MELTING - development guide

Marine Dumousseau Nicolas Le Novère
lenov@ebi.ac.uk

August 2009

Contents

1 Introduction

This document describes the general layout of the code and can help the developpers to quickly add new models, corrections into the program. The API is documented in the Javadoc.If you want to see the details about the different models and corrections and the program usages, you can read the MELTING documentation.

2 Package layout

- *examples*
  This package contains all the main classes to test the program.
  - *test*
    It contains all the experimental Data and sequences to test the program.
- *melting*
  This package contains the general classes used by most classes of the program. It also contains the main class which runs MELTING. This package also contains the CompatibleMain class which allow the user to use MELTING 5 with the MELTING 4 option syntax (except for the option '-q')
  - *approximativeMethods*
    It contains all the classes which implement an approximative formula to compute the melting temperature.
  - *configuration*
    It contains all the classes which register the option, models and corrections names. It contains the classes which manage the default options and which map the models and correction names with the matching classes.
  - *correctionMethods*
    It regroups some of the base implementations for ion or denaturing agent corrections.
  - *exceptions*
    It contains all the exceptions used in the program.
  - *handlers*
    It contains all the Handler classes necessary to parse the XML files.
  - *ionCorrection*
    - *magnesiumCorrections*
      It contains all the classes which implement a magnesium correction.
    - *mixedNaMgCorrections*
      It contains all the classes which implement a mixed sodium, magnesium correction.
    - *sodiumCorrections*
      It contains all the classes which implement a sodium correction.

    * *sodiumEquivalence*
      It contains all the classes which implement a formula to compute a sodium-equivalent concentration.

- *methodInterfaces*
  It contains all the interfaces.
- *nearestNeighborModel*
  It contains the class which uses a Nearest-Neighbor approach to compute the enthalpy, entropy and melting temperature.
- *otherCorrections*

    * *dmsoCorrections*
      It contains all the classes which implement a DMSO correction.
    * *formamideCorrections*
      It contains all the classes which implement a formamide correction.

- *patternModels*
  It contains the base implementation of each thermodynamic model.

    * *cngPatterns*
      It contains the class which implements a thermodynamic model for CNG repetitions computation.
    * *cricksPair*
      It contains the classes which implement a thermodynamic model for Crick's pairs computation.
    * *InternalLoops*
      It contains the classes which implement a thermodynamic model for internal loop computation.
    * *longBulge*
      It contains the classes which implement a thermodynamic model for long bulge loop computation.
    * *longDanglingEnds*
      It contains the classes which implement a thermodynamic model for long dangling end computation.
    * *secondDanglingEnds*
      It contains the classes which implement a thermodynamic model for two adjacent dangling end computation.
    * *singleBulge*
      It contains the classes which implement a thermodynamic model for single bulge loop computation.
    * *singleDanglingEnds*
      It contains the classes which implement a thermodynamic model for single dangling end computation.
    * *singleMismatch*
      It contains the classes which implement a thermodynamic model for single mismatch computation.

* *specificAcids*
  It contains the classes which implement a thermodynamic model for specific or modified nucleic acids computation.
* *tandemMismatches*
  It contains the classes which implement a thermodynamic model for tandem mismatches computation.
* *wobble*
  It contains the classes which implement a thermodynamic model for inosine and GU base pair computation.

– *sequences*
  It contains the classes which allow to represent nucleic acids and sequences and analyze them.

3 How to add new approximative formulas

1) Creates a new class which implements the MeltingComputationMethod interface or which extends the ApproximativeMode class in the melting.approximativeMethods package.

The ApproximativeMode class already implements the public functions *RegisterMethods getRegister()* and *void setUpVariables(HashMap<String, String> options)* of the MeltingComputationMethod interface. This last method is useful to compute an equivalent sodium concentration if other cations than sodium are entered by the user.

If you extend the ApproximativeMode class, you just need to implement the public functions *ThermoResult computesThermodynamics()* and *boolean isApplicable()* of the MeltingComputationMethod interface. These methods are respectively important to compute the melting temperature with the approximative formula and to define the conditions of application of this formula. (read the Javadoc for further information). You also can override the different ApproximativeMode methods.

If you don't extend the ApproximativeMode class, you have to implement all the MeltingComputationMethod methods. (*RegisterMethods getRegister()*, *void setUpVariables(HashMap<String, String> options)*, *ThermoResult computesThermodynamics()* and *boolean isApplicable()*).

Don't forget to add a private static String as instance variable of the class. This String must represent the approximative formula and must be printed when the verbose mode is required by the user (see the following example).

```
// Create a private static String which represents the
// approximative formula
private static String temperatureFormula = "formula";
```

```
[...]

public ThermoResult computesThermodynamics(){

// To print the article reference of the approximative
// formula if the verbose mode is required.
OptionManagement.meltingLogger.log(Level.FINE, "from
                                Ahsen et al. (2001)");

// To print the approximative formula (the private
 // static String)
OptionManagement.meltingLogger.log(Level.FINE,
                                  temperatureFormula);

[...]
}
```

2) Register the approximative formula name and the class which represents it in the RegisterMethods class (melting.configuration package). You have to add in the function *private void initialiseApproximativeMethods()* of RegisterMethods this following line :

```
private void initialiseApproximativeMethods(){
[...]

// To map the formula name to the class which
// implements it.
        approximativeMethod.put("formula-Name",
                                  ClassName.class);
}
```

4 How to add new thermodynamic model

1) Creates a new class which implements the PatternComputationMethod interface or which extends the PatternComputation class in the melting.patternModels package.

If the structure computed by the new class is already registered by the program, you can create your class in the appropriate package (cngPatterns, cricksPair, InternalLoops, longBulge, longDanglingEnds, secondDanglingEnds, singleBulge, singleDanglingEnds, singleMismatch, specificAcids, tandemMismatches or wobble).

