow between a blue circle and a green pentagon means a mapping between data and JGromacs objects.

As shown in **Figure 1**, the above-mentioned 5 levels of structural information are represented by 4 different JGromacs classes. Single structures/models and single polypeptide chains can be stored by objects of the class *Structure*.

Single residues are stored by objects of the class *Residue*. Single atoms are stored by objects of the class *Atom*. Finally, single 3-dimensional atomic coordinates are stored by objects of the class *Point3D*.

On the other hand, one can import ensembles of conformations by parsing XTC and TRR trajectory files or reading structural (e.g. NMR) ensembles from PDB files. MD trajectories and structural ensembles can be stored by objects of the class *Trajectory*. Individual frames of an MD trajectory or structural ensemble can be retrieved either as objects of the class *Structure* or objects of the class *PointList* which store the list of atomic coordinates only.

Mathematical objects such as 3-dimensional points, point sets and angles are stored in objects of the JGromacs classes *Point3D*, *PointList* and *Angle*, respectively.

Figure 2 shows how atomic index sets used to define subsets of atoms of interest are represented by JGromacs classes. Atomic index sets and lists of index sets can be read from NDX files using the parsers provided by the *jgromacs.io* subpackage. Atomic index sets are stored in objects of the class *IndexSet*, while lists of atomic index sets are stored in objects of the class *IndexSetList*. Similarly, one can use index sets to define a subset of trajectory frames. Index sets defining subsets of frames are stored by objects of the class *FrameIndexSet*.

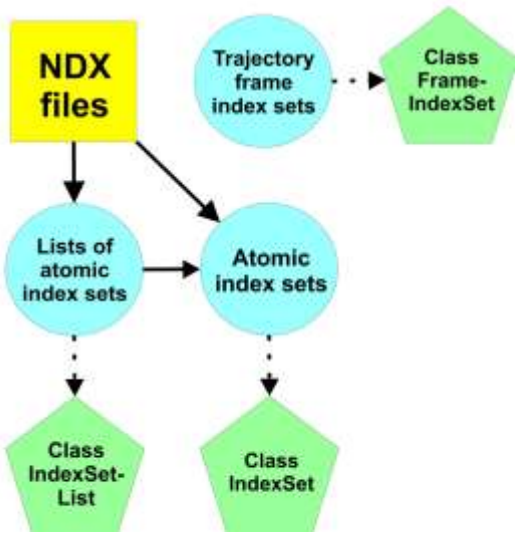


Figure 2: Classes for storing atomic and frame index sets. Blue circles represent different levels of data, green pentagons represent Java classes. An arrow between two blue circles means a hierarchical relationship, while an arrow between a blue circle and a green pentagon means a mapping between data and JGromacs objects.

The subpackage *jgromacs.data* also provides classes for handling protein sequences and alignments. **Figure 3** shows how sequence and alignment data read from FASTA files are stored in JGromacs objects. Pairwise and multiple sequence alignments are stored in objects of the class *Alignment*. Single sequences are stored in objects of the class *Sequence*. Single positions in an amino acid sequence are stored in objects of the class *SequencePosition*.

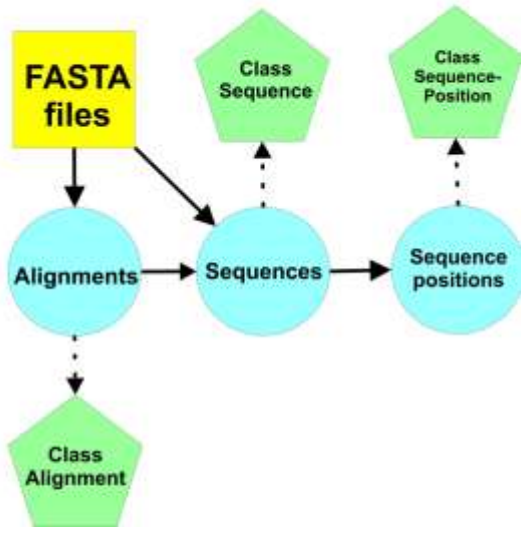


Figure 3: Classes for storing sequence and alignment data. Blue circles represent different levels of data, green pentagons represent Java classes. An arrow between two blue circles means a hierarchical relationship, while an arrow between a blue circle and a green pentagon means a mapping between data and JGromacs objects.

For further information about the classes in *jgromacs.data* and the other subpackages please read the other sections of this documentation.

Quick Start Guide

This section gives you a brief step-by-step tutorial to some of the basic features of JGromacs such as reading and analysing structures, trajectories and index sets, using protein sequence data, running Gromacs commands from within your Java code, creating simple JGromacs applications etc.

If you are interested in the more complete documentation of all the subpackages, classes and methods, please see the further sections of this document or the HTML-based JavaDoc documentation on our website. You can download an example dataset from our website in case you would like to try out the examples below. (A more complete library of example codes is also accessible on our webpage.)

How to read structures?

JGromacs represents structures, protein models and chains by objects of the class Structure. Reading structures from PDB and GRO files into Structure objects is simple. You only have to use the parsers in the class IOData. The following lines read the contents of coordinate files into JGromacs objects:

```
Structure s = IOData.readStructureFromPDB("example.pdb");

Structure s2 = IOData.readStructureFromGRO("example.gro");
```

Once the structures are read into Structure objects, you can access their properties by calling the appropriate methods of the objects. For example, the number of atoms, residues or chains in the structure can be obtained by:

```
int numOfAtoms = s.getNumberOfAtoms();

int numOfResidues = s.getNumberOfResidues();

int numOfChains = s.getNumberOfChains();
```

How to read alternative protein models?

