Optimisation of silvicultural scenarios under Capsis using Apache tools. Illustration with the Nelder & Mead method.

Gilles Le Moguédec

January 2010
Version 1.00

Contents

1 Introduction

The Caspis script mode allows to run automatically hundreds of silvicultural scenarios. This functionality can be used to optimise these scenarios according to a user-defined criterion.

Here, we first present the use of some existing optimisation tools defined by Apache. These tools are now included in Capsis and can be used in script mode. We restrict this presentation to the Nelder & Mead optimisation method. Other available methods may be presented in further versions of this document.

This presentation ends with some recommandations to adapt these tools for Capsis models. The reader is supposed to be familiar with the Capsis script mode.

2 Starting with optimisation using Apache Tools

2.1 Core of an optimisation program

With the existing optimisation tools, it is very easy to conduct an optimisation. At first, an optimiser object has to be defined. Its technical parameters can be modified. Then, the function to optimise (objective function) has to be defined and also the starting point of optimisation. For the Nelder & Mead optimisation method, this starting point must be completed by the definition of the initial simplex.

Once all these elements are defined, the only thing to do is to launch the optimiser and to save its results that may be analysed later.

Basic instructions would for that be:

```java
// 1-Define a Nelder-Mead optimizer
NelderMead nm = new NelderMead() ;

// 2-Define if the optimisation will be a minimisation or a maximisation
GoalType goal = GoalType.MINIMIZE ;

// 3-Define the function to optimize
// Here, a function that takes two real variables x and y as input
MyFunction fn = new MyFunction() ;

// 4-Initial Parameters and initial simplex;
// here, random starting point. Must be an array.
double x=10*Math.random()-5 , y = 10*Math.random()-5 ;
double[] startPoint = {x,y} ;
// Variations from initial point to generate initial Simplex
double[] deltaParam= {10*Math.random()-5 , 10*Math.random()-5} ;
nm.setStartConfiguration(deltaParam) ;

// 5-Launch the optimizer and save the result
RealPointValuePair result = null ;
result = nm.optimize(fn,goal,startPoint) ;

// 6-Print the result
// The optimal point is contained in an array of double :
System.out.println("x = "+result.getPoint()[0]) ;
System.out.println("y = "+result.getPoint()[1]) ;
// The optimal value is accessed by the method getValue() :
System.out.println("f(x,y) = "+result.getValue()) ;
```

Three special Classes devoted to optimisation are used in this program: **NelderMead**, **GoalType** and **RealPointValuePair**. A fourth one, **SimpleScalarValueChecker** can also be used to control some technical aspects of the optimisation. All of them will be presented in next section.

These classes are called in the program with the following statements :

```java
import org.apache.commons.math.optimization.direct.NelderMead ;
import org.apache.commons.math.optimization.GoalType ;
import org.apache.commons.math.optimization.RealPointValuePair ;
import org.apache.commons.math.optimization.SimpleScalarValueChecker ;
```

The last class necessary for optimisation purposes is an user-defined one. In the previous example it is **MyFunction**. The function class will be detailed later. All we need to know about it for the moment is that it must contain a public method called « value » that computes the value of the function. This function must take an array of double as input and deliver a double as output.

```java
public class MyFunction .... {
	...
	public double value(double[] parameters) ... {
	...
	}
}
```

2.2 *Pre-defined Classes for optimisation*

2.2.1 The NelderMead Class

A **NelderMead** object is an optimiser, that is the program that will apply the Nelder & Mead simplex algorithm to an user-defined function that will be defined later. It is possible to modify some technical parameters used in this algorithm.

Constructors:

There are two constructors for this object :

```
public NelderMead()
public NelderMead(double rho, double khi, double gamma, double sigma)
```

The coefficients that appears in the second constructor (see Figure 1) are respectively :

`rho`	(ρ):	reflection coefficient, default value = 1.00;
`khi`	(χ):	expansion coefficient, default value = 2.00;
`gamma`	(γ):	contraction coefficient, default value = 0.50;
`sigma`	(σ):	shrinkage coefficient, default value = 0.50 .

The first constructor uses default values for these coefficients.

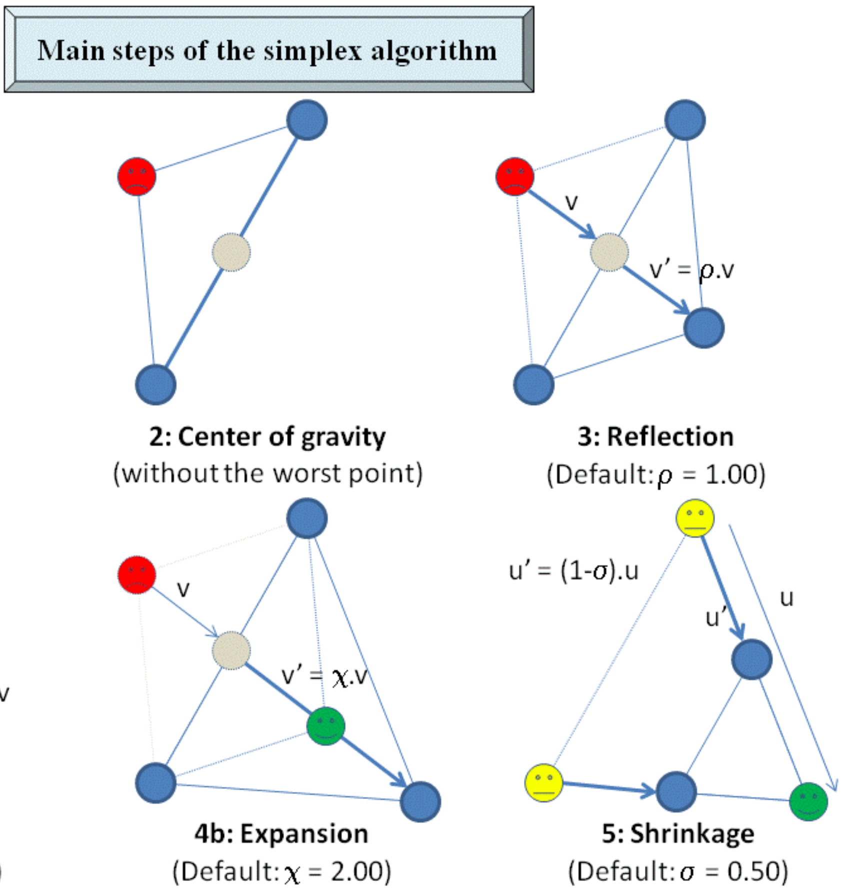

Figure 1: Main steps of the Nealder & Mead simplex algorithm.

Main methods:

Among the methods of this class, the main ones are :

- `public RealPointValuePair optimize(fn,goal,startPoint)`

This method launch the optimizer to an instance `fn` of the user-defined function, in order to maximize or minimize its value, according to the value given to the **GoalType** object `goal`. The starting point coordonates are contained in the array of double `StartPoint`. If convergence is reached, the method returns a **RealPointValuePair** object that contains both the optimal point and the optimal value.

- `public void setStartConfiguration(double[] steps)`

The Nelder & Mead method starts with an initial simplex, that is a set of initial points. This method defines the initial set of points from a starting point (defined elsewhere) and a vector which coordonates will be successively added to the starting point coordonates in order to determine the successive vertexes of the initial simplexe. See Figure 2a. As a consequence, the resulting simplex has edges parallel to each of the main axes of the parameters space. In order for the resulting simplex to have the same dimension than the parameter space, none of the input vector coordonates can be null (but they can be negative).

- `public void setStartConfiguration(double[][] referenceSimplex)`

This is another method to define the initial simplex that uses a basic simplex as input. This simplex is defined by a set of initial points that must be linearly independent. The initial simplex is obtained by translating the whole simplex so that its first point is located at the starting point. See Figure 2b.

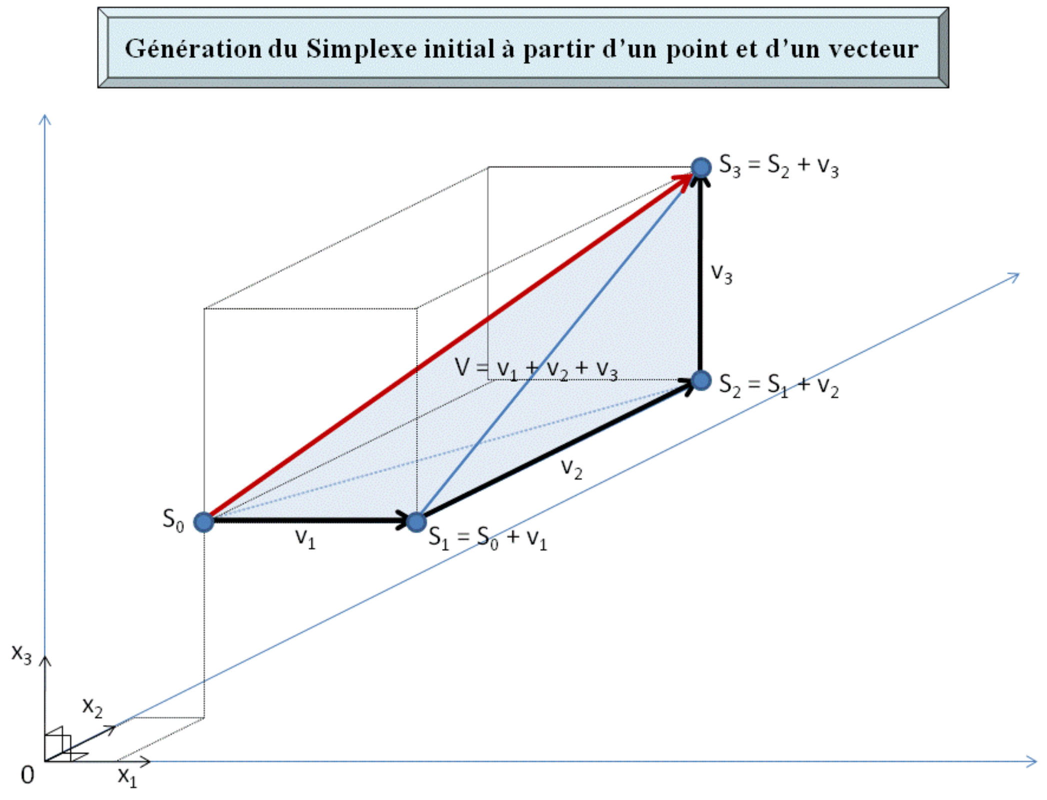

Figure 2a: Generating the initial simplex from a starting point and an vector

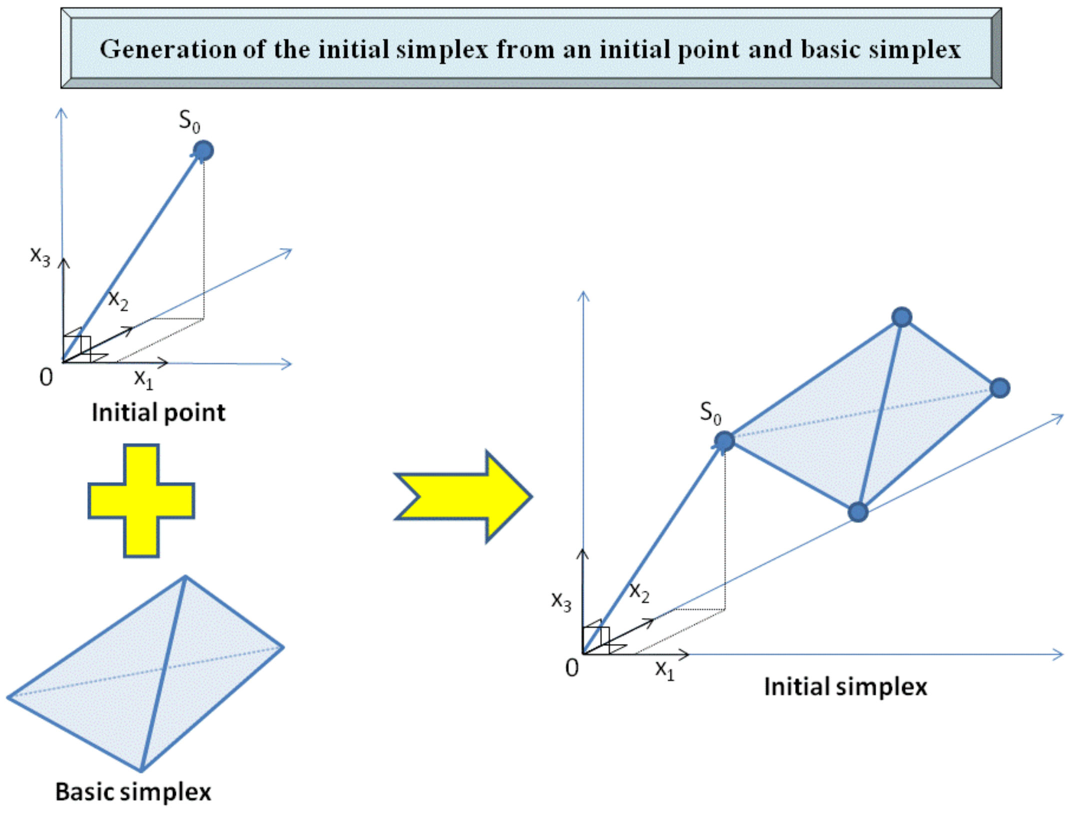

Figure 2b: Generating the initial simplex from a starting point and a basic simplex

Optional methods

- `public void setMaxEvaluations(int maxEvaluations)`

This method allows to modify the maximum number of the function evaluations during optimisation. If the number of function evaluations exceeds this maximum number, the algorithm will stop before convergence. By default, this maximum number is defined by the biggest integer value permitted by the system.

- `public void setMaxIterations(int maxIterations)`

This method allows to modify the maximum number the iterations during optimisation. If the number of iterations exceeds this maximum number, the algorithm will stop before convergence. By default, this maximum number is defined by the biggest integer value permitted by the system.

- public **int** getMaxEvaluations()

This method returns the maximum number permitted for the function evaluations.

- public **int** getMaxIterations()

This method returns the maximum number of iterations allowed.

- public **int** getEvaluations()

This method returns the number of function evaluations performed.

- public **int** getIterations()

This method returns the number of iterations performed.

- `public void setConvergenceChecker(RealConvergenceChecker convChecker)`

A **RealConvergenceChecker** (see later in this section) object contains the definition convergence conditions and tests if convergence has occured. Hence this methods allows to define the convergence criteria. Default convergence criteria are used if this method is not called.

2.2.2 The GoalType Class

This very simple class allows to say if the optimization will be a minimization or a maximization. A **GoalType** object can take only two values MINIMIZE or MAXIMIZE. It is used by the optimize method of the **NelderMead** Class.

2.2.3 The RealPointValuePair Class

This class performs the association between a point (an array of double) and the value of the function at this point (a double). It can be used to catch the result of the optimize method of the **NelderMead** Class if convergence has occured.

Main methods

- `public double[] getPoint()`

Returns the coordonates of the point as an array of double.

- `public double getValue()`

Returns the value of the function zt this point.

2.2.4 The SimpleScalarValueChecker Class

The **SimpleScalarValueChecker** Class contains a method that uses the value of the function between two successive iterations of the optimisation algorithm to check if convergence has occured. Two convergence criteria are computed: an absolute difference criterion and a relative difference criterion. Each of these criteria is associated to a threshold value. As soon as one of the two criteria takes a value lower than its associated threshold, the program considers that convergence has occured.

This class implements the interface **RealConvergenceChecker** where a boolean method called « converged » tests the convergence between the two last evaluated points :

```
public boolean converged(int iteration, RealPointValuePair previous,
RealPointValuePair current)
```

Constructors

- `public SimpleScalarValueChecker()`

Build an instance with the default thresholds: 100 times the lowest striclty positive double allowed by the system fot the absolute difference, 100 times the lowest inversible positive double allowed by the system for the relative difference.

- `public SimpleScalarValueChecker(double absoluteThreshold, double relativeThreshold)`

Build an instance with the specified thresholds.

2.3 Definition of the objective function

The function used as objective function can be contained in any Java Class that implements the Apache interface **MultivariateRealFunction**. It must contain a public method called « value » that computes the value of the function. This function must take an array of double as input and deliver a double as output.

```
public class MyFunction implements MultivariateRealFunction {
      ...
      public double value(double[] parameters) ... {
      ...
      }
}
```

As an example, here is the code for a Java Class that implements the Himmelblau function, a function classicaly used to test optimisation procedures. This 2-variables function can be written as:

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

It presents one local maximum and four local minima.

```
import org.apache.commons.math.analysis.MultivariateRealFunction ;
import org.apache.commons.math.FunctionEvaluationException ;
import java.lang.IllegalArgumentException ;
public class Himmelblau extends Object implements MultivariateRealFunction{

      // Constructor
      public Himmelblau(){}

      // value method with explicit variable names ;
      public double value(double x, double y)
            throws FunctionEvaluationException, IllegalArgumentException{
            try {
                  double f = Math.pow(x*x + y -11,2) + Math.pow(x + y*y -7,2) ;
                  return(f) ;
            } catch (Exception e) {
                  e.printStackTrace();
                  return Double.NaN. ;
            }
      }

      // value method with an array as input ;
      public double value(double[] parametres)
            throws FunctionEvaluationException, IllegalArgumentException{
            double valeur = value(parametres[0],parametres[1]) ;
            return(valeur) ;
        }
}
```

That's all ! However, one can remark that in the previous example, two « value » method have been defined. The first one uses explicit variables names as input instead of an array of double. Since the functions to optimize can be very complicated, it is easier to first write the function with explicit parameters names, and then to overwrite this method by taking an array of double as input instead of individual variables.

3 Improvements of optimisation programs

The objects and methods presented in the previous section are sufficient to build and run an optimisation program under java. However, they may not be sufficient to analyse the results of an optimisation.

For example it is not possible to have access to the iterations details. If an optimal solution has been found, the optimal point and the optimal value can be accessed. In that case the iteration details may not be necessary. But if optimisation has failed, such details can help to understand the problem.

Another point is that the user may have some reasons to modify the definition of convergence. With the current tools, it is possible to control the thresholds used for the absolute and the convergence criteria, but not the way they are written and the fact that convergence occurs as soon as one of them is verified. It is not possible to modify them or to add another criterion.

We will then suggest some improvements to solve these restrictions.

3.1 Improvements of the objective function definition

The main suggested improvement is to add an array to the function Object that will contain a trace of each function calls.

Consequently to this addition, the following elements have to to be added or modified:

- the constructor to initialize the history ;
- the value method must now update the history :
- a getHistory() method to access to this history ;
- a historyToString() method to convert this history to String for outputs;
- in addition, a getHeader() method to return the names of the colums.

Here is the code for the modified **Himmelblau** Class :

```
import org.apache.commons.math.analysis.MultivariateRealFunction ;
import org.apache.commons.math.FunctionEvaluationException ;
import java.lang.IllegalArgumentException ;
import java.util.List ;
import java.util.ArrayList ;
```

```
public class Himmelblau extends Object implements MultivariateRealFunction{

      // Historic of the function calls
      private List<double[]> history ;

      // Modified constructor
      public Himmelblau(){
            history = new ArrayList<double[]>() ;
      }

      // modification of the first value() method
      public double value(double x, double y)
            throws FunctionEvaluationException, IllegalArgumentException{
            // Computation of the function
            try {
                  double f = Math.pow(x*x + y -11,2) + Math.pow(x + y*y -7,2) ;
            // update the history
                  double[] resume = {x , y, f } ;
                  history.add(resume) ;
                  return(f) ;
            } catch (Exception e) {
                  e.printStackTrace();
                  return Double.NaN ;
            }
      }

      // Second value method unchanged
      public double value(double[] parametres)
            throws FunctionEvaluationException, IllegalArgumentException{
            double valeur = value(parametres[0],parametres[1]) ;
            return(valeur) ;
      }

      // Get the history of function calls as a list //
      public List<double[]> getHistory(){
            return(history) ;
      }

      // Return the header of history as a String
```

```
    public static String getHeader(){
        return ("x\ty\tf(x,y)") ;
    }

    // Convert the history of function calls to a String for printing */
    public String historyToString() {
        String s = new String() ;
        for (double[] resume : history){
            String s2 = Double.toString(resume[0]) ;
            for (int i =1 ; i < resume.length ; i++) {
                s2 += "\t"+resume[i] ;
                }
            s2 += "\n" ;
            s += s2 ;
        }
        return(s) ;
    }

    // The same, with the possibility to add a header */
    public String historyToString(boolean header) {
        String s = new String() ;
        if (header) {
            s += this.getHeader() + "\n" ;
        }
        s += this.historyToString() ;
    }
}
```

This modified function allows to print the history of function calls in the outputs of the optimisation program.

3.2 *Replacement of the SimpleScalarValueChecker Class*

The original **SimpleScalarValueChecker** Class lacks of tools to retrieve the details of the optimisation. Unfortutately, most of its elements (fields or methods) are private, so that it is impossible to build an object with accessors through inheritance. Instead, we have copied the original **SimpleScalarValueChecker** and modified it in order to have a better access to its components. Any object that implements the interface **RealConvergenceChecker** can be used.

With this interface, the object has to implement a « converged » method to check if convergence has occured between the two last evaluated points. If the user intends to modify the convergence rules, it is where the modifications have to be done.

Here is the code for this modified class. As for the objective function, a history field has been added with corresponding accessors and toString() methods. Some accessors have been added to access to the other fields of the object.

```
import org.apache.commons.math.util.MathUtils;
import org.apache.commons.math.optimization.RealConvergenceChecker ;
import org.apache.commons.math.optimization.RealPointValuePair ;
import java.util.List ;
import java.util.ArrayList ;

public class ModifiedSimpleScalarValueChecker implements RealConvergenceChecker
{
      /** Default relative threshold. */
      private static final double DEFAULT_RELATIVE_THRESHOLD = 100 *
MathUtils.EPSILON;

      /** Default absolute threshold. */
      private static final double DEFAULT_ABSOLUTE_THRESHOLD = 100 *
MathUtils.SAFE_MIN;

      /** Relative tolerance threshold. */
      private final double relativeThreshold;

      /** Absolute tolerance threshold. */
      private final double absoluteThreshold;

      /** Absolute convergence criterion  */
      private double absoluteConvergence ;

      /** Relative convergence criterion  */
      private double relativeConvergence ;

      /** Has the convergence occured ? */
      private boolean convergence ;

      /** History of the convergence evaluation calls */
      private List<double[]> history ;

      /** Build an instance with default threshold.
      */
      public ModifiedSimpleScalarValueChecker() {
            this.relativeThreshold = DEFAULT_RELATIVE_THRESHOLD;
            this.absoluteThreshold = DEFAULT_ABSOLUTE_THRESHOLD;
            this.absoluteConvergence = Double.NaN ;
            this.relativeConvergence = Double.NaN  ;
            this.convergence = false ;
            this.history = new ArrayList<double[]>() ;
      }

      /** Build an instance with a specified threshold.
      * <p>
      * In order to perform only relative checks, the absolute tolerance
      * must be set to a negative value. In order to perform only absolute
      * checks, the relative tolerance must be set to a negative value.
      * </p>
      * @param relativeThreshold relative tolerance threshold
      * @param absoluteThreshold absolute tolerance threshold
      */
      public ModifiedSimpleScalarValueChecker(final double absoluteThreshold,
                                               final double relativeThreshold) {
            this.relativeThreshold = relativeThreshold;
            this.absoluteThreshold = absoluteThreshold;
            this.absoluteConvergence = Double.NaN  ;
            this.relativeConvergence = Double.NaN  ;
            this.convergence = false ;
            this.history = new ArrayList<double[]>() ;
      }

      /** {@inheritDoc}
      *   Test the convergence criteria by comparing the current iteration to
the previous one.
      *   The convergence is obtained as soon as one of the elementary
convergence criteria
      *   (absolute and relative convergence criterion) is verified.
      */
      public boolean converged(final int iteration,
                                final RealPointValuePair previous,
                                final RealPointValuePair current) {
            final double p          = previous.getValue();
            final double c          = current.getValue();
            final double difference = Math.abs(p - c);
            final double size       = Math.max(Math.abs(p), Math.abs(c));
            this.absoluteConvergence = difference ;
            this.relativeConvergence = (size>MathUtils.SAFE_MIN) ?
difference/size : Double.POSITIVE_INFINITY;
            this.convergence = (difference <= (size * relativeThreshold)) ||
(difference <= absoluteThreshold) ;
            double[] resume = {iteration, this.absoluteConvergence,
this.relativeConvergence } ;
            history.add(resume) ;
            return (this.convergence);
      }

      /** Return the threshold for the absolute convergence criterion */
      public double getAbsoluteThreshold(){
            return this.absoluteThreshold ;
      }

      /** Return the threshold for the relative convergence criterion */
      public double getRelativeThreshold(){
            return this.relativeThreshold ;
      }

      /** Return the current value of the absolute convergence criterion */
      public double getAbsoluteConvergence(){
            return this.absoluteConvergence ;
      }

      /** Return the current value of the relative convergence criterion */
      public double getRelativeConvergence(){
            return this.relativeConvergence ;
      }

      /** Return the current convergence status */
      public boolean getConvergence(){
            return this.convergence ;
      }

      /** Get the history of convergence criteria as a list */
      public List<double[]> getHistory(){
            return(history) ;
      }

      /** Return the history of convergence criteria as a String */
```

```
    public String historyToString() {
          String s = "Iteration\tAbsoluteConvergence\tRelativeConvergence\n" ;
          for (double[] resume : history){
                s+=(resume[0]+"\t"+resume[1]+"\t"+resume[2]+"\n") ;
          }
          return(s) ;
    }

}
```

3.3 Functionalities added by these modifications

The **ModifiedSimpleScalarValueChecker** Class is now used instead of the original **SimpleScalarValueChecker** class by :

```
ModifiedSimpleScalarValueChecker convergenceCriteria
                               = new ModifiedSimpleScalarValueChecker() ;
nm.setConvergenceChecker(convergenceCriteria) ;
```

where « nm » contains the optimiser object.

The main functionality added by these modifications are the program outputs. For example, if the objective function object in the program is called « fn » , the history of function calls and of convergence evaluation can now be printed :

```
fn.historyToString(true) ;
convergenceCriteria.historyToString() ;
```

In addition, if convergence has not occured, the best point encoutered during the optimisation process can now be retrieved by searching the best value of the function evaluations history.

4 Adaptation for an optimisation under Capsis

A first, do not forget to include all the modified tools into Capsis packages.

Apart this detail, there is nothing to modify in the main optimisation program. The only element to modify is the definition of the objective function

In order to be runable with the Nelder-Mead method, this function must implement the **MultivariateRealFunction** Class. The « value » method of this interface will need to call:

1. a Capsis Script that can run a whole Capsis project from a finite set of real parameters;
2. a method to compute the objective function associated to a Capsis project.

Suppose that you have already defined a method « runProject(parameter1, parameter2, parameter3) » that returns a Capsis Project from a set of 3 scenario parameters (parameter1, parameter2, parameter3) and another method evaluateProject(project) that returns a double as the result of the evaluation of the Capis project given as input. A « value » method will be written with the following skeletton:

```
....
import capsis.kernel.Project;
import capsis.kernel.Step;
import capsis.script.C4Script ;
....
public class FunctionToOptimize implements MultivariateRealFunction {
      ....
      public double value(double parameter1, double parameter2, double
parameter3) throws FunctionEvaluationException, IllegalArgumentException{
            ...
            Project currentProject = runProject(parameter1, parameter2,
parameter3) ;
            double performance = evaluateProject(currentProject) ;
            ...
            return performance ;
      }
      ....
}
```

The previous code would return an error: the value method that takes an array of double as input has not yet been defined. However it can not be directly done by overwriting the value method like that:

```
public double value(double[] parameterArray) ...{
      return value(parameterArray[0], parameterArray[1], parameterArray[2]) ;
}
```

Generally, the scenario parameters can take their values within a given interval like [a;b] or are restricted to positive values. The NelderMead method run on points which coordonates can take any real value. At first, methods to convert real values to parameters values must be written.

```
      double realToParameter1(double []realArray) {....}
      double realToParameter2(double []realArray) {....}
      double realToParameter1(double []realArray) {....}
```

With these additional methods, the needed value method can be written as :

```
public double value(double[] realArray) ...{
      double param1 = realToParameter1(realArray) ;
      double param2 = realToParameter1(realArray) ;
      double param3 = realToParameter1(realArray) ;
      return value(param1, param2, param3) ;
}
```

With this method, the optimal set of scenario parameters will be given as an optimal array of reals that is not directly interpretable. The previous methods will be used in the optimisation script to convert the optimal array into an understandable set of scenario parameters :

```
double optimalParam1 = fn.realToParameter1(optimalArray) ;
double optimalParam2 = fn.realToParameter2(optimalArray) ;
double optimalParam3 = fn.realToParameter3(optimalArray) ;
```

where « fn » is an instance of the objective function defined in the optimisation script.

The last point would be if the user intend to start the optimisation from a given set of parameters. In this case, the reciprocal operation should be performed. The class that defines the objective function should contain a last method that perform the reciprocal conversion:

```
double[] parameterToArray(parameter1, parameter2, parameter3) {...}
```

An example of mathematical transformation that converts a real to a positive one would be the exponential function, which reciprocal is the logarithm. Other transformations can be used for the conversion of a real to a bounded value such as ArcTangent function for example.

That's all. Additional elements or methods can be defined to improve the outputs, but the previously defined elements are sufficient to use the Apache tools for silvivultural optimisation purposes under Capsis.