The PatternComputation class contains all the base implementations of each PatternComputationMethod method except for this function : *boolean isApplicable(Environment*

environment, int pos1, int pos2).

You have to implement this method to compute the enthalpy and entropy of a motif in the duplex. You also have to override the function *boolean isApplicable(Environment environment, int pos1,int pos2)* to define the conditions of application of the new thermodynamic model.

2) Always register the new model in the RegisterMethod class in the melting.configuration package. Depending on which structure in the duplex your new model computes, you will have to add one of these following lines :

- *New model for Crick's pairs computation*

```
private void initialiseCricksMethods(){
[...]

// To map the model name to the class which
// implements it.
        cricksMethod.put("model-Name",
                          ClassName.class);
}
```

- *New model for single mismatch computation*

```
private void initialiseSingleMismatchMethods(){
[...]

// To map the model name to the class which implements it.
        singleMismatchMethod.put("model-Name", ClassName.class);
}
```

- *New model for tandem mismatches computation*

```
private void initialiseTandemMismatchMethods(){
[...]

// To map the model name to the class which
// implements it.
        tandemMismatchMethod.put("model-Name",
```

```
                               ClassName.class);
}
```

- *New model for GU base pairs computation*

```
private void initialiseWobbleMismatchMethods(){
[...]

// To map the model name to the class which
// implements it.
        wobbleMethod.put("model-Name",
                        ClassName.class);
}
```

- *New model for internal loop computation*

```
private void initialiseInternalLoopMethods(){
[...]

// To map the model name to the class which
// implements it.
        internalLoopMethod.put("model-Name",
                               ClassName.class);
}
```

- *New model for single bulge loop computation*

```
private void initialiseSingleBulgeLoopMethods(){
[...]

// To map the model name to the class which
// implements it.
        singleBulgeLoopMethod.put("model-Name",
                                  ClassName.class);
}
```

- *New model for long bulge loop computation*

```
private void initialiseLongBulgeLoopMethods(){
[...]

// To map the model name to the class which
// implements it.
        longBulgeLoopMethod.put("model-Name",
                                ClassName.class);
}
```

- *New model for single dangling end computation*

```
private void initialiseSingleDanglingEndMethods(){
[...]

// To map the model name to the class which
// implements it.
        singleDanglingEndMethod.put("model-Name",
                                    ClassName.class);
}
```

- *New model for double dangling end computation*

```
private void initialiseDoubleDanglingEndMethods(){
[...]

// To map the model name to the class which
// implements it.
        doubleDanglingEndMethod.put("model-Name",
                                    ClassName.class);
}
```

- *New model for long dangling end computation*

```
private void initialiseLongDanglingEndMethods(){
[...]

// To map the model name to the class which
// implements it.
```

```
        longDanglingEndMethod.put("model-Name",
                                  ClassName.class);
}
```

- *New model for CNG repeats computation*

```
private void initialiseCNGRepeatsMethods(){
[...]

// To map the model name to the class which
// implements it.
        CNGRepeatsMethod.put("model-Name",
                            ClassName.class);
}
```

- *New model for inosine computation*

```
private void initialiseInosineMethods(){
[...]

// To map the model name to the class
// which implements it.
        inosineMethod.put("model-Name",
                          ClassName.class);
}
```

- *New model for azobenzene computation*

```
private void initialiseAzobenzeneMethods(){
[...]

// To map the model name to the class
// which implements it.
        azobenzeneMethod.put("model-Name",
                            ClassName.class);
}
```

- *New model for locked nucleic acid computation*

```
private void initialiseLockedAcidMethods(){
[...]

// To map the model name to the class which
// implements it.
        lockedAcidMethod.put("model-Name",
                                  ClassName.class);
}
```

- *New model for hydroxyadenosine computation*

```
private void initialiseHydroxyadenosineMethods(){
[...]

// To map the model name to the class which
// implements it.
        hydroxyadenosineMethod.put("model-Name",
                                        ClassName.class);
}
```

3) Create a *public static String defaultFileName* as instance variable of the class. It represents the name of the XML file containing the thermodynamic parameters for this model. You must print it if the user requires the verbose mode. You can create another *public static String* which contains the thermodynamic formula of the model and print it during the verbose mode.

For each message you want to print during the verbose mode, you must write this line : *OptionManagement.meltingLogger.log(Level.FINE, "message to print");*

4) You always must override or implement this function : *void initialiseFileName(String methodName)*. It is necessary to write that the new class can use the thermodynamic parameters of its default File or use the thermodynamic parameters of another file required by the user.

```
@Override
public void initialiseFileName(String methodName){
super.initialiseFileName(methodName);

if (this.fileName == null){
this.fileName = defaultFileName; // The public static String
```

```
                                       // of this class.
}
}
```

Some base implementations have been written for some non specific thermodynamic models, maybe your new class can extend one of the following base implementations. (but you always have to do the steps 1 to 4)

4.0.1 Thermodynamic model for Crick's pairs computation

If it is possible, you directly can create a new class which extends the CricksNNMethod class. (melting.patternModels.cricksPair)

The CrickNNMethod class already implements the following public functions of the PatternComputationMethod interface.: *ThermoResult computeThermodynamics(NucleotidSequences sequences,int pos1, int pos2, ThermoResult result)*, *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2)*, and *ThermoResult computesHybridizationInitiation(Environment environment).*

A CricksNNMethod can compute the enthalpy and entropy of a perfectly matching structure by adding the thermodynamic parameters of each Crick's base pair. The implemented function *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2)* can determine if a thermodynamic parameter for one of the Crick's pair is missing. Finally, the implemented function *ThermoResult computesHybridizationInitiation(Environment environment)* is the base implementation of the hybridization initiation computation and the symetry correction for self complementary sequences.

If the hybridization initiation can be computed with the function *public ThermoResult computesHybridizationInitiation(Environment environment)* of one of the following classes : DecomposedInitiation or GlobalInitiation, you directly can create a new class which extends DecomposedInitiation or GlobalInitiation.