When a single PDB file contains multiple structures, you can read a particular protein model identified by the model serial number. The following code reads Model 2 from the PDB file into a Structure object:

```
Structure s = IOData.readStructureFromPDB("example.pdb", 2);
```

How to read an ensemble of structures?

In case you need all models stored in a PDB file you can read the ensemble into an array of Structure objects:

```
Structure[ ] E = IOData.readStructuresFromPDB("example.pdb");
```

Alternatively, you can read an array of Structure objects from a directory. In that case all PDB files in the directory will be read into the array. The following parser reads all PDB files in the current directory:

```
Structure[ ] E=IOData.readStructuresFromPDBsInDirectory ("");
```

How to retrieve protein chains?

You may also need to retrieve different chains from a structure and handle them as separate Structure objects. The following code splits a single structure (*struct*) containing multiple chains into an array of Structure objects storing each chain separately.

```
Structure[ ] chains = struct.getChains();
```

How to retrieve residues?

In JGromacs, amino acid residues, water molecules and other molecules are represented by objects of the class Residue. A Structure object is a collection of Residue objects, which can be easily retrieved from the structure. For example the following code extracts two residues from the Structure object *struct*. The first residue is selected by its “residue index”, while the second is identified by its “list index” (its position in the list of residues.)

```
Residue res = struct.getResidueByIndex(24);

Residue res2 = struct.getResidue(13);
```

You can access the properties of residues by calling the appropriate methods of the Residue objects. For example, the following code obtains the name, chain ID and 3-letter code of the residue.

```
String name = res.getName();

String chainID = res.getChainID();

String code = res.get3LetterCode();
```

You can also overwrite some of the properties of the residue. For example, the residue index can be reset by:

```
res.setIndex(78);
```

How to retrieve atoms?

In JGromacs, atoms are represented by objects of the class Atom. A Residue object is a collection of Atom objects, which can be easily retrieved from the residue. For example, the following code extracts two atoms from the Residue object *res*. The first atom is selected by its “atom index”, while the second is identified by its “list index” (its position in the list of atoms.)

```
Atom atom = res.getAtomByIndex(91);

Atom atom2 = res.getAtom(100);
```

On the other hand, atoms can be directly accessed from Structure objects. For example, the following code retrieves an atom of given index from a Structure object *struct*:

```
Atom atom = struct.getAtomByIndex(77);
```

You can access and reset the properties of atoms by calling the appropriate methods of the Atom objects. For example, the following code obtains the occupancy and resets the B-value of the atom:

```
double occupancy = atom.getOccupancy();

atom.setBvalue(24.19);
```

How to extract special atoms from residues?

The methods of Residue class make it simple to retrieve atoms of interest. Suppose *res* represents an amino acid residue. You can get for example the alpha carbon atom, N-terminal nitrogen atom or carbonyl oxygen atom by:

```
Atom alphaCarbon = res.getAlphaCarbon();

Atom NTerminalNitrogen = res.getNTerminalNitrogen();

Atom carbonylOxygen = res.getCarbonylOxygen();
```

How to retrieve atomic coordinates?

Atomic coordinates are represented as 3-dimensional points by objects of the class Point3D. Each Atom object contains a Point3D object to store its (x,y,z) coordinates. You can access the coordinates of a single atom or even retrieve the coordinates of all atoms in a structure as a PointList object. The class PointList represents a list of 3-dimensional points which can be transformed by mathematical operations. The following code extracts the coordinates of a single atom (*atom*) as a Point3D object. It then retrieves the list of coordinates of all atoms in a structure (*struct*) as a PointList object:

```
Point3D coordinates = atom.getCoordinates();

PointList allCoordinates = struct.getAllAtomCoordinates();
```

You can also move atoms to different positions by resetting the coordinates of a single atom or all atoms in a structure:

```
atom.setCoordinates(new Point3D(1.1,-4.3,3.9));

struct.setAllAtomCoordinates(allCoordinates);
```

How to rotate a set of points

Simple geometric transformations such as rotation and translation can be applied on a set of points. For example, the following code rotates the points in a PointList object (*points*) using a given rotation matrix:

```
points.rotate(rotationMatrix);
```

How to calculate inter-atomic distances?

The distance between two atoms or two points are easy to calculate. The following code gives the distance between two points (*p1* and *p2*) and two atoms (*atom1* and *atom2*):

```
double distance = p1.distance(p2);

double distance2 = atom1.distance(atom2);
```

In addition you can calculate the distance between two residues (*res1* and *res2*) that may mean the distance between their alpha carbon atoms, their closest atoms or their closest heavy (non-hydrogen) atoms:

```
double distance = res1.distanceAlphaCarbons(res2);

double distance2 = res1.distanceClosest(res2);

double distance3 = res1.distanceClosestHeavy(res2);
```

How to calculate dihedral angles?

In JGromacs, angles are represented by objects of the class Angle. You can easily calculate a number of relevant angles in a structure. For example, the following code calculates the dihedral angle Psi of the residue of index 18 in the Structure object *struct*. Secondly, it calculates the side chain dihedral angle Chi2 of the same residue:

```
double psi = Angles.getDihedralPsi(struct, 18);

double chi2 = Angles.getDihedralChi2(struct, 18);
```

How to deal with atom and residue types?

In JGromacs, atom and residue types are represented by objects of the classes AtomType and ResidueType. Each Atom objects contains an AtomType object defining the type of the atom. Similarly, each Residue object contains a ResidueType object defining the type of the residue. You can access or reset atom and residue types:

```
AtomType atype = atom.getAtomType();

ResidueType rtype = res.getResidueType();

atom.setAtomType(new AtomType(“Na”));

res.setResidueType(new ResidueType(“His”));
```

How to read atomic index sets?

Index sets are collections of atomic indices that are useful for defining subsets of atoms in a structure. Gromacs stores index sets in NDX files that you can read using the parsers in the class IOData. In JGromacs, index sets are represented by objects of the class IndexSet. For example, the following code reads the first index set from an NDX file into an IndexSet object:

```
IndexSet iSet = IOData.readIndexSetFromNDX("example.ndx");
```

Alternatively, if your NDX file contains multiple index sets, you can read the the whole list of index sets into an IndexSetList object:

```
IndexSetList iList =
                IOData.readIndexSetListFromNDX("example.ndx");
```

An IndexSetList object is a simple list of index sets which you can access by referring to their names or list indices. You can also add new index sets to the list etc. For example, the following code gets two index sets from the IndexSetList object *iList* by referring to their list index and name. Then it adds a new empty index set of the name “Anything” to *iList*.

```
IndexSet iSet = iList.getIndexSet(1);

IndexSet iSet2 = iList.getIndexSet(“SideChain”);

iList.addIndexSet(new IndexSet(“Anything”));
```

How to define more complex index sets?

You can use the basic set operations (intersection, union, subtraction) to define index sets of complex criteria. For example, the following code creates an index set which is the union of two index sets (*set1* and *set2*) from which a third index set (*set3*) is subtracted:

```
IndexSet result = set1.union(set2);

result = result.subtract(set3);
```

How to extract substructures using index sets?

Index sets can be used to extract a subset of atoms of interest from a Structure object. The substructure defined by the index set is retrieved as another Structure object. The following code extracts the substructure defined by the index set *iSet* from the Structure object *struct*:

```
Structure subStructure = struct.getSubStructure(iSet);
```

How to extract default index sets from structures?

Given a Structure object, you can generate 12 default index sets: System (all the atoms in the system), Protein (all protein atoms), Protein-H (all protein atoms except hydrogens), C-alpha (alpha carbon atoms), Backbone (all backbone atoms), MainChain (backbone atoms plus carbonyl oxygens), MainChain+Cb (main chain atoms plus beta carbons), MainChain+H (main chain atoms plus hydrogens), SideChain (all non-main chain atoms), SideChain-H (all side chain atoms except hydrogens), Non-Protein (all non-protein atoms) and Water (all water atoms). Gromacs has the same definition for these 12 default index sets. For example, the following code will give you the index sets corresponding to alpha carbon and side chain atoms. The index set defining alpha carbon atoms is then used to extract a Structure object containing these atoms only:

```
IndexSet alphaCarbons = struct.getAlphaCarbonIndexSet();

IndexSet sideChains = struct.getSideChainIndexSet();

Structure alpha = struct.getSubStructure(alphaCarbons);
```

How to use protein sequence data?

In JGromacs, amino acid sequences are represented by objects of the class Sequence. You can read protein sequences from FASTA files using the parser in the input/output class IOData. Another option is retrieving the amino acid sequence from a Structure object representing a protein. Pairwise or multiple sequence alignments can also be read from FASTA files into objects of the JGromacs class Alignment. For example, the following code reads a sequence from a FASTA file into a Sequence object. It also extracts the protein sequence from a Structure object (*struct*). Finally, it reads an alignment from a FASTA file into an Alignment object:

```
Sequence seq =
        IOData.readSequenceFromFASTA(“example.fasta”);

Sequence seq2 = struct.getSequence();

Alignment align =
        IOData.readAlignmentFromFASTA(“example2.fasta”);
```

JGromacs makes it simple to integrate sequence and structure/trajectory data. Suppose you would like to compare the dynamics of conserved residues across a set of homologous proteins. Starting from a multiple sequence alignment, you may need to retrieve the index sets of those residues in each protein that are not aligned to gaps in the alignment. These index sets can be used for further analysis; e.g. comparing the RMSF fluctuations of conserved residue positions. For example, starting from an Alignment object (*align*), the following code extracts the index sets of match positions (i.e. columns where there is no gap in the alignment) of the first and second sequences:

```
IndexSet iSet1 = align.getMatchPositionIndices(0);

IndexSet iSet2 = align.getMatchPositionIndices(1);
```

How to read trajectories?

In JGromacs, trajectory data are stored in objects of the class Trajectory. The input/output class IOData provides parsers that make it simple to read data from XTC and TRR files into Trajectory objects. A Structure object is also necessary for initializing the trajectory, since XTC and TRR files contain only coordinate data. The following code first reads a GRO file into a Structure object. Secondly, it reads an XTC and a TRR file into Trajectory objects.

```
Structure s = IOData.readStructureFromGRO(“example.gro”);

Trajectory sim = IOData.readTrajectory(s, “example.xtc”);

Trajectory sim2 = IOData.readTrajectory(s, “example.trr”);
```

Note, the code above works only if Gromacs is installed on your machine, as it makes use of the “gmxdump” Gromacs command to process trajectory files. Alternatively, if your XTC or TRR file has already been preprocessed by the “gmxdump –f filename > example.dat” command, the dumped file can be read by the following parser which works independently of the Gromacs package:

```
Trajectory sim = IOData.readDumpedTrajectory(s,“example.dat”);
```

How to obtain simulation frames?

Frames of a trajectory can be extracted either as a Structure object or as a PointList object. The following code retrieves two frames from the Trajectory object *sim* by referring to the frame indices:

```
Structure frame = sim.getFrameAsStructure(111);

PointList frame2 = sim.getFrameAsPointList(222);
```

It is also simple to add or remove frames to or from a Trajectory object. For example, the following code adds a new frame defined by a PointList object (*newframe*) and removes the frame of the index 1234:

```
sim.addFrame(newframe);

sim.removeFrame(1234);
```

How to extract segments of trajectories?

You can retrieve a certain segment of a trajectory by selecting its first and last frames and sampling frequency. The segment trajectory is returned as a Trajectory object. For example, the following code extracts 100 frames from the Trajectory object *sim*. The first and last frames of the resulting segment are the 1000th and the 4000th frames of the original trajectory and every 30th frame is retrieved:

```
Trajectory segment = sim.getSubTrajectory(1000,4000,30);
```

How to extract subtrajectories using index sets?

Index sets can be used to extract trajectories that contain only subsets of atoms. The subtrajectory defined by the index set is retrieved as another Trajectory object. The following code extracts the subtrajectory defined by the index set *iSet* from the Trajectory object *sim*:

```
Trajectory subTrajectory = sim.getSubTrajectory(iSet);
```

How to get String representations of objects?

Every JGromacs object has a String representation, because all JGromacs data classes implement the toString() method. For example, the following code prints out the String representations of a Structure object (*struct*), a Residue object (*res*), an Atom object (*atom*) and a Trajectory object (*sim*):

```
System.out.println(struct);

System.out.println(res);

System.out.println(atom);

System.out.println(sim);
```

Some JGromacs data classes also provide a method called toStringInfo() which give you summary information about the given object. For example, the following code prints out the summary information about a Sequence object (*seq*) and a Structure object (*struct*):

```
System.out.println(seq.toStringInfo());

System.out.println(struct.toStringInfo());
```

How to calculate the contact matrix

The subpackage *jgromacs.analysis* provides you with a basic analysis toolbox covering a range of topics implemented in 6 classes: Distances, Dynamics, Angles, GNM, Similarity and Superposition. Analysis tools are static methods, so you can access them without creating instances of these classes. For example, the following code uses the method *getContactMatrix*() of the class Distances to calculate the contact matrix of a protein structure. The input structure is represented by a Structure object (*struct*). In this example, the distance between two residues is defined as the distance between their alpha carbon atoms. The distance cutoff defining contacts is set to 7 Angstroms:

```
Matrix contacts = Distances.getContactMatrix(struct,
                                  Distances.ALPHACARBON, 0.7);
```

How to superpose two structures

Another example for the use of the *jgromacs.analysis* toolbox is the superposition of two protein structures. The following code performs unweighted superposition of a Structure object (*struct*) and a reference Structure object (*reference*), minimising the RMSD between the two structures. The resulting Structure object is the superposed version of the input structure:

```
Structure result = Superposition.superposeTo(struct, reference);
```

How to extract distance time series

A third example for the use of the *jgromacs.analysis* toolbox is the way you can get the time series of the distance between two atoms in the course of a trajectory. For example, the following code retrieves the time series of the distance between two atoms selected by their indices (103 and 204):

```
ArrayList<Double> timeseries =
                Distances.getDistanceTimeSeries(sim, 103, 204);
```

How to do PCA

A fourth example for the use of the *jgromacs.analysis* toolbox is Principal Component Analysis (PCA) of trajectory data. The following code performs PCA of a trajectory represented by a Trajectory object (*sim*). The Matrix array returned contains two elements: the diagonal eigenvalue matrix and the matrix containing the corresponding principal component vectors:

```
Matrix[ ] result = Dynamics.getPCA(sim);

Matrix D = result[0];   // Eigenvalues

Matrix V = result[1];   // PC vectors
```

How to create JGromacs applications

Adding a simple but smart command line user interface to your Java application is easy. You just have to create a class that extends the class Application of the subpackage jgromacs.ui. Doing so, your application will have a UI that incorporates many aspects of the Gromacs command line UI. For example, the following code creates a class extending the parent class Application:

```
public class MyApplication extends Application {
```

Within the constructor of your class, you have to set the name of the XML configuration file you use to configure the UI of your application. In addition, you may set if you would like your application to write a log file. For example, the following constructor defines the name of the XML configuration file (myfile.xml), the name of the log file (mylogfile.log) and makes the application to write a log file:

```
 public MyApplication() {
      super();
      XMLFileName = “myfile.xml”;
      setLogFileName(“mylogfile.log”);
      setWritingToLogFile(true);
 }
```

Do not use the main() method of your class as the entry point of your application. Instead, let the method main() just contain the following code:

```
public static void main(String[] args) {
     MyApplication app = new MyApplication();
     app.run(args);
}
```

Override the method runCore() of the parent class Application. The method runCore() will be the entry point of your application, you can insert your own code here:

```
public void runCore() {

  // Your code: the program itself

}
```

Within runCore() you can access the arguments given by the user in the command line. The list of arguments handled by your application can be specified in the XML configuration file. (See the description of the syntax of XML configuration files on page 36.) Command line arguments are identified by unique flags (String IDs). For example, the following code checks if the argument defined by flag “-n” is given by the user and returns the parameter value assigned to the argument defined by flag “-s”:

```
boolean given = isArgumentGiven(“n”);

String value = getArgumentValue(“s”);
```

In case the user runs your application with the only argument “-h”, as in Gromacs, a help page will show up summarizing the meta data and options of the application.
Within runCore() you can also write text to the log file at any point:

```
writeToLogFile(“This text is going into the log file”);
```

For more information on creating applications with JGromacs UI, see the separate section on page 35. You may also want to look at the class TemplateApplication in jgormacs.ui and the example code UIDemo that demonstrate the correct structure of JGromacs applications.

How to write data back to Gromacs files?

Would you like to save your data to the disk in Gromacs file formats? Methods of the input/output class IOData let you write JGromacs objects back to Gromacs files. This makes it possible to integrate Java and Gromacs tools into a single data analysis pipeline. For example, the following code writes a Structure object (*struct*) to GRO and PDB files. A Sequence object (*seq*) is written to a FASTA file. An Alignment object (*align*) is saved to a FASTA file. An IndexSet object (*iSet*) is written to an NDX file. Finally, an IndexSetList object (*iList*) is saved to an NDX file:

```
IOData.writeStructureToGRO(“output.gro”,struct);

IOData.writeStructureToPDB(“output.pdb”,struct);

IOData.writeSequeceToFASTA(“output.fasta”,seq);

IOData.writeAlignmentToFASTA(“output2.fasta”,align);

IOData.writeIndexSetToNDX(“output.ndx”,iSet);

IOData.writeIndexSetListToNDX(“output2.ndx”,iList);
```

How to run Gromacs commands from Java code?

Another option to integrate Java and Gromacs tools is to run Gromacs commands from within your Java code and read the resulting output files back to JGromacs objects. The method *runGromacsCommand()* of the class IOData makes it really simple to do so. You can execute any Gromacs (or other) commands as in the Unix shell (sh). This means for example that you can use shell pipes (>,>>,<,|) in the command String. Once the command is executed, the output files listed in a String array are automatically read into JGromacs objects.

For example, the following code executes the Gromacs command *make_ndx* from the Java code. The Gromacs tool will create the default index sets from an input coordinate file (*example.gro*). Note, that the shell pipe **|** is used to write to the standard input of make_ndx. The resulting index sets are written by make_ndx to the file *output.ndx* which is automatically read into an IndexSetList object:

```
String[ ] files = {“output.ndx”};
Object[ ] result = IOData. runGromacsCommand(“echo q |
                    make_ndx –f example.gro -o output.ndx”, files);
IndexSetList iList = result[1];
```

Example

A simple JGromacs example code is presented in this section. The example illustrates how to retrieve and compare conformational ensembles using JGromacs. The code first reads trajectory data from a TRR file then calculates the medoid frame of the trajectory based on the standard RMSD similarity measure. After that it extracts the subset of frames that are more similar to the medoid frame than a RMSD cutoff of 0.1 nm. Finally, the resulting ensemble of conformations and the total conformational ensemble of the trajectory are compared by calculating the ensemble averaged RMSD (Brüschweiler, 2002) between the two sets.

Note that the medoid frame is calculated as a PointList object using the getMedoidRMSD() method of class Similarity in the jgromacs.analysis toolbox. The subset of simulation frames similar to the medoid is defined by a FrameIndexSet object calculated with the findSimilarFramesRMSD() method of class Similarity. The ensemble averaged RMSD (eRMSD) is computed by the method getEnsembleAveragedRMSD() of class Dynamics in the jgromacs.analysis package. Since this method takes two Trajectory objects as input, the selected frames are first extracted as a subtrajectory.

```
// Reading trajectory data from TRR file
Structure s = IOData.readStructureFromGRO(“calpha.gro”);
Trajectory t = IOData.readTrajectory(s,”calpha.trr”);

// Calculating medoid frame of the trajectory
PointList medoid = Similarity.getMedoidRMSD(t);

// Retrieving frames that are similar to the medoid
FrameIndexSet frames =
            Similarity.findSimilarFramesRMSD(t, medoid, 0.1);

// Extracting similar frames as a subtrajectory
Trajectory t2 = t.getSubTrajectory(frames);

// Calculating ensemble averaged RMSD
double eRMSD = Dynamics.getEnsembleAveragedRMSD(t,t2);

// Printing out the result
System.out.println(“eRMSD: ”+eRMSD);
```

For a further list of downloadable example codes see the next section.

Example codes

The following library of example codes can be downloaded from our website.

Package: input

ReadDataMatrix.java – How to read Matrix and ArrayList objects from file
ReadIndexSets.java – How to read index sets and index set lists from NDX file
ReadSequences.java – How to read sequences and alignments from FASTA file
ReadStructures.java – How to read structures from PDB or GRO file
ReadTrajectory.java – How to read trajectories from XTC or dXTC (dumped XTC) file

Package: output

WriteDataMatrices.java – How to write Matrix and ArrayList objects to file
WriteIndexSets.java – How to write index sets or index set lists to NDX file
WriteSequences.java – How to write sequences or alignments to FASTA file
WriteStructures.java – How to write structures to PDB or GRO file

Package: data

IndexSets.java – How to use index sets and index set lists
PointLists.java – How to use point lists and points
Sequences.java – How to use sequences, sequence positions and alignments
Structures.java – How to use structures, residues and atoms
Trajectories.java – How to use trajectories, frames and frame lists

Package: analysis

ContactMatrices.java – How to calculate contact matrices
Dihedrals.java – How to calculate dihedrals and other angles of interest
DistanceMatrices.java – How to calculate distance matrices
DistanceMisc.java – How to do miscellaneous things about distances
DistanceTimeSeries.java – How to extract distance time series data
DynamicsMisc.java – How to do miscellaneous things about dynamics
FindFrames.java – How to extract simulation frames of interest
Fluctuations.java – How to calculate residue fluctuations
GaussianNetworkModel.java – How to use Gaussian Network Models
PCA.java – How to perform Principal Component Analysis (PCA)
Similarities.java – How to calculate similarity of conformations
Superposing.java – How to superpose structures

Package: ui

UIDemo.java – How to build an application with JGromacs user interface

Package: advanced

WaterInBindingPocket.java – How to find water molecules in the binding pocket
MDvsGNM.java – How to compare fluctuation profiles from MD and GNM
ConservedResidues.java – How to compare the dynamics of conserved residues
CompareEnsembles.java – How to select and compare conformational ensembles
WeightedSuperposition.java – How to calculate weighted superposition of two structures
CompareContacts.java – How to compare residue contacts in different conformations
CheckSampling.java – How to check the sampling in a simulation
DistanceDistributions.java – How to normalize atomic distances

Classes

JGromacs is a lightweight library comprising a total of 25 Java classes:

Subpackage jgromacs.data
Class Atom
Class Residue
Class Structure
Class Trajectory
Class IndexSet
Class IndexSetList
Class Sequence
Class SequencePosition
Class Alignment
Class Point3D
Class PointList
Class Angle
Class FrameIndexSet

Subpackage jgromacs.io
Class IOData
Class IOMath

Subpackage jgromacs.db
Class AtomType
Class ResidueType

Subpackage jgromacs.analysis
Class Distances
Class Dynamics
Class Superposition
Class Angles
Class Similarity
Class GNM

Subpackage jgromacs.ui
Class Application
Class TemplateApplication

Descriptions of classes

Below is a brief description of each class found in the JGromacs v1.0. API. For a more detailed description of all methods of all classes, please read the JGromacs JavaDoc documentation downloadable from the project's website.

Subpackage jgromacs.data

This subpackage contains classes that represent different levels of structural data. JGromacs objects have a hierarchical relationships, e.g. an object representing a residue is a collection of objects representing atoms while an object representing a structure is a collection of objects representing residues. This object-oriented representation makes it easier to access and manipulate structural and trajectory data. Data can be imported from Gromacs files (of PDB, GRO, NDX, XTC and TRR formats) and FASTA files into six classes called Structure, Trajectory, IndexSet, IndexSetList, Sequence and Alignment. Parsers for reading Gromacs files are provided by the class IOData in subpackage jgromacs.io. This class also contains static methods for writing data back to Gromacs files.
Note that all classes in subpackage jgromacs.data implement the methods equals(), clone(), and toString(). Classes Residue, Structure, IndexSet, IndexSetList, Sequence, Alignment and PointList also provide a method called toStringInfo() that returns summary information about the object.

Class Atom

Objects of this class represent a single atom. An Atom object has a name, index, atom type and (x,y,z) coordinates. Atomic coordinates are stored in Point3D objects. The type of the atom is stored in an object of class AtomType found in the subpackage jgromacs.db.
Methods: The class Atom provides getter and setter methods to return and set the properties of the atom. The class also contains a method for calculating the Euclidean distance between two atoms. Methods for recognizing particular atom types (e.g. gamma carbon, N-terminal nitrogen, carbonyl oxygen) are also included.

Class Residue

Objects of this class represent a single residue. A Residue object has a name, index, residue type and chain ID. In addition, a Residue object is a collection of Atom objects. The type of the residue is stored in an object of class ResidueType found in the subpackage jgromacs.db.
Methods: The class Residue provides getter and setter methods to return and set the properties of the residue. Methods for modifying the residue (i.e. modifying its atoms or atomic coordinates) are included. There are methods for extracting particular atoms (e.g. alpha carbon, C-terminal carbon, delta carbon) and particular index groups (e.g. main chain atoms, side chain atoms). The class contains methods for calculating the distance between two

residues based on different definitions (e.g. distance between alpha carbon atoms, distance between closest atoms etc.)

Class Structure

Objects of this class represent a single protein structure. A Structure object has a name and is a collection of objects of the class Residue.

Methods: The class Structure provides getter and setter methods to return and set the properties of the structure. A series of methods for modifying the structure (i.e. modifying its residues, atoms or atomic coordinates) are also provided. There are methods for extracting special index groups (e.g. heavy protein atoms, backbone atoms, heavy side chain atoms). Additional methods such as retrieving the protein sequence from the structure or returning sub-structures are also included.

Class Trajectory

Objects of this class represent a molecular dynamics trajectory. A Trajectory object has a name, start time and time step. In addition, a Trajectory object is a list of simulation frames which can be extracted as Structure or PointList objects.

Methods: The class Trajectory provides getter and setter methods to return and set the properties of the trajectory. Methods for modifying the trajectory (i.e. adding and removing frames) are also included. The class contains additional methods for retrieving a certain subset of frames from the simulation.

Class IndexSet

Objects of this class represent a single atomic index set. An IndexSet object has a name and is a collection of atomic indices.

Methods: The class IndexSet provides getter and setter methods to return and set the properties of the index set. Methods for modifying the index set (i.e. adding and removing indices) are included. The class contains methods that implement basic set operations (intersection, subtraction and union) between index sets enabling to define atomic index sets of complex criteria.

Class IndexSetList

Objects of this class represent a list of IndexSet objects.

Methods: The class IndexSetList provides getter methods to return the properties of the list. Methods for modifying the index set list (i.e. adding and removing index sets) are also included. The class contains an additional method for returning the union of all index sets as a single index set.

Class Sequence

Objects of this class represent a single protein sequence. A Sequence object has a name and is a collection of SequencePosition objects.

Methods: The class Sequence provides getter and setter methods to return and set the properties of the amino acid sequence. Methods for modifying the sequence (i.e. adding, inserting and removing sequence positions and gaps) are included. The class contains additional methods such as extraction of sub-sequences, reversing and concatating sequences etc.

Class SequencePosition

Objects of this class represent a single sequence position which is the building block of Sequence objects. A SequencePosition object has an index, residue type and annotation. The type of the residue is stored in an object of class ResidueType found in the subpackage jgromacs.db.
Methods: The class SequencePosition provides getter and setter methods to return and set the properties of the sequence position.

Class Alignment

Objects of this class represent a multiple sequence alignment. An Alignment object has a name and is a collection of Sequence objects.
Methods: The class Alignment provides getter and setter methods to return and set the properties of the alignment. Methods for modifying the alignment (i.e. adding and removing sequences or alignment columns) are also included. The class provides additional methods such as extracting the consensus sequence and calculating the collapsed alignment (i.e. columns of the alignment that do not contain gaps) etc.

Class Point3D

Objects of this class represent a single 3-dimensional point or the vector pointing to this point from the origin.
Methods: The class Point3D provides getter and setter methods to return and set the coordinates of the point. The class also contains a method for calculating the Euclidean distance between two points. Methods implementing basic vector operations such as addition and subtraction of two vectors, multiplication by scalar, inner product and cross product are also included.

Class PointList

Objects of this class represent a list of Point3D objects.
Methods: The class PointList contains methods for modifying the point list (i.e. adding and removing points). Additional methods for rotating and translating the point set and calculating its centroid are also included.

Class Angle

Objects of this class represent a single angle. The value of the angle can be converted between degrees and radians.

Class FrameIndexSet

Objects of this class represent a frame index set (defining a subset of trajectory frames). A FrameIndexSet object has a name and is a collection of frame indices.
Methods: The class FrameIndexSet provides getter and setter methods to return and set the properties of the frame index set. Methods for modifying the frame index set (i.e. adding and removing indices) are included. The class contains methods implementing basic set operations (intersection, subtraction

and union) between frame index sets that make it possible to define frame index sets of complex criteria.

Subpackage jgromacs.io

This subpackage contains 2 classes providing static input/output methods for classes in the subpackage jgromacs.data. JGromacs can read and write PDB, GRO, XTC, TRR, dumped XTC and TRR, NDX and FASTA files. Matrices and vectors can also be read from and saved to text files.

Class IOData

IOData provides input and output methods for Structure, Trajectory, IndexSet, IndexSetList, Sequence and Alignment objects. Note that JGromacs understands the format of dumped XTC and TRR files: i.e. the standard output of the “gmxdump –f file.xtc” command of Gromacs. It can read either a previously dumped trajectory file or a raw XTC/TRR file by calling gmxdump as an external command. The second option requires Gromacs (gmxdump) to be installed. IOData provides a method for running external Gromacs commands from within the Java code and automatically reading the output files in as JGromacs objects.

Class IOMath

IOMath provides input and output methods for ArrayList and jama.Matrix objects.

Subpackage jgromacs.db

This subpackage contains 2 classes defining atom and residue types. Atom and Residue objects have types defined by these classes.

Class AtomType

Objects of this class represent a certain atom type which can be any element of the periodic table. Each Atom object contains an AtomType object specifying the type of the atom.
Methods: The class AtomType provides methods for returning the name and code of the atom type.

Class ResidueType

Objects of this class represent a certain amino acid type which can be any of the 20 standard amino acids. It can also represent a water molecule or an unknown residue. Each Residue object contains a ResidueType object specifying the type of the residue.
Methods: The class ResidueType provides methods for returning the name and 1-letter/3-letter code of the residue type.

Subpackage jgromacs.analysis

The subpackage contains 6 classes that provide static methods for various analysis tasks. The input arguments of these methods are objects of the classes found in subpackage jgromacs.data. This basic analysis toolbox is split into 6 different topics covered by the 6 classes below.

Class Distances

This class provides static methods for analysing inter-atomic distances. It provides methods for calculating the distance matrix of a set of points, atoms or residues (where the distance between residues is defined either as the distance of their alpha carbon atoms, their closest atoms or their closest heavy atoms). The class contains methods for calculating the mean distance matrix of a set of atoms or residues based on an ensemble of conformations.
It also provides methods for extracting the contact matrix of a structure and the contact matrix of the mean structure of an ensemble. It can also calculate the frequency-based contact matrix in which two residues are said to be in contact if they are in contact in at least a given percentage of simulation frames.
Methods for extracting the distance time series between two atoms in the course of a trajectory and calculating its summary statistics (mean, variance, minimum, maximum and range) are included.
The class contains methods for retrieving those atoms in a structure that are closer to a reference atom (or a reference set of atoms) than a given radius. Another method is provided for finding the atom of a given atom set that is located closest to a reference set of atoms. In addition, the class provides methods for extracting that simulation frame in which two atoms are closest or most distant from each other.
Other methods such as calculating the distance between a single atom and a set of atoms or between two sets of atoms (defined as the minimum of all pairwise distances) are also included.
Finally, the class provides methods for retrieving the index set of frames of a trajectory in which two atoms are closer or more distant from each other than a given distance cutoff.

Class Dynamics

This class contains static methods for analysing protein dynamics. It provides methods for calculating the 3Nx3N coordinate covariance and correlation matrix and the NxN atomic covariance and correlation matrix based on a trajectory.
Methods for principal component analysis (PCA) are also included such as computing the principal component vectors and the corresponding eigenvalues or calculating the cumulative variance profile.
To compare the essential modes of motion, the class provides methods for calculating the root mean squared inner product (RMSIP) between two sets of principal components and the covariance matrix overlap between two covariance matrices.

In order to characterise the fluctuation of residues, the root mean squared fluctuation (RMSF) profile can also be computed. The class provides methods for computing the F fluctuation matrix, where F_{ij} is the variance of D_{ij} inter-residue distance in the course of the simulation. Methods are included to calculate the relative fluctuation between two sets of residues (defined as the mean of the specified fluctuation submatrix entries).
In addition, the class implements more complex concepts from the literature such as the dynamical network of a protein (Sethi et al. 2009), the structural radius of a conformational ensemble (Kuzmanic and Zagrovic 2010), the ensemble averaged RMSD between two conformational ensembles (Brüschweiler 2002) and the contact probability map of a trajectory (Wei et al, 2009).

Class Superposition

This class contains static methods for performing unweighted and weighted superposition of protein structures. It uses the Kabsch algorithm to find the best rotation matrix that minimizes RMSD (Root Mean Squared Deviation) between two sets of points.

Class Angles

This class provides static methods for calculating the angles between vectors and planes. It also contains methods for calculating angles of interest including backbone dihedrals (Phi, Psi) and side chain dihedrals (Chi1, Chi2, Chi3, Chi4 and Chi5) in protein structures and their time series based on a trajectory. A method for calculating the Ramachandran plot is also included in the class.

Class Similarity

This class contains static methods for measuring the structural similarity between two protein conformations. Different measures of structural similarity (e.g. RMSD: coordinate RMSD, dRMSD: distance RMSD, wdRMSD: weighted distance RMSD) can be calculated.
The class also provides methods for measuring the deviation of a single atom between two structures (using different definition: RMSDi, dRMSDi, wdRMSDi).
Methods for calculating the similarity matrix of a set of conformations using different definitions of structural similarity (e.g. RMSD, dRMSD, wdRMSD) are also included. The class provides methods for retrieving the time series of similarity between consecutive simulation frames compared to a common reference structure.
Methods for calculating the medoid structure of a trajectory using RMSD or dRMSD similarity measures are also available.
Finally, the class contains methods for computing the difference distance matrix between two point sets or two structures. In addition, a method is provided for extracting the index set of frames of a trajectory that are more similar to a reference frame than a similarity (RMSD or dRMSD) cutoff.

Definitions used in class Similarity:

– RMSD(A,B):
  RMSD similarity between two structures A and B after superposition
– dRMSD(A,B) = sqrt(1/(N^2) sum_{ij}^N((d_{ij}^A- d_{ij}^B)^2))
  where d_{ij}^A is the distance of atom i and j in structure A.
– wdRMSD(A,B) = sqrt(1/W sum_{ij}^N(w_{ij}(d_{ij}^A- d_{ij}^B)^2))
  where w is the weight matrix and W= sum_{ij}^N w_{ij}
– RMSDi(A,B):
  the deviation of atom i between its positions in structure A and B
  after superposition of the two structures
– dRMSDi(A,B) = sqrt(1/N sum_j^N((d_{ij}^A- d_{ij}^B)^2))
– wdRMSDi(A,B) = sqrt(1/W sum_j^N(w_{ij} (d_{ij}^A- d_{ij}^B)^2))
  where W= sum_j^N w_{ij}

Class GNM

This class represents a Gaussian Network Model created for a protein structure. It provides methods for calculating the contact matrix, the Kirchhoff matrix, the orthogonal eigenvector matrix and the diagonal eigenvalue matrix. A method for computing the mean square fluctuation (MSF) profile is also included.

Subpackage jgromacs.ui

This subpackage contains 2 classes that facilitate the development of JGromacs applications that have user-friendly interfaces. Some features of Gromacs UI such as command line argument handling, log files, help messages and index group queries are incorporated in JGromacs UI as well. Developing a JGromacs application involves writing the core code and configuring the application interface through an XML file.

Class Application

This is the parent class of JGromacs applications. Applications extending this class have a JGromacs user interface that can be easily configured with the help of an XML configuration file. The class does not have public methods. The protected method runCore() must be implemented in the child class: it is the entry point of the application. Protected methods isArgumentGiven() and getArgumentValue() can be used in the child class to access command line arguments provided by the user. For more information on writing JGromacs applications please see page 35.

Class TemplateApplication

This is a template class for writing JGromacs applications. TemplateApplication extends class Application. As it is described by the comments in this class, the programmer needs to insert his own code in the method runCore() which overrides the method runCore() of class Application.

JGromacs user interface

The subpackage *jgromacs.ui* provides a tool to create Java applications with a command line user interface. The JGromacs UI incorporates many aspects of the Gromacs UI, supporting help messages, log files, command line argument parsing and can easily be set up with an XML configuration file.

To create a Java application that communicates with the user via a JGromacs user interface, the following steps need to be done:

1. Create a class that extends the class *Application* (of *jgromacs.ui*)

2. Create a configuration XML file that defines the UI of your application. In the constructor of your class set the name of the XML file (*XMLFileName = …*).

3. Within the *main(String[] args)* method of your class create an object of your class and call the *run(args)* method of this object (passing over the args argument of main()). This is the only code main() should contain.

4. Override the method *runCore()* in your class and insert your own code here. Instead of the method *main()*, the method *runCore()* will be the entry point of your Java program.

5. Within *runCore()* you can access the arguments given by the user in the command line. Use the methods *isArgumentGiven()* and *getArgumentValue()*.

6. Within the constructor of your class you can set if your application writes a log file. The name of the log file can be defined here as well. Use the methods *setWritingToLogFile()* and *setLogFileName()*.

7. You may want to write text to the log file on the fly. Use the method *writeToLogFile()* within *runCore()*

(Note that class *TemplateApplication* as well as the example code *UIDemo* demonstrate the correct structure of JGromacs applications.)

What is the XML configuration file for?

The XML configuration file provides a simple way to define meta information about the application (application name, description, author name, version number) and to define the list of command line arguments to be handled by the program. Each command line argument is defined by its argument flag, description and default value.

Syntax of the XML configuration file:

The whole content of the XML file must be between
<application> and </application> tags

First part of XML file:
Between <meta> and </meta> tags: meta information about the application:
  Between <name> and </name> tags: name of the application
  Between <description> and </description> tags: description of the application
  Between <author> and </author> tags: developer of the application
  Between <version> and </version> tags: version of the application

Second part of XML file:
between <arguments> and </arguments> tags: definition of arguments

Each argument is defined between <argument> and </argument> tags;
  Between <flag> and </flag> tags: flag of the argument
  Between <description> and </description> tags: description of the argument
  Between <default> and </default> tags: default value of the argument

<argument> elements also have two attributes:
  type: type of the argument (its value can be “input”, “output” or “setting”)
  optional: is the argument optional? (its value can be “true” or “false”)

(Note that *template.xml* demonstrates the correct structure of XML configuration files.)

Copyright notice