

The OptQuest Engine Java and .NET Developer's Guide

Table Of Contents

Introduction to optimization	1
What is optimization?	1
How the OptQuest Engine works	1
Using the documentation.....	2
Platforms.....	2
Defining an optimization.....	3
Overview.....	3
Defining decision variables.....	3
Defining user-controlled variables	5
Defining the objective	5
Defining linear constraints	6
Defining non-linear constraints (Requirements).....	8
String Expressions.....	9
Defining stop criteria.....	11
Running the optimization	12
When the optimization completes	13
Running additional iterations	14
Analyzing results	15
Solutions.....	15
Feasible and infeasible solutions	15
Constraints as goals	17
Simulation optimization	19
Overview.....	19
Advanced Features.....	19
Decision variables	19
User controlled variables	20
Constraints	20
Objective.....	21
Optimization Options	21
Running the optimization	21
Displaying results	22
General replication algorithm.....	23
Running additional replications	25
Ranking and selection algorithm	26
Portfolio optimization.....	29
Overview.....	29
Portfolio project decision variables.....	30

Portfolio project observations	31
Discount rate	31
Constraints	31
Budget and resource constraints.....	32
Quality measure constraint.....	32
Mutually exclusive projects.....	32
Delayed Start of Project.....	33
Project dependencies	33
Average Score Constraint	33
Balance Constraint	34
Portfolio requirement	34
Portfolio objective	34
Portfolio measures and statistics.....	35
Running the optimization	36
Parallel Implementation.....	39
Overview.....	39
Parallel Optimization Behavior	40
Parallel evaluation methods	41
Parallel Implementation with Replications.....	42
Stopping parallel evaluation	43
Neural Net	45
Overview.....	45
Creating nodes	46
Creating arcs	46
Training the neural net.....	47
Using a neural net	47
Advanced topics.....	49
Exception Handling.....	49
Efficient Frontier	49
Modifying an optimization problem.....	50
Additional optimization.....	53
Permutation groups	53
Saving and using solutions.....	53
Tuning OptQuest for a particular problem.....	54
Solution Search Algorithms	55
OptQuest Solution Generation	57
Overview.....	57
Discrete variable enumeration.....	57
Suggesting possible solutions	58

Modifying a solution.....	58
Constraints and solution generation.....	59
Precision properties.....	59
Rejecting Solutions.....	59
Version and licensing information.....	61
Version information.....	61
Licensing.....	61
Glossary.....	63
binary variable.....	63
linear constraint.....	63
constraint-feasible.....	63
continuous variable.....	63
decision variable.....	63
design variable.....	63
discrete variable.....	64
efficient frontier.....	64
infeasible.....	64
metaheuristics.....	64
objective function.....	64
optimization.....	64
permutation variable.....	64
non-linear constraints (requirement).....	64
scatter search.....	64
Tabu search.....	64
Glossary.....	67
Index.....	69

Introduction to optimization

What is optimization?

Optimization technology allows analysts to search for optimal solutions to complex business and engineering problems.

The OptQuest® Engine, OptTek Systems, Inc.'s optimization software, enables you to find the best answer to questions such as:

- What is the maximum return on budgets allocated to different uses, given uncertain product demand, machine reliability, and raw material availability?
- What is the most effective configuration of machines for production scheduling under variable conditions of demand and operation?
- What are the most effective location and release sequencing of raw materials to minimize waiting time?
- What are the optimal work force allocations to minimize lead time and labor costs?
- What are the optimal allocations for an investment portfolio?

Investigating every possible scenario can be extremely time-consuming and, in many cases, impossible. The OptQuest Engine replaces the inaccuracy of trial-and-error with a potent search engine that pinpoints the best decision. Users can quickly determine what combination of variables results in achieving their objectives.

How the OptQuest Engine works

The OptQuest Engine incorporates metaheuristics to guide its search algorithm toward better solutions. This approach remembers which solutions worked well and recombines them into new, better solutions. In addition, this technique does not get trapped in local solutions or get thrown off course by noisy (uncertain) model data.

The OptQuest Engine combines Tabu search, scatter search, integer programming, and neural networks into a single, composite search algorithm providing maximum efficiency in identifying new scenarios. OptTek's well-researched technology has been shown to be orders of magnitude faster than other

approaches. The OptQuest Engine is also flexible, intelligent, and easy to use. And, according to experts in the field, it is the most reliable optimization software on the market today.

Using the documentation

The intent of this document is to provide an overview of the features of the OptQuest Engine. Please refer to the The Java and .NET OptQuest Engine API documentation included with your version of the OptQuest Engine for detailed information on the classes and methods.