4.0.2 Thermodynamic model for single bulge loop computation

If it is possible, you directly can create a new class which extends the GlobalSingleBuleLoop class (melting.patternModels.singleBulge).

The GlobalSingleBuleLoop class already implements the following public functions of the PatternComputationMethod interface.: *ThermoResult computeThermodynamics(NucleotidSequences sequences,int pos1, int pos2, ThermoResult result)*, and *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2).*

A GlobalSingleBuleLoop can compute the enthalpy and entropy of a single bulge loop by adding the thermodynamic parameters for the trinucleotide containing the single bulge loop. The implemented function *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2)* can determine if a thermodynamic parameter for the trinucleotide containing the single bulge loop is missing.

Finally, the implemented function *int[] correctPositions(int pos1, int pos2, int duplexLength)* is necessary to take into account the adjacent base pairs of the single bulge loop.

4.0.3 Thermodynamic model for inosine computation

If it is possible, you directly can create a new class which extends the InosineNNMethod class (melting.patternModels.wobble).

The InosineNNMethod class already implements the following public functions of the PatternComputationMethod interface.: *ThermoResult computeThermodynamics(NucleotidSequences sequences,int pos1, int pos2, ThermoResult result)*, and *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2)*.

An InosineNNMethod can compute the enthalpy and entropy of a Crick's pair containing an inosine base by adding the thermodynamic parameters for each Crick's pair containing an inosine base. The implemented function *isMissingParameters(NucleotidSequences sequences, int pos1, int pos2)* can determine if a thermodynamic parameter for one of the Crick's pair containing an inosine base is missing.

Finally, the implemented function *int[] correctPositions(int pos1, int pos2, int duplexLength)* is necessary to take into account the adjacent base pairs of the base pair containing the inosine.

5 How to create a new duplex structure

1) Create a new package with the name of the structure.

2) Create a new instance variable *private static HashMap<String, Class<? extends PatternComputationMethod» newStructureMethod* of the class RegisterMethods in the melting.configuration package.

```
/**
* HasMap newStructureMethod : contains all the methods
* for the new structure computation.
*/
private static HashMap<String, Class<? extends
        PatternComputationMethod>> newStructureMethod =
new HashMap<String, Class<? extends PatternComputationMethod>>();
```

3) Create a new method in the RegisterMethod class to initialise the *<String, Class<? extends PatternComputationMethod» newStructureMethod* you created. It must contains all the relationships between the new model names and the matching implemented class:

```
private void initialisenewStructureMethods(){
newStructureMethod.put("model1-Name", classModel1-Name.class);
newStructureMethod.put("model2-Name", classModel2-Name.class);
newStructureMethod.put("model3-Name", classModel3-Name.class);
[...]
}
```

4) Call this method in the constructor of RegisterMethod :

```
public RegisterMethods(){
[...]

initialisenewStructureMethods();
}
```

5) Create a new *public static final String* as instance variable of the OptionManagement class in the melting.configuration package. This String represents the new option name you choose to change the default model used to compute the new structure.

```
/**
* Option name to choose another method to compute the
* new structure.
*/
public static final String newStructureOption-Name =
                                        "option-name";
```

6) Fix the default model name to use for each type of hybridization. You have to add the following lines into the following methods of OptionManagement :

```
/**
* initialises the DNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialisesDNADefaultOptions() {
[...]
```

```
this.DNADefaultOptions.put(newStructureOption-Name,
                          "DNA-defaultModel-Name");

}

/**
* initialises the RNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseRNADefaultOptions() {
[...]
this.RNADefaultOptions.put(newStructureOption-Name,
                          "RNA-defaultModel-Name");

}

/**
* initialises the hybridDefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseHybridDefaultOptions() {
[...]
this.hybridDefaultOptions.put(newStructureOption-Name,
                         "DNA/RNA-defaultModel-Name");

}

/**
* initialises the mRNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseMRNADefaultOptions() {
[...]
this.mRNADefaultOptions.put(newStructureOption-Name,
                          "mRNA-defaultModel-Name");

}
```

7) Register the new option in the HashMap *registerPatternModels* of OptionManagement. You just have to add this following line into the method *private void initialiseRegisterPatternModels()* of OptionManagement :

```
/**
* Initialises the registerPatternModels HashMap of the OptionManagement objec
```

```
*/
private void initialiseRegisterPatternModels(){
[...]

registerPatternModels.add(newStructureOption-Name);
}
```

8) Add in the method *private void initialisePatternModels()* of RegsiterMethods the following line to register the new structure.

```
private void initialisePatternModels(){
[...]

// It creates a relationship between the option name
// for the new structure and the HashMap containing
// the models and the class which can compute the new
// structure.
PatternModel.put(OptionManagement.newStructureMethod,
                                  newStructureMethod);
```

9) Add a new method in the NucleotiSequences class in the melting.sequences package to be abble to recognize if a structure between two positions in the duplex matches the new structure you created.

```
// new method to recognize the new structure in the duplex
public boolean isNewStructure(int pos1, int pos2){

[if the subsequences between the positions pos1 and pos2
in the duplex match the new structure, you must return true.]

}
```

10) Create a new *private PatternComputationMethod* instance variable in the NearestNeighborMode class in the melting.nearestNeighborMode package. This new instance represents an object PatternComputationMethod which is a new instance of one of your implemented class which can compute the new structure.

```
/**
* PatternComputationMethod newStructureMethod : represents
* the model for new structure computation.
*/
private PatternComputationMethod newStructureMethod;
```

11) Create a new method in the NearestNeighborMode class to initialise the PatternComputationMethod newStructureMethod :

```
private void initialiseNewStructureMethod(){
// Get the option name (public static final instance
// variable of OptionManagement) which allows to change
// the model to compute the new structure.
String optionName = OptionManagement.newStructureOption-Name;

// Get the model name (model name) which allows to change
// the model to compute the new structure and initialise the
// PatternComputationMethod newStructureMethod.
String methodName = this.environment.getOptions().get(optionName);
this.newStructureMethod = initialiseMethod(optionName, methodName);
}
```

12) If the new structure you added contains perfectly matching base pairs, maybe you have to change the method *private int [] getPositionsPattern(int pos1)* of the NearestNeighborMode class.

This method defines the positions of a structure in the duplex (a perfectly matching structure or a structure composed of non perfectly matching base pairs or composed of modified nucleic acid(s)).

If you need the adjacent base pairs to the non perfectly matching base pair or the modified nucleic acid, you can add a method which corrects the positions of the structure in the duplex, in the class computing the new structure. (see the following example and the Javadoc)

```
private int[] correctPositions(int pos1, int pos2,
                                int duplexLength){
if (pos1 > 0){
pos1 --;
}
if (pos2 < duplexLength - 1){
pos2 ++;
}
int [] positions = {pos1, pos2};
return positions;
}
```

13) Change the method *private PatternComputationMethod getAppropriatePatternModel(int [] positions)* of NearestNeighborMode to add your new structure initialisation.

```
// Method to get the adapted PatternComputationMethod to
// compute the structure defined at the positions int []
// positions.
private PatternComputationMethod getAppropriatePatternModel
                                        (int [] positions){

// if the new structure is always a terminal structure, you
// can change the method here.
if (positions[0] == 0 || positions[1] ==
         environment.getSequences().getDuplexLength() - 1){

[...]
  // call the NucleotidSequences method to recognize the identity
  // of the new structure
  else if (environment.getSequences().isNewStructure(positions[0],
                                                  positions[1]))){

    if (this.newStructureMethod == null){
initialiseNewStructureMethod(); // initialise the
                                //PatternComputationMethod
                                // newStructureMethod
}
return this.newStructureMethod;
  }
}

// if the structure is not always a terminal structure, you can
// change the method here.
[...]

  // call the NucleotidSequences method to recognize the identity
  // of the new structure
  else if (environment.getSequences().isNewStructure(positions[0],
                                                  positions[1]))){

    if (this.newStructureMethod == null){
initialiseNewStructureMethod(); // initialise the
                                //PatternComputationMethod
                                // newStructureMethod
}
return this.newStructureMethod;
  }
  [...]
}
```

14) Create the new class(es) representing your model(s) for the new structure as it is explained in the section *How to add new thermodynamic model*.

6 How to add new nucleic acids

1) Add the name to the new nucleic acid in the SpecificAcidNames enum in the melting.sequences package :

```
public enum SpecificAcidNames {

inosine, lockedNucleicAcid, hydroxyadenine, azobenzene,
newNucleicAcidName
}
```

2) Register the new nucleic acid syntax in the ArrayList *existingNucleicAcids* of BasePair in the melting.sequences package. You just have to add the following line into the method *public static void initialiseNucleicAcidList()* of BasePair :

```
/**
* initialises the ArrayList existingNucleicAcids of the
* BasePair class.
*/
public static void initialiseNucleicAcidList(){
[...]

// You have to choose a syntax (String representation)
// of the new nucleic acid in the String sequence.
// It is this syntax which will be recognized by the
// program when it analyzes the sequences.
existingNucleicAcids.add("newAcid-syntax");
}
```

3) Create a relationship between the nucleic acid syntax in the ArrayList *existingNucleicAcids* of BasePair and the nucleic acid name registered in the SpecificAcidNames enum. You have to add the following line into the method *public static void initialiseModifiedAcidHashmap()* of NucleotidSequences in the melting.sequences package :

```
/**
* initialises the HasMap modifiedAcidNames of the
* NucleotiSequences class.
```

```
*/
public static void initialiseModifiedAcidHashmap(){
[...]

modifiedAcidNames.put("newAcid-syntax",
    SpecificAcidNames.newNucleicAcidName);
}
```

4) Create a new class to manage the computation of the new nucleic acid as it is explained in the section *How to add new thermodynamic model.* (a new nucleic acid is considered as a new structure in the computation of the enthalpy and entropy of the Crick's pair containing the new nucleic acid.)

7 How to add new corrections for Na, Mg, K, Tris, DMSO and/or formamide

7.1 New ion correction

1) Create a new class which implements the CorrectionMethod interface or, if it is adapted, which extends the EntropyCorrection class in the melting.correctionMethods package.

The class must be created in the adapted package : melting.ionCorrection.magnesiumCorrection if it is a new magnesium correction, melting.ionCorrection.mixedNaMgCorrection if it is a new mixed monovalent correction, magnesium correction or melting.ionCorrection.sodiumCorrection if it is a new sodium correction.

If you just implement the CorrectionMethod interface, you have to implement the public methods *boolean isApplicable(Environment environment)* and *ThermoResult correctMeltingResults(Environment environment)* The first method is important to define the conditions of application of the ion correction and the second is important to correct the computed melting temperature.

The EntropyCorrection is a base implementation for ion corrections which directly correct the computed entropy and then compute te melting temperature. If you extend EntropyCorrection, you have to override the public method *boolean isApplicable(Environment environment)* to define the conditions of application of the ion correction.

The public method *ThermoResult correctMeltingResults(Environment environment)* is already implemented by EntropyCorrection but you have to override the method *protected double correctEntropy(Environment environment)* to correct the computed entropy.

2) Register the ion correction name and the class which represents it in the RegisterMethods class (melting.configuration package). You have to add into the function *private void initialiseIonCorrectionMethod()* of RegisterMethods this following line :

```
private void initialiseIonCorrectionMethod(){
[...]

ionCorrection.put("sodiumCorrection-Name",
                              Class-Name.class);
}
```

3) Don't forget to add a *private static String* instance variable in your class. This String represents the correction formula you applied to the computed melting temperature or the computed entropy and must be printed if the verbose mode is required by the user.

```
// Create a private static String which represents the
// correction formula
private static String correctionFormula = "formula";

[...]

// To print the article reference of the correction
// formula if the verbose mode is required.
OptionManagement.meltingLogger.log(Level.FINE, "article
                           reference of the correction");

// To print the correction formula (the private static String)
OptionManagement.meltingLogger.log(Level.FINE,
                                  correctionFormula);

[...]
}
```

4) In case of sodium corrections, you can use the method *public static computesNaEquivalent(environment)* of the Helper class in melting package to convert the sodium concentration entered by the user into a sodium equivalent concentration which takes into account the other cations entered by the user.

```
double NaEq = Helper.computesNaEquivalent(environment);
```

7.2 New sodium equivalence formula

1) Create a new class (in the melting.ionCorrection.sodiumEquivalence package) which implements the SodiumEquivalenceMethod interface or, if it is adapted, which extends the SodiumEquivalent class in the melting.ionCorrection.sodiumEquivalence package.

If you just implement the SodiumEquivalenceMethod interface, you have to implement the public methods *double computeSodiumEquivalent(double Na, double Mg, double K, double Tris, double dNTP)* and *boolean isApplicable(HashMap<String, String> options)*.

The first is important to compute a sodium equivalence depending on the ions entered by the user. The second method is important to define the conditions of application of the sodium equivalent formula.

The SodiumEquivalent is a base implementation for sodium equivalence computation If you extend SodiumEquivalent, you have to override the public method *boolean isApplicable(HashMap<String, String> options)* to define the conditions of application of the sodium equivalence. The public method *double computeSodiumEquivalent(double Na, double Mg, double K, double Tris, double dNTP)* has to be implemented to compute the sodium equivalence.

2) Register the sodium equivalence name and the class which represents it in the RegisterMethods class (melting.configuration package). You have to add into the function *private void initialiseNaEqMethods()* of RegisterMethods this following line :

```
private void initialiseNaEqMethods(){
[...]

NaEqMethod.put("sodiumEquivalence-Name", Class-Name.class);
}
```

3) Don't forget to add a *private static String* instance variable in your class. This String represents the correction formula you used to compute the sodium equivalent concentration and must be printed if the verbose mode is required by the user.

```
// Create a private static String which represents the
// sodium equivalence formula.
```

```
private static String equivalenceFormula = "formula";

[...]

// To print the article reference of the sodium equivalence
// formula if the verbose mode is required.
OptionManagement.meltingLogger.log(Level.FINE, "article
                                reference of the formula");

// To print the correction formula (the private static String)
OptionManagement.meltingLogger.log(Level.FINE,
                                      equivalenceFormula);

[...]
}
```

7.3 New DMSO and formamide corrections

1) Create a new class which implements the CorrectionMethod interface or, if it is adapted for a new DMSO correction, which extends the DNADMSOCorrections class in the melting.correctionMethods package. You must create your class int the adapted package : melting.otherCorrections.dmsoCorrections package if it is a DMSO correction or melting.otherCorrections.formamideCorrections package if it is a formamide correction.

If you just implement the CorrectionMethod interface, you have to implement the public methods *boolean isApplicable(Environment environment)* and *ThermoResult correctMeltingResults(Environment environment)* The first method is important to define the conditions of application of the correction and the second is important to correct the computed melting temperature.

The DNADMSOCorrections is a base implementation for DMSO corrections and is focused on DNA sequences. If you extend DNADMSOCorrections, you have to override the public method *boolean isApplicable(Environment environment)* to define the conditions of application of the DMSO correction. The public method *ThermoResult correctMeltingResults(Environment environment)* has to be implemented to compute the DMSO correction.

2) Register the correction name and the class which represents it in the RegisterMethods class (melting.configuration package). You have to add into one of these functions of RegisterMethods : *private void initialiseDMSOCorrectionMethod()* or *private void initialiseFormamideCorrectionMethod()*. this following line :

```
/**
* initialises the DMSOCorrectionMethod HashMap of the
* RegisterMethods object.
*/
private void initialiseDMSOCorrectionMethod(){
[...]

DMSOCorrection.put("DMSOCorrection-Name",
                           Class-Name.class);
}

/**
* initialises the formamideCorrectionMethod HashMap of the
* RegisterMethods object.
*/
private void initialiseFormamideCorrectionMethod(){
[...]
formamideCorrection.put("formamideCorrection-Name",
                              Class-Name.class);
}
```

3) Don't forget to add a *private static String* instance variable in your class. This String represents the correction formula must be printed if the verbose mode is required by the user.

```
// Create a private static String which represents the
// correction formula.
private static String correctionFormula = "formula";

[...]

// To print the article reference of the correction
// formula if the verbose mode is required.
OptionManagement.meltingLogger.log(Level.FINE, "article
                              reference of the formula");

// To print the correction formula (the private static String)
OptionManagement.meltingLogger.log(Level.FINE,
                                      correctionFormula);

[...]
}
```

8 How to add new ion and denaturing agent species

1) Create a new method in the Environment class from the melting package. This method must facilitate the usage of the new ion or denaturing agent species concentration in the program.

```
public double getNewSpecies() {
if (concentrations.containsKey("newSpecies-Name")){
return concentrations.get("newSpecies-Name");
}
return 0;
}
```

2) If the new species concentration is a "required ion concentration", that's to say the new species can be the only one species in the solution (no other ions are necessary), you have to change the method *private boolean isRequiredConcentrations()* in the Environment class.

```
private boolean isRequiredConcentrations(){
double Na = 0;
double Mg = 0;
double K = 0;
double Tris = 0;

// The new species must be initialised
double NewSpecies = 0;

if (concentrations.containsKey("Na")){
Na = concentrations.get("Na");
}
if (concentrations.containsKey("Mg")){
Mg = concentrations.get("Mg");
}
if (concentrations.containsKey("K")){
K = concentrations.get("K");
}
if (concentrations.containsKey("Tris")){
Tris = concentrations.get("Tris");
}

// To get the concentration of the new species
if (concentrations.containsKey("newSpecies")){
Tris = concentrations.get("newSpecies");
}
```

```
// the new species concentration must be strictly positive
if (Na > 0 || K > 0 || Mg > 0 || Tris > 0 || newSpecies > 0){
return true;
}
return false;
}
```

Now, the future steps depend on the identity of the new species you want to add.

8.1 New ion species

MELTING is currently using the algorithm from Owczarzy et al, 2008 (see the MELTING documentation for the complete reference.)to correct the computed melting temperature depending on the ion concentrations. This algorithm can take into account the effect of monovalent cations and one divalent cation : the magnesium.

1) If the new ion species can be integrated into the algorithm of Owczarzy et al, 2008, you have to change the method *public CorrectionMethod getIonCorrectionMethod (Environment environment)* of the RegisterMethods class in the melting.configuration package. Otherwise, you must report to the following section for *new denaturing agents* even thought the new species is not a denaturing agent.

```
public CorrectionMethod getIonCorrectionMethod
                                  (Environment environment){

// A specific ion correction is required by the user.
if (environment.getOptions().containsKey
                    (OptionManagement.ionCorrection)){
[...]
}

// No specific ion correction is required, the
// algorithm from Owczarzy et al, 2008 will now be
// used. You have to include your new ion species
// here.
else{

// If it is a new monovalent cation, you must change
// the monovalent concentration.
double monovalent = environment.getNa() + environment.getK()
                  + environment.getTris() / 2
                  + environment.getNewSpecies();
```

```
[...]

// Here are the important variable you may have to
// change to integrate your new ion species (if it
// is a divalent cation or other ion and you know a
// relationship between magnesium concentration and
// this new ion species.)
double Mg = environment.getMg() - environment.getDNTP();
double ratio = Math.sqrt(Mg) / monovalent;

[...]
}
}
```

2) If you know a relationship between your new ion species and Na, Mg, Tris or K, don't forget to add your new ion species in the different classes computing a sodium equivalence in the melting.ionCorrection.SodiumEquivalent package. You will have to change the following methods :

In the different classes from the melting.ionCorrection.SodiumEquivalent package.

```
public double computeSodiumEquivalent(double Na, double Mg,
      double K,double Tris, double dNTP, double newSpecies) {

// Change the base implementation in the SodiumEquivalent
// class too.
double NaEq = super.getSodiumEquivalent(Na, Mg, K, Tris,
                     dNTP, double newSpecies, parameter);

[...]

return NaEq;
}
```

In the SodiumEquivalentMethod interface from the melting.methodInterfaces package.

```
public double computeSodiumEquivalent(double Na, double Mg,
            K, double Tris, double dNTP, double newSpecies);
```

In the ApproximativeMode class from the melting.approximativeMethods

```
public void setUpVariables(HashMap<String, String> options) {
```

```
this.environment = new Environment(options);

if (isNaEqPossible()){
if (environment.getMg() > 0 || environment.getK() > 0
    || environment.getTris() > 0
    || environment.getNewSpecies() > 0){
[...]
}
[...]
}
[...]
}
```

In the Helper class from the melting package

```
public static double computesNaEquivalent(Environment
                                          environment){
double NaEq = environment.getNa() + environment.getK()
              + environment.getTris() / 2
              + environment.getNewSpecies();
[...]
}
```

8.2 New denaturing agent species or other species

1) If the new species is an ion which can't be included in the algorithm from Owczarzy et al. 2008 or if it is a new denaturing agent, you have to create a new instance variable of RegisterMethods in the melting.cinfiguration package. The new *private static HashMap<String, Class<? extends CorrectionMethod»* register all the corrections for the new species.

```
/**
* HasMap formamideCorrection : contains all the methods for
* the new species correction.
*/
private static HashMap<String, Class<? extends CorrectionMethod>>
                                   newSpeciesCorrection =
  new HashMap<String, Class<? extends CorrectionMethod>>();
```

2) You have to create a new method in RegisterMethods to initialise the new HasMap
:

```
/**
* initialises the newSpeciesCorrectionMethod HashMap of
* the RegisterMethods object.
*/
private void initialiseNewSpeciesCorrectionMethod(){
[...]
newSpeciesCorrection.put("NewSpeciesCorrection-Name",
                                    ClassName.class);
}
```

3) You have to create a new option to give the possibility to change the correction for the new species. You must add a new *public static final String* in the OptionManagement class to register the name of the new option. (melting.configuration package)

```
/**
* Option name for to change the default correction for the
* new species.
*/
public static final String newSpeciesOption = "Option-Name";
```

4) Choose a default new species correction for each type of hybridization in the following methods of OptionManagement :

```
/**
* initialises the DNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialisesDNADefaultOptions() {
[...]

this.DNADefaultOptions.put(newSpeciesOption,
                     "DNAdefaultCorrection-Name");
}

/**
* initialises the RNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseRNADefaultOptions() {
[...]

this.RNADefaultOptions.put(newSpeciesOption,
```

```
                  "RNAdefaultCorrection-Name");

}

/**
* initialises the hybridDefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseHybridDefaultOptions() {
[...]

this.hybridDefaultOptions.put(newSpeciesOption,
                  "DNARNAdefaultCorrection-Name");

}

/**
* initialises the mRNADefaultOptions HashMap of the
* OptionManagement object.
*/
private void initialiseMRNADefaultOptions() {
[...]

this.mRNADefaultOptions.put(newSpeciesOption,
                  "mRNAdefaultCorrection-Name");

}
```

5) You have to register the new option in the HashMap *registerEnvironmentOptions* of OptionManagement. You just have to add the following line into the method *private void initialiseRegisterEnvironmentOptions()* of OptionManagement :

```
/**
* Initialises the registerEnvironmentOptions HashMap of the
* OptionManagement object.
*/
private void initialiseRegisterEnvironmentOptions(){
[...]

registerEnvironmentOptions.add(newSpeciesOption);

}
```

6) You have to register the new species and the new corrections for it in RegisterMethods (melting.configuration package). You must add the following line to the method *private void initialiseOtherCorrectionMethod()* :

```
/**
* initialises the otherCorrectionMethod HashMap of the
* RegisterMethods object.
*/
private void initialiseOtherCorrectionMethod(){
[...]

// create a relationship between the new option and
//the corrections registered for the new species.
otherCorrection.put(OptionManagement.newSpeciesOption,
                                 newSpeciesCorrection);
}
```

7) You have to complete the method *public ThermoResult computeOtherMeltingCorrections(Environment environment)* of RegisterMethods. This method is important to correct the melting temperature if another ion or denaturing agent species are present :

```
public ThermoResult computeOtherMeltingCorrections(Environment
                                                 environment){
[...]

 // Check if the new species is present in the environment
 if (environment.getNewSpecies() > 0){

// Get the correction associated with the option name of the
// new species.
CorrectionMethod newSpeciesCorrection =
getCorrectionMethod(OptionManagement.newSpeciesCorrection,
environment.getOptions().get(OptionManagement.newSpeciesCorrection));

if (newSpeciesCorrection == null){
throw new NoExistingMethodException("There is no implemented
                                   new species correction.");
}
else if (newSpeciesCorrection.isApplicable(environment)){
environment.setResult
    (newSpeciesCorrection.correctMeltingResults(environment));
}
else {
```

```
throw new MethodNotApplicableException("The new species correction
   is not applicable with this environment
   (option " + OptionManagement.newSpeciesCorrection + ").");
}
}
```

8) Create a new class for the new species corrections as it is explained in the section *How to add new corrections for Na, Mg, K, Tris, DMSO and/or formamide*.

9 How to change the default textitMelting options

You can change the default textitMelting options in the OptionManagement class from the melting.configuration package. There are different default oprions for each type of hybridization.

```
/**
 * initialises the DNADefaultOptions HashMap of the
 * OptionManagement object.
 */
private void initialisesDNADefaultOptions() {
this.DNADefaultOptions.put(NNMethod, "san04");
this.DNADefaultOptions.put(singleMismatchMethod, "allsanpey");
this.DNADefaultOptions.put(tandemMismatchMethod, "allsanpey");
this.DNADefaultOptions.put(internalLoopMethod, "san04");
this.DNADefaultOptions.put(singleDanglingEndMethod, "bom00");
this.DNADefaultOptions.put(doubleDanglingEndMethod, "sugdna02");
this.DNADefaultOptions.put(singleBulgeLoopMethod, "tan04");
this.DNADefaultOptions.put(longDanglingEndMethod, "sugdna02");
this.DNADefaultOptions.put(longBulgeLoopMethod, "san04");
this.DNADefaultOptions.put(approximativeMode, "wetdna91");
this.DNADefaultOptions.put(DMSOCorrection, "ahs01");
this.DNADefaultOptions.put(formamideCorrection, "bla96");
this.DNADefaultOptions.put(inosineMethod, "san05");
this.DNADefaultOptions.put(hydroxyadenineMethod, "sug01");
this.DNADefaultOptions.put(azobenzeneMethod, "asa05");
this.DNADefaultOptions.put(lockedAcidMethod, "mct04");
this.DNADefaultOptions.put(NaEquivalentMethod, "ahs01");

}

/**
 * initialises the RNADefaultOptions HashMap of the
 * OptionManagement object.
 */
```

```
private void initialiseRNADefaultOptions() {
this.RNADefaultOptions.put(NNMethod, "xia98");
this.RNADefaultOptions.put(singleMismatchMethod, "zno07");
this.RNADefaultOptions.put(wobbleBaseMethod, "tur99");
this.RNADefaultOptions.put(tandemMismatchMethod, "tur06");
this.RNADefaultOptions.put(internalLoopMethod, "tur06");
this.RNADefaultOptions.put(singleBulgeLoopMethod, "tur06");
this.RNADefaultOptions.put(longBulgeLoopMethod, "tur06");
this.RNADefaultOptions.put(CNGMethod, "bro05");
this.RNADefaultOptions.put(approximativeMode, "wetrna91");
this.RNADefaultOptions.put(inosineMethod, "zno07");
this.RNADefaultOptions.put(NaEquivalentMethod, "ahs01");
this.RNADefaultOptions.put(DMSOCorrection, "ahs01");
this.RNADefaultOptions.put(formamideCorrection, "bla96");
this.RNADefaultOptions.put(singleDanglingEndMethod, "ser08");
this.RNADefaultOptions.put(doubleDanglingEndMethod, "ser06");
this.RNADefaultOptions.put(longDanglingEndMethod, "sugrna02");

}

/**
 * initialises the hybridDefaultOptions HashMap of the
 * OptionManagement object.
 */
private void initialiseHybridDefaultOptions() {
this.hybridDefaultOptions.put(NNMethod, "sug95");
this.hybridDefaultOptions.put(approximativeMode, "Wetdnarna91");
this.hybridDefaultOptions.put(NaEquivalentMethod, "ahs01");
this.hybridDefaultOptions.put(DMSOCorrection, "ahs01");
this.hybridDefaultOptions.put(formamideCorrection, "bla96");

}

/**
 * initialises the mRNADefaultOptions HashMap of the
 * OptionManagement object.
 */
private void initialiseMRNADefaultOptions() {
this.mRNADefaultOptions.put(NNMethod, "tur06");
this.mRNADefaultOptions.put(NaEquivalentMethod, "ahs01");
this.mRNADefaultOptions.put(DMSOCorrection, "ahs01");
this.mRNADefaultOptions.put(formamideCorrection, "bla96");

}
```

10 XML Files

10.1 General information

All the XML files containing the thermodynamic parameters are in the Data folder. In each file, I put the data set of a scientific article or I collected the complementary data set of several articles.

I have used the name *data* for the main node of each XML file. The name of a *data* subnode is choosen depending on the structure and the model they are made for. You can see the different existing subnode and existing attributes in the following section. The enthalpy and entropy value are in cal/mol and are put as character of the subsubnodes enthalpy and entropy. (see the following example)

```
<data type="crick">
<neighbor sequence="AA/TT">
        <enthalpy>-7900.0</enthalpy>
            <entropy>-22.2</entropy>
    </neighbor>
    <neighbor sequence="AC/TG">
        <enthalpy>-8400.0</enthalpy>
            <entropy>-22.4</entropy>
    </neighbor>

    [...]

     <initiation type="per-G/C">
            <enthalpy>100.0</enthalpy>
            <entropy>-2.8</entropy>
        </initiation>

        <symmetry>
            <enthalpy>0.0</enthalpy>
            <entropy>-1.4</entropy>
        </symmetry>
</data>
```

Each enthalpy and entropy value are stocked into a Thermodynamics object. (see the Javadoc for the Thermodynamics class from melting package)

10.2 List of existing nodes and attributes

10.2.1 Parameters for perfectly matching structures

Node name	Attributes	attributes values
data	type	crick single-mismatch tandem-mismatch long-mismatch modified-nucleotides repeats single-bulge-loop long-bulge-loop single-dangling-end double-dangling-end long-dangling-end wobble

Subnode name	Attributes	attributes values
neighbor	sequence	crick's pair XX/XX
initiation	type	one-GC-Pair all-AT-pairs per-A/T perG/C
terminal	type	per-A/U per-A/T 5-T/A

10.2.2 Parameters for structures containing wobble base pair, single mimsatch, tandem mismatches or internal loop

Subnode name	**Attributes**	**attributes values**
mismatch	sequence	crick's pair XX/XX
		YRR/RRY
		RYY/YYR
		YYR/RYY
		YRY/RYR
		RRY/YYR
	size	X (number of nucleotides)
	type	initiation
		first-non-canonical-pair
	loop	1x2
		1xn-n>2
		2x3
		others-non-2x2
		1x1
	closing	G/C
		C/G
		U/A
		A/U
		not-G/C
closure	type	per-A/U
		per-G/U
asymetry		
penalty	type	G/G-adjacent-AA-or-nonCanonicalPyrimidine
		AG-GA-UU-adjacent-UU-CU-CC-AA
parameters	sequence	A/A
		G/G
		U/U

10.2.3 Parameters for structures containing bulge loop

Subnode name	Attributes	attributes values
bulge	size sequence type	X (number of nucleotides) trinucleotide XXX/XXX initiation
closure	type	per-A/U per-G/U

10.2.4 Parameters for structures containing specific nucleic acids

Subnode name	Attributes	attributes values
modified	sequence type sens	XXX/XX or XXXX/XX trans cys 3 5
closure	type	per-A/U per-G/U
terminal	type	per-I/U

10.2.5 Parameters for structures containing CNG repeats

Subnode name	Attributes	attributes values
CNG	sequence repeats	CNG pattern XXX 2 to 7

10.2.6 Parameters for structures containing dangling end

Subnode name	Attributes	attributes values
dangling	sequence	dangling end XX/X, XXX/X or XXXX/X
	sens	5
		3

10.3 what to change if you add new subnodes or new attributes

The current Handler classes in the package melting.handlers can manage this type of node hierarchy :

```
<data type="strucure-type">
<subnode attribute1="value1" attribute2="value2">
<enthalpy>xx[...]xx.x</enthalpy>
<entropy>xx[...]xx.x</entropy>
</subnode>

    [...]

<subnode sequence="AA/TT">
<enthalpy>xx[...]xx.x</enthalpy>
<entropy>xx[...]xx.x</entropy>
</subnode>
</data>
```

1) You have to register your new attribute in the DataHandler class in the melting.handlers package. You need to change the method *public void endElement(String uri, String localName, String name)* to build the matching key in the dictionnary which will contain the thermodynamic parameters :

```
@Override
public void endElement(String uri, String localName, String name)
throws SAXException {
if (subHandler != null) {
subHandler.endElement(uri, localName, name);
if (subHandler.hasCompletedNode()) {
ThermoHandler handler = (ThermoHandler) subHandler;
String key = name;
if (handler.getAttribut().containsKey("type")) {
key += handler.getAttribut().get("type");
}
```

```
[...]

// Add your new attribute here
if (handler.getAttribut().containsKey("newAttribute-Name")) {
key += "subnode-Name" + handler.getAttribut().get("newAttribute-Name");
}
[...]
}
```

The dictionnary key for each thermodynamic parameter mostly has the following syntax,

Subnode-nameAttribute1Value1Attribute2Value2

but it can be different for some attributes. (see the method in details)

2) You have to create (or change) a method in the DataCollect class from the melting package to more easily get the thermodynamic parameters you need. See the example below :

```
/**
* to get the Thermodynamics object containing the parameters
* for the base pair (base1, base2) next to the mismatching
* base pair.
* @param string base1 : from the sequence (5'3')
* @param string base2 : from the complementary sequence (3'5')
* @return Thermodynamics object containing the parameters for
* the base pair (base1, base2) next to the mismatching base pair.
*/
public Thermodynamics getClosureValue(String base1, String base2){
Thermodynamics s = data.get("closure"+"per-"+base1 + "/"
                            + base2);
return s;
}

// Your method can be similar to the following method
public Thermodynamics getNewThermodynamicParameter1(arg-1, ..., arg-n){
Thermodynamics s = data.get("node-name"+"attribute-1"+value-1
                             +[...]+"attribute-n"+value-n);
return s;
}
```

11 How to change the default ion corrections

You can change the default ion corrections in the method *public CorrectionMethod getIonCorrectionMethod (Environment environment)* of the class RegisterMethod from the melting.configuration package.