How to use jas.statistics package

Michele Sonnessa
(sonnessa@di.unito.it)

1.How JAS retrieves data from objects..... 2
2.The encapsulation system..... 4
3.How statistics are updated..... 6
4.CrossSection object..... 7
5.The Series object..... 8
6.The statistic functions..... 9

The *jas.statistic* package is a statistical library specifically designed to be executed in a simulation context. Since data sets collected from simulations are frequently updated and sometimes data structures change at runtime, the code is optimize to reduce memory occupancy and CPU time consumption.

The present guide shows step by step the package features and its use.

The package structure is composed by three sections:

1. the *jas.statistics* package contains the main interfaces and classes;
2. the *jas.statistics.reflectors* package contains classes that retrieve data from common java objects;
3. the *jas.statistics.functions* package contains the functions that compute statistics on data sets. The statistics computing algorithms are mainly based on the *cern.jet.stat* package.

1. How JAS retrieves data from objects

In order to compute statistics, a statistical computer must be able to dynamically collect data from simulation objects. It represents a problem, since the statistical library classes do not know the structure of the target objects (designed by users) and so they cannot access their internal data using instructions like myObject.getDatum().

The easiest solution to solve the problem is represented by the use of the reflectors contained by the jas.statistics.refelectors package. These classes use the Java Reflection to inspect dynamically the target objects' structure and data.

Let's consider an example. An agent represented by the class MyAgent contains two integer variables called *age* and *children*, as described by the following code:

```
public class MyAgent {
  int income, age;
}
```

Suppose that the user needs to create a time series of the variable *income* for this agent. A typical instruction could be:

```
MyAgent myAgent = new MyAgent();
Series.Integer series = new Series.Integer(myAgent, "income", false /*a var*/);
```

The constructor of the Series.Integer class automatically creates an IntegerInvoker object that reads the income variable within an instance of the MyAgent class. This way, every time the time series object has to be updated (with the updateSource() method), the current value of agent's income is appended to the series internal data array.

The reflection mechanism is very simple and elegant but, unfortunately, very inefficient, since it is about 20 time slower than a native direct access! So, in order to increase the speed[1], we need to access objects natively.

JAS defines a method, for direct access, based on the I*Source[2] interfaces. Each object containing interesting data to be collected should implement one or more of the following interfaces, according to the type of data to be provided.

Single value output	Multiple value output (array)
IDoubleSource	IDoubleArraySource
IFloatSource	IFloatArraySource
ILongSource	ILongArraySource
IIntSource	IIntArraySource

In order to use these interfaces to natively access data inside the MyAgent class, its code has to be modified as follows:

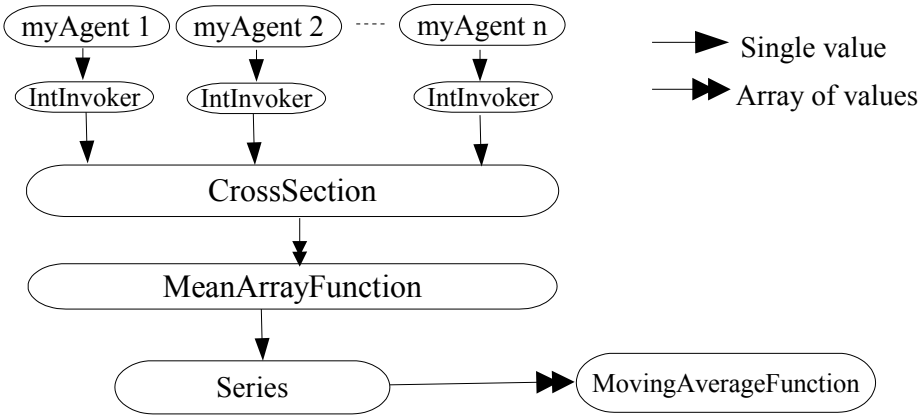
[1] The use of the direct access method improves also the accuracy of the java exception catching.

[2] With the *I*Source* notation, we intend all the interfaces ending with the "*Source"* string, like *IDoubleSource, IFloatSource, …, IIntArraySource, …*, contained in the *jas.statistics* package.

```
public class MyAgent implements jas.statistics.IIntSource {
  public static final int AGE = 0;
  public static final int INCOME = 1;

  int income, age;

  public int getIntValue(int variableID)
  {
    switch (variableID)
    {
      case AGE: return age;
      case INCOME: return income;
    }
  }
}
```



The series object previously defined can be now created using the following instructions:

```
MyAgent myAgent = new MyAgent();
Series series = new Series.Integer(myAgent, MyAgent.INCOME);
```

This way, the series object will now access the target object's variables through its IIntSource interface, simply by passing to its getIntValue method the right constant value (*MyAgent.INCOME*).
Although boring, this mechanism is more efficient than the previous one. It is recommended for long run simulations. However, the choice between using the reflection or the native access is left to the user.

2. The encapsulation system

The I*Source interfaces are used to sequentially encapsulate different computational operations.

Suppose you want to compute, at every simulation time step, the moving average of the mean value of the agents' income. This value might be useful, for instance, to understand if the simulation reached an equilibrium.

In order to obtain the moving average we need to perform the following jobs, at each simulation time step:

1. collect data from all the agents contained in a list;
2. compute the average value of the collected data;
3. store the value into a time series object;
4. using the series, compute the current moving average.

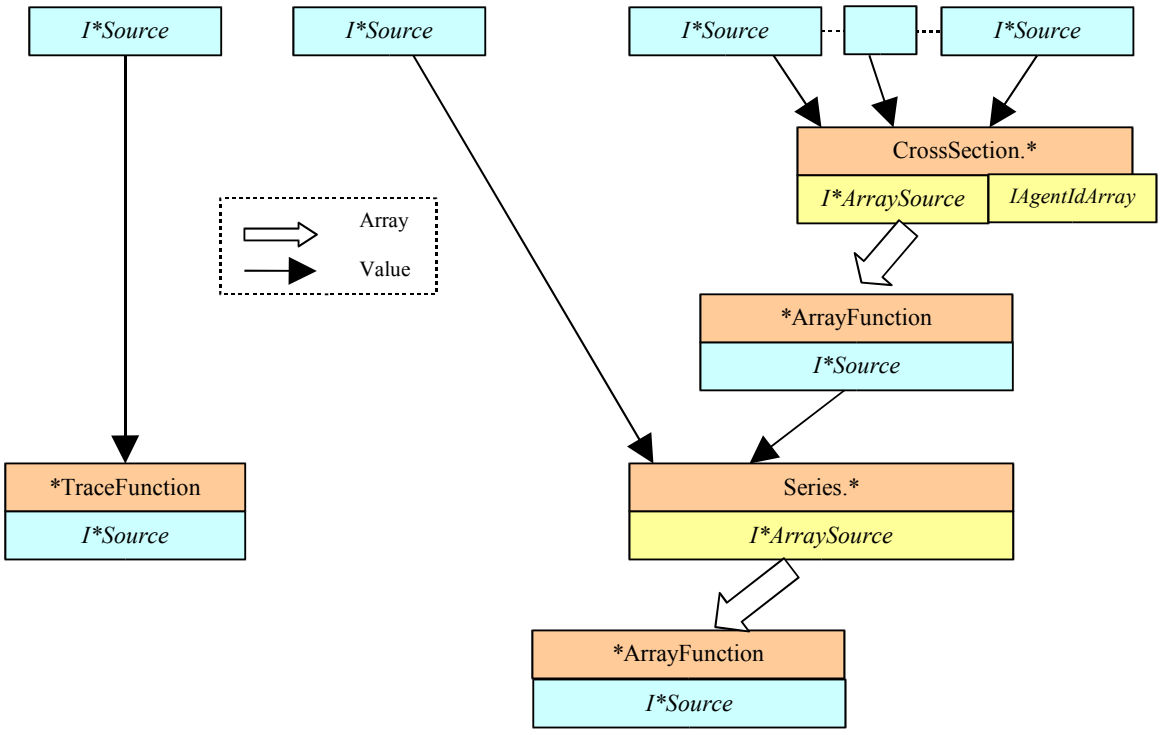
Thanks to the encapsulation system we can create a stack of operations and then obtain the value simply invoking one method. The figure below shows how to build the moving average computer.

Don't worry! The code to build this operation is simpler than its visual representation, as shown by the following instructions:

```
CrossSection.Integer crossSection =
   new CrossSection.Integer(agentList, "income", false);[3]
Series.Double series = new Series.Double(new MeanFunction(crossSection));
MovingAverageArrayFunction ma =
   new MovingAverageArrayFunction(series, 3 /*moving average window*/);
```

Every time the ma object receives the updateSource() command, the entire stack is automatically updated and the current moving average of the last three periods becomes available.

A comprehensive schema of the encapsulation system is shown in the following figure.



[3] If MyAgent implements the IIntSource interface the instruction becomes:

```
CrossSection.Integer crossSection =
    new CrossSection.Integer(agentList, MyAgent.AGE);
```

The statistic computers can return single values, via the I*Source interface, and array of values, via the I*ArraySource one. A data source can be processed by a *Function[4] object, which applies the function and return a value, via an I*Source interface.

Every time an object implements an I*Source interface it can be inserted in the encapsulation stack as a source of data used by the subsequent consumer in the stack. The encapsulation allows an infinite number of operations to be sequentially executed, with a single update operation.

It is very important to point out that each array consumer object must receive as source an array source data, while single value consumers work only with single values sources.

The functions contained by the jas.statistics.functions package are divided in two main groups:

The *ArrayFunction objects work with I*ArraySource sources which are refresh at every updateSource() call.

The *TraceFunction objects work with single value sources (I*Source). Obviously a single value cannot be used to create a statistics, so these functions trace the value over time. For instance, the MeanTraceFunction computes the average value, by storing the sum and the count of the values it receives over time.

[4] With the *Function notation we intend all the functions inheriting from the AbstractFunction, contained in the jas.statistics.functions package.

3. How statistics are updated

If user had to update all the elements in the encapsulation system, the system would be very complex to be managed. In the previous example, the reader should update the crossSection object, than the series and finally the ma one, to obtain the moving average.

Fortunately, JAS automatically updates the statistical widgets, using the IUpdatableSource interface. Each statistical computer which retrieves data from an I*Source source, checks if the source implements the IUpdatableSource interface and, if it does, updates it before reading data.

Through this method each object in the stack is recursively updated. This makes statistics very easy to be managed, but it may cause some problems when the same source is present in more than one stack. In this case it would be updated twice or more. Imagine a situation in which a time series is forced to be updated two times in the same simulation step, it would append twice the current data.

JAS avoids this possible drawback checking the simulation time before invoking the updateSource(), ignoring objects already updated. Obviously this choice does not permit to refresh data more than once per simulation step. In order to bypass this constraint, the user has to explicitly set to false the checkingTime property of the statistics computer.

Trying to summarize the updating mechanism, we can enumerate the following rules of thumb:

1. Each I*Source consumer has to check if the source implements the IUpdatableSource interface and, in positive case, invoke its updateSource() method before reading data.
2. When updated, each I*Source source has to check current simulation time and perform the update only if the time is different from the latest update time.
3. In order to force an I*Source object to bypass the time checking, its checkingTime property must be explicitly set to false (using the setCheckingTime(false);) instruction.

4. CrossSection object

The CrossSection object retrieves the variable's value from each agent contained in a Java collection. If objects implement the I*Source interface data are read directly, otherwise they are collected through a type specific reflector (*Invoker).

At every update the cross section its current data cache and creates dynamically a new array of values, with the same dimension of the source collection.

The CrossSection class provides four implementations to natively support the main Java data types. The available implementations are:

- CrossSection.Double, which implements the IDoubleArraySource interface;
- CrossSection.Float, which implements the IFloatArraySource interface;
- CrossSection.Integer, which implements the IIntArraySource interface;
- CrossSection.Long, which implements the ILongArraySource interface.

So, for instance, a cross section reading float values has to be created using the CrossSection.Float constructor. The four implementations support the specific I*ArraySource interface, in order to provide an array of the specific data type.

If the user wants to collect data only from agents with particular characteristics, she can adopt the ICollectionFilter interface. Passing to the cross section an object with the ICollectionFilter interface (via the setFilter() method), it collects only the values from the agents filtered by the custom filter.

If, for instance, we would like to compute the average income of the only "adult" agents in the agent list, we have to define a filter as follows:

```
public class Filter implements ICollectionFilter {
  public boolean isFiltered(Object object) {
    return ( ((MyAgent) object).age >= 18 );
  }
}
```

Passing an instance of the Filter class to the cross section, we will obtain an array representing the age of only the "adult" agents.

The CrossSection can be updated directly invoking the updateSource() method or, through the JAS ISimEventListener interface, by invoking the performAction (Sim.EVENT_UPDATE) method. See the documentation relative to the jas.engine package for more details about the ISimEventListener interface.

The CrossSection prevent repetitive updates during a simulation step, as described in the previous section. If the user wants to bypass the time checking to always force the update, she has to disable the feature using the following instructions:

```
CrossSection.Long cs = new CrossSection.Long(anAgent, "aLongVariable", false);
cs.setCheckingTime(false);
```

5. The Series object

The Series is a long time memory data collector. It requires a data I*Source source and append at each update the value to the list. If objects are I*Source it retrieves data directly, otherwise it uses a type specific reflector (*Invoker).

The Series class provides four implementations to support natively the main Java data types. The available implementations are:

- Series.Double, which implements the IDoubleArraySource interface;
- Series.Float, which implements the IFloatArraySource interface;
- Series.Integer, which implements the IIntArraySource interface;
- Series.Long, which implements the ILongArraySource interface.

It means that a series reading long values must be created using the Series.Long constructor. The four implementations support the specific I*ArraySource. It means that each cross section is able to return the data array of the specific data type.

The series in not yet a time series, because it does not store the time when data have been stored. In order to have a time series, the user has to append the series to the TimeSeries object, which can contain more than one series, synchronizing them with the time.

The Series can be updated directly invoking the updateSource() method, or using the JAS standard ISimEventListener interface, passing the Sim.EVENT_UPDATE constant. See the documentation relative to the jas.engine package for details about the ISimEventListener interface.

The Series prevent repetitive updates during a simulation step, as described previously. If the user wants to bypass the time checking to always force the update, she has to disable the feature using the following instructions:

```
Series.Long s = new Series.Long(anAgent, "aLongVariable", false);
s.setCheckingTime(false);
```

WARNING: Disabling the time checking allows a series to append more than one value per time unit. This may result inconsistent if used in TimeSeries object.

6. The statistic functions

All the statistic functions manage time checking using an instance of the TimeChecker class, which avoid the function to be applied more than one time per simulation step. If the user needs to bypass the time checking, she has to disable the feature.

The following table describes the aim of the *TraceFunction functions which operate on single source values over time:

Function	Description
MinTraceFunction	It verifies the source value over time keeping the lowest value ever received.
MaxTraceFunction	It verifies the source value over time keeping the highest value ever received.
MultiTraceFunction	It computes the minimum, the maximum, the sum, the mean and the variance by storing the sums and the count of the values received time by time.

The following table describes the aim of the *ArrayFunction function which operate on array of source values relative to the current simulation time:

Function	Description
MinArrayFunction	It finds the lowest value in the array.
MaxArrayFunction	It finds the highest value in the array.
MeanVarianceArrayFunction	It computes the average and the variance for the values in the array.