Platforms

The OptQuest Engine optimization software is available in the following formats

OptQuest.jar	This product is written entirely in Java.
OptQuestNET.dll	This product is written entirely in C#.
COMOptQuest.dll	This is a COM interface to the OptQuestsNET.dll and can be used by any Windows application that can interact with a COM object.

Please contact optinfo@OptTek.com for more information on platforms and languages.

Defining an optimization

Overview

The simplest optimization problem contains the following information:

- decision variables
- objective function
- the goal of the optimization (to either minimize or maximize the value of the objective function)

Optimization is an iterative process where:

- The OptQuest Engine calculates possible solutions for the decision variables
- The objective function and non-linear constraints are evaluated using the solutions suggested by the OptQuest Engine
- The results are analyzed by the OptQuest Engine and a new set of possible solutions is calculated

The OptQuest Engine

The COptQuestOptimization class allows you to define and run an optimization problem. In the simplest case, you:

- Add decision variable objects to the COptQuestOptimization object
- Add an objective function object to the COptQuestOptimization object and set the goal of the optimization (maximize or minimize)
- Extend the COptQuestOptimization class and override the COptQuestOptimization::Evaluate() method. In the Evaluate() method, you provide the code to calculate the objective value using the solution provided by OptQuest. (Note that this may not be necessary for portfolio optimization or if your objective is a mathematical expression on the decision variables.)
- Define how long the optimization should run by setting the stop properties of the COptQuestOptimization class

For more complex optimizations, you may:

- Define linear constraints by adding constraint objects
- Define non-linear constraints by adding requirement objects
- Display the progress of the optimization by overriding the COptQuestOptimization::MonitorStatus() method to provide feedback on the progress of the optimization.
- Provide solution analysis after the optimization completes.

Defining decision variables

A decision variable is a quantity that the decision-maker controls. For example, the number of nurses to employ during the morning shift in an emergency room may be a decision variable in an optimization model for labor scheduling. The OptQuest Engine manipulates decision variables in search of values that produce the optimal value for the objective function.

Decision variables are added to an optimization problem by instantiating a decision variable and adding the instance to the optimization using the `COptQuestOptimization::AddVariable()` method.

You can optionally assign a name to a decision variable and the variable names can be used in string expressions for defining a `COptQuestStringObjective` or a `COptQuestStringConstraint`. If the variable name contains special characters such as blanks, enclose the variable name in square brackets or curly braces when creating the string expression. For example, "2500*[Machine 1] + 3000 * [Machine 2] <= 100000".

Defining the range of valid values

You define the range of values for a decision variable by setting the lower bound and upper bound of the variable. It is important to pick meaningful bounds for your decision variables so the OptQuest Engine doesn't waste time exploring solutions that are not meaningful.

You can further limit the set of valid values by defining a sub-range of values that are excluded as possible solutions for a decision variable. The `COptQuestVariable::SetExclusiveRange()` method can be used to define a sub-range of values that will be excluded as possible solutions. For example, if your decision variable can have a value between 0 and 1 but not between 0 and .25, you would set the low bound to 0, the high bound to 1 and an exclusive range of 0 and .25. Any value > 0 and < .25 would be excluded.

Multiple exclusive ranges can be defined on one decision variable.

The OptQuest Engine supports the following type of decision variables.

<code>COptQuestContinuousVariable</code>	A continuous variable may take on any value between the lower bound and upper bound.
<code>COptQuestDiscreteVariable</code>	A discrete variable may take on values between the lower bound and upper bound and increments of the step size.
<code>COptQuestIntegerVariable</code>	A discrete variable with integer bounds and a step size of 1
<code>COptQuestBinaryVariable</code>	A discrete variable with a value of 0 or 1
<code>COptQuestProjectVariable</code>	Defines a project in a portfolio optimization. See Portfolio optimization
<code>COptQuestDesignVariable</code>	Used for decisions where the value of the variable represents an alternative, and not a quantity. Design variables are useful in optimization problems where the decision variables consist of choosing the best alternative from a catalog, and a larger number may not imply the commitment of more resources. For example, the design variable may represent the choices red, green, blue or orange. Design variables are defined by a lower bound, an upper bound, and a step size that controls the number of choices available within the specified range.
<code>COptQuestPermutationVariable</code>	Used to solve sequencing problems. For example, you could use this variable to determine the order in which paint should be mixed to minimize cleanup time between color changes. The value of a permutation variable represents the order within the sequence.
<code>COptQuestEnumerationVariable</code>	The possible values are limited to an enumerated list of

values such as 7,10 and 25.

Defining user-controlled variables

User-controlled variables are variables that are involved in an optimization problem, but their values are set by you rather than the OptQuest Engine. The value for a user-controlled variable may be the result of a calculation that involves one or more decision variables or it may be the output of a simulation. User-controlled variables can be used to define the objective function or non-linear constraints (requirements.)

User-controlled variables are added to an optimization problem by instantiating a `COptQuestUserControlledVariable` and adding the instance to the optimization with the `COptQuestOptimization::AddVariable()` method.

The values of user-controlled variables should be set by you in the `COptQuestOptimization::Evaluate()` method. Use the `COptQuestSolution.SetVariableValue()` method to assign a value to the solution given to you by the `Evaluate()` method.

You can optionally assign a name to a user-controlled variable and the variable names can be used in string expressions for defining a `COptQuestStringObjective` or a `COptQuestStringConstraint`. If the variable name contains special characters such as blanks, enclose the variable name in square brackets or curly braces when creating the string expression. For example, "[Risk Factor] <= .10;"

Defining the objective

Optimization is the process of searching for the values of the decision variables that provide a maximum or a minimum value of the objective.

The objective is added to an optimization problem by instantiating an objective object and adding the instance to the optimization using the `COptQuestOptimization::AddObjective()` method.

<code>COptQuestUserControlledObjective</code>	Use a <code>COptQuestUserControlledObjective</code> when you will be calculating the value of the objective using a custom calculation or running a simulation where one of the outputs of the simulation is the objective. You will need to override the <code>COptQuestOptimization::Evaluate(COptQuestSolution solution)</code> method, calculate the objective value and set the value in the solution being evaluated. Use the <code>COptQuestSolution::SetObjectiveValue()</code> method to set the value of the objective.
<code>COptQuestStringObjective</code>	Used when the objective can be expressed as a mathematical expression using decision variables or user-controlled variables. If user-controlled variables are used in the expression, you will need to override the <code>COptQuestOptimization::Evaluate(COptQuestSolution solution)</code> method and set the values of the user-controlled variables using the <code>COptQuestSolution.SetVariableValue()</code> method. The

	<p>OptQuest Engine would calculate the objective value using the values you provided.</p> <p>If the mathematical expression does not use user-controlled variables, the OptQuest Engine can perform the calculation without any additional information from you. In this case it is not necessary to override the Evaluate() method to calculate the objective.</p>
COptQuestObjectiveFunction	<p>Used when the objective is a linear expression in the form :</p> $\text{coeff1} * \text{var1} + \text{coeff2} * \text{var2} + \dots \text{coeffn} * \text{varN}$ <p>where var1 to varN are decision variables.</p> <p>If the objective function and all constraints are linear, the OptQuest Engine uses a MILP linear solver rather than the iterative, heuristic search.</p>
COptQuestPortfolioObjective	<p>Used when the optimization is a portfolio optimization and the objective is minimizing or maximizing a statistic on a measure. To use a portfolio objective, the decision variables must be COptQuestProjectVariable with observations or period observations defined.</p> <p>For portfolio optimizations, it is not necessary to override the Evaluate() method to calculate the objective.</p> <p>See Portfolio optimization.</p>
COptQuestWeightedMultiObjective	<p>Combines sub-objectives to create a single multi-objective where each sub-objective is assigned a weight. For example, you could use a COptQuestWeightedMultiObjective to define an objective that maximizes the mean of the NPV and minimizes the standard of deviation of the return. Each sub-objective has a direction (minimize or maximize) and a coefficient/weight that is applied to the objective.</p>
COptQuestFrontierMultiObjective	<p>The COptQuestFrontierMultiObjective multi-objective finds a set of solutions that are on the frontier. When the optimization completes, you will get an array of solutions. See Multi-Objective Optimization</p>

Defining linear constraints

Linear constraints describe a linear relationship among decision variables. A linear constraint is a mathematical expression where linear terms (i.e., a coefficient multiplied by a decision variable) are added or subtracted and the resulting expression is forced to be greater-than-or-equal, less-than-or-equal, or exactly equal to a right-hand side value.

The following are examples of linear constraints on the decision variables:

$$\text{Var1} + \text{Var2} + \text{Var3} + \text{Var4} + \text{Var5} = 10500$$

$$0 \leq \text{Var1} + 2 * \text{Var2} - \text{Var3} \leq 5000$$

$$\text{Var1} - 3 * \text{Var5} \geq 300$$

$$\text{Var1} \geq 6 \text{ or } \text{Var2} \geq 6 \text{ or } \text{Var1} + \text{Var2} = 4$$

The following is an example of a linear constraint with variable bounds (See Efficient Frontier). The left hand side will be evaluated against the value 1000, then 5000 and finally 20000.

$$\text{Var1} + \text{Var2} < (1000, 5000, 20000)$$

A linear constraint is defined by the COptQuestLEConstraint, COptQuestEQConstraint or COptQuestGEConstraint class. Linear constraints can be added to an optimization problem by instantiating a constraint and adding the instance to the optimization using the COptQuestOptimization::AddConstraint() method.

If the constraint object is an instance of COptQuestLEConstraint, COptQuestEQConstraint or COptQuestGEConstraint, you define the constraint by defining the constant value of the right-hand side and the list of variables and their coefficients. The right-hand side value is set by calling the SetRHS() method. The decision variables and their coefficients are defined by calling the AddVariable() method of the constraint object for each decision variable.

In the example $\text{Var1} - 3 * \text{Var5} \geq 300$, you would instantiate a COptQuestGEConstraint. You would set the right-hand side by calling SetRHS(300). You would call AddVariable(Var1,1) and AddVariable(Var5,-3) to add the variables and their coefficients. You would add the constraint to the optimization by calling COptQuestOptimization::AddConstraint().

You can combine constraints to create a logical "or" expression by instantiating a COptQuestOrConstraint object and adding the constraints in the "or" expression using the COptQuestOrConstraint::AddConstraint() method. The constraint is satisfied if at least one of the constraints is satisfied.

You can also define a linear constraint using the COptQuestStringConstraint class which allows you to define the constraint as a mathematical string expression. The examples above are all valid string expressions defining a linear constraint. In the example $\text{Var1} - 3 * \text{Var5} \geq 300$, you would instantiate a COptQuestStringConstraint and call SetEquation("Var1 - 3*Var5 >= 300"). You can create a logical "or" expression by using the "or" keyword in the expression of a COptQuestStringConstraint. See String Expressions.

The OptQuest Engine supports the following linear constraint classes:

COptQuestEQConstraint	Used to define an equality constraint such as $\text{Var1} + 3 * \text{Var2} = 150$
COptQuestLEConstraint	Used to define a less than or equal constraint such as $\text{Var1} + 3 * \text{Var2} \leq 150$
COptQuestGEConstraint	Used to define a greater than or equal constraint such as $\text{Var1} + 3 * \text{Var2} \geq 150$
COptQuestOrConstraint	Used to combine constraints to create a logical "or" operation among the constraints. For example, $\text{Var1} \geq 150$ or $\text{Var1} - \text{Var2} \leq 0$ or $3 * \text{Var3} \leq 100$ would be defined by adding a COptQuestGEConstraint, a COptQuestLEConstraint and another COptQuestLEConstraint to the COptQuestOrConstraint.
COptQuestStringConstraint	Used to define a constraint as a string expression.

Defining non-linear constraints (Requirements)

A non-linear constraint is a mathematical expression describing a non-linear relationship among decision variables and/or user-controlled variables. A non-linear constraint can also be a restriction on a response that requires its value to fall within a specified range. For example, a requirement for a simulation problem may to keep risk below a specified value where risk is calculated by the simulation.

The following are examples of non-linear constraints (requirements):

$$\text{Var1} * \text{Var2} + \text{Var3} + \text{Var4} + \text{Var5} \leq 10500$$

$$0 \leq \text{Var1} + 2 * \text{Var2} * \text{Var3} \leq 5000$$

$$\text{Var1} - 3 * \text{Var4} * \text{Var5} \geq 300$$

$$\text{Var1} * \text{Var2} \geq 6 \text{ or } \text{Var2} * \text{Var3} \geq 6 \text{ or } \text{Var3} * \text{Var4} \geq 6$$

$$\text{Var1} * \text{Var2} \geq 6 \text{ or } \text{Var2} + \text{Var3} \geq 6$$

The following is an example of a non-linear constraint with variable bounds (See Efficient Frontier). The left hand side will be evaluated against the value 1000, then 5000 and finally 20000.

$$\text{Var1} * \text{Var2} < (1000, 5000, 20000)$$

A requirement object is defined by the `COptQuestLowerRequirement`, `COptQuestUpperRequirement` or `COptQuestDualRequirement` class. When you use a requirement object in an optimization problem, you will need to set the value of the requirement object in the `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method. Set the requirement value using the `COptQuestSolution::SetRequirementValue()` method.

The `COptQuestLowerRequirement` is used if the value of the requirement must be greater than or equal to a specified lower bound. A `COptQuestUpperRequirement` is used if the value of the requirement must be less than or equal to a specified upper bound. If the requirement value is bounded by both a lower bound and an upper bound, use a `COptQuestDualRequirement` object. Non-linear constraints can be added to an optimization problem by instantiating a requirement and adding the instance to the optimization using the `COptQuestOptimization::AddRequirement()` method.

The OptQuest Engine checks the value of the requirement against the requirement's bound(s). If the value satisfies the requirement, the requirement is feasible for that solution. If the value violates the requirement, the requirement is infeasible. If a solution violates any requirement, the solution is considered infeasible. The OptQuest Engine uses the feasibility of the requirement in its search to find optimal, feasible solutions. See Feasible and infeasible solutions.

You can combine requirements to create a logical "or" expression by instantiating a `COptQuestOrRequirement` object and adding the requirements in the "or" expression using the `COptQuestOrRequirement::AddRequirement()` method.

A `COptQuestStringConstraint` may also define a non-linear constraint if the mathematical expression of the string constraint is a non-linear expression of decision variables or contains user-controlled variables.

The examples above are all valid string expressions defining a non-linear constraint. To use a string constraint, instantiate a `COptQuestStringConstraint` object and set the string expression using the `COptQuestStringConstraint::SetEquation()` method. Use the `COptQuestOptimization::AddConstraint()` method to add the constraint to the optimization. You can also create a logical "or" expression by using the "or" keyword in the expression of a `COptQuestStringConstraint`. See String expressions.

If the expression includes user-controlled variables, you set the values of the user-controlled variables in the `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method. The OptQuest Engine will calculate the value of the expression using the values you supplied and determine the feasibility of the non-linear requirement.

If you are doing portfolio optimization, the requirement object can be an instance of a `COptQuestPortfolioLowerRequirement` or an instance of a `COptQuestPortfolioUpperRequirement` object. The portfolio requirements are calculated as a statistic on a measure. To use a portfolio requirement, the decision variables must be `COptQuestProjectVariables` with observations or period observations. See Portfolio requirement.

Designing the value of the requirement

It is important that the value of a requirement provide a measure that can be used by the OptQuest Engine to determine a distance from feasibility. **A requirement value that indicates a pass/fail condition will not help the OptQuest Engine in its search for solutions since every solution that fails is considered equally bad.** It is better to assign a range of values to the requirement where a solution that is closer to the bound of the requirement can be treated as superior to a solution whose value is farther away from the bound.

The OptQuest Engine supports the following non-linear constraint classes.

<code>COptQuestLowerRequirement</code>	Used when the result needs to be greater than or equal to the lower bound. For example, $Output1 \geq 150$
<code>COptQuestUpperRequirement</code>	Used when the result needs to be less than or equal to an upper bound. For example, $Output1 \leq 150$
<code>COptQuestDualRequirement</code>	Used when the result needs to be between two values. For example, $50 \leq Output1 \leq 150$
<code>COptQuestOrRequirement</code>	Used to combine requirements to create a logical "or" operation among the requirements. For example, $Output1 \geq 150$ or $Var1 \leq Var2$ or $Output2 * Var3 \leq 100$
<code>COptQuestPortfolioLowerRequirement</code>	Used for portfolio optimization. The requirement identifies a measure and a statistic on the measure. The result must be greater than or equal to the lower bound of the requirement. To use a portfolio requirement, the decision variables must be <code>COptQuestProjectVariables</code> with observations or period observations.
<code>COptQuestPortfolioUpperRequirement</code>	Used for portfolio optimization. The requirement identifies a measure and a statistic on the measure. The result must be less than or equal to the upper bound of the requirement. To use a portfolio requirement, the decision variables must be <code>COptQuestProjectVariables</code> with observations or period observations.
<code>COptQuestStringConstraint</code>	Used to define a non-linear constraint as a mathematical expression using decision variables and (optionally) user-controlled variables.

String Expressions

The `COptQuestStringConstraint` and `COptQuestStringObjective` classes allow you to define a constraint or an objective using a string to define a mathematical expression. To use this capability, you must provide unique names for decision variable and user-controlled variables that are added to the optimization using the `COptQuestOptimization::AddVariable()` method. When the mathematical expression is parsed, the variable names are used to locate the decision variable or user-controlled variable and retrieve the value used to evaluate the expression.

The following are examples of valid string expressions defining an objective function. An objective function expression cannot use the operators =, <=, >=.

$$250 * \exp(\text{Var1} + 0.67 * \text{Var2})$$

The following are examples of valid string expressions defining a linear constraint. Constraint expressions must use at least one of the operators =, <=, >=. Constraint expressions can also use the "or" keyword to define a logical or constraint.

$$\text{Var1} + 3 * