Optimization Experiment Functions

You can use the following functions to control the Optimization experiment, retrieve the data on its execution status and use it as a framework for creating custom experiment UI.

Controlling execution

Function	Description
`void run()`	Starts the experiment execution from the current state. If the model does not exist yet, the function resets the experiment, creates and starts the model.
`void pause()`	Pauses the experiment execution.
`void step()`	Performs one step of experiment execution. If the model does not exist yet, the function resets the experiment, creates and starts the model.
`void stop()`	Terminates the experiment execution.
`void close()`	This method returns immediately and performs the following actions in a separate thread: • Stops experiment if it is not stopped, • Destroys the model, • Closes the experiment window (only if the model is started in the application mode).
`Experiment.State getState()`	Returns the current state of the experiment: `IDLE`, `PAUSED`, `RUNNING`, `FINISHED`, `ERROR`, or `PLEASE_WAIT`.
`double getRunTimeSeconds()`	Returns the duration of the experiment execution in seconds, excluding pause times.
`int getRunCount()`	Returns the number of the current simulation run, i.e., the number of times the model was destroyed.
`double getProgress()`	Returns the progress of the experiment: a number between `0` and `1` corresponding to the currently completed part of the experiment (based on iteration count or time limit), or `-1` if the progress cannot be calculated.
`int getParallelEvaluatorsCount()`	Returns the number of parallel evaluators used in this experiment. On multicore / multiprocessor systems that allow parallel execution this number may be greater than 1.

Objective

Function	Description
`double getCurrentObjectiveValue()`	Returns the value of the objective function for the current solution.

`double getBestObjectiveValue()`	Returns the value of the objective function for the optimal currently found solution. Note that the solution may be infeasible. To check the solution feasibility, call the `isBestSolutionFeasible()` method.
`double getSelectedNthBestObjectiveValue()`	Returns the objective value for the Nth best solution identified by the method `selectNthBestSolution(int)`.

Solution

Function	Description
`boolean isBestSolutionFeasible()`	Returns `true` if the optimal solution satisfies all constraints and requirements; returns `false` otherwise.
`boolean isCurrentSolutionBest()`	Returns `true` if the solution is currently the optimal one; returns `false` otherwise.
`boolean isCurrentSolutionFeasible()`	Returns `true` if the current solution satisfies all constraints and requirements; returns `false` otherwise.
`boolean isSelectedNthBestSolutionFeasible()`	Returns `true` if the Nth best solution satisfies all constraints and requirements; returns `false` otherwise.
`void selectNthBestSolution (int bestSolutionIndex)`	This method locates the Nth best solution and sets up the data for subsequent method calls that retrieve specific pieces of information (for example, for the `getSelectedNthBestObjectiveValue()` and `getSelectedNthBestParamValue(COptQuestVariable)` methods). *Parameter:* `bestSolutionIndex` - the index of the optimal solution (passing `1` will locate the optimal solution, `2` - the second optimal, etc.)

Optimization parameters

Function	Description
`double getCurrentParamValue (COptQuestVariable optimizationParameterVariable)`	Returns the value of the given optimization parameter variable for the current solution.
`double getBestParamValue (COptQuestVariable optimizationParameterVariable)`	Returns the value of the given optimization parameter variable for the optimal currently found solution. Note that the solution may be infeasible. To check the solution feasibility, call the `isBestSolutionFeasible()` method.
`double getSelectedNthBestParamValue (COptQuestVariable optimizationParameterVariable)`	Returns the value of the variable for the Nth best solution identified by calling the `selectNthBestSolution(int)` method.

Iterations

Function	Description
`int getCurrentIteration()`	Returns the current value of iteration counter.
`int getBestIteration()`	Returns the iteration that resulted in the optimal currently found solution. Note that the solution may be infeasible. To check the solution feasibility, call the `isBestSolutionFeasible()` method.
`int getMaximumIterations()`	Returns the total number of iterations.
`int getNumberOfCompletedIterations()`	Returns the current value of iteration counter.
`int getSelectedNthBestIteration()`	Returns the iteration number for the Nth best solution identified by the `selectNthBestSolution(int)` method.

Replications

Before calling the optimization experiment methods you may need to ensure that replications are used (call the `isUseReplications()` method).

Function	Description
`boolean isUseReplications()`	Returns `true` if the experiment uses replications; returns `false` otherwise.
`int getCurrentReplication()`	Returns the replication number for the currently evaluated solution.
`int getBestReplicationsNumber()`	Returns the number of replications that were run to get the optimal solution. Note that the solution may be infeasible. To check the solution feasibility, call the `isBestSolutionFeasible()` method.
`int getSelectedNthBestReplicationsNumber()`	Returns the number of replications for the Nth best solution identified by the method selectNthBestSolution(int)

Accessing the model

Function	Description
`Engine getEngine()`	Returns the engine executing the model. To access the model's top-level agent (typically, `Main`), call `getEngine().getRoot();`
`Presentation getPresentation()`	Returns the presentation object of the model, or `null` if there is none.

Restoring the model state from snapshot

Function	Description
`void setLoadRootFromSnapshot( String snapshotFileName)`	Tells the simulation experiment to load the top-level agent from AnyLogic snapshot file. This method is only available in AnyLogic Professional.

	Parameter: `snapshotFileName` - the name of the AnyLogic snapshot file, for example: `"C:\\My Model.als"`
`boolean isLoadRootFromSnapshot()`	Returns `true` if the experiment is configured to start the simulation from the state loaded from the snapshot file; returns `false` otherwise.
`String getSnapshotFileName()`	Returns the name of the snapshot file, from which this experiment is configured to start the simulation.

Error handling

Function	Description
`RuntimeException error(Throwable cause, String errorText)`	Signals an error during the model run by throwing a `RuntimeException` with `errorText` preceded by the agent full name. This method never returns, it throws runtime exception by itself. The return type is defined for the cases when you would like to use the following form of call: `throw error("my message");` *Parameters:* `cause` - the cause (which will be saved for more detailed message), may be `null`. `errorText` - the text describing the error that will be displayed.
`RuntimeException errorInModel(Throwable cause, String errorText)`	Signals a model logic error during the model run by throwing a `ModelException` with specified error text preceded by the agent full name. This method never returns, it throws runtime exception by itself. The return type is defined for the cases when you would like to use the following form of call: `throw errorInModel("my message");` This method differs from `error()` in the way of displaying error message: model logic errors are 'softer' than other errors, they use to happen in the models and signal the modeler that model might need some parameters adjustments. Examples are '*agent was unable to leave flowchart block because subsequent block was busy*', '*insufficient capacity of pallet rack*' etc. *Parameters:* `cause` - the cause (which will be saved for more detailed message), may be `null`. `errorText` - the text describing the error that will be displayed.

Command-line arguments

Function	Description
`String[] getCommandLineArguments()`	Returns an array of Command-line arguments passed to this experiment on model start. Never returns `null`: if no arguments are passed, an empty array is returned. You can call this function in the experiment's **Additional class code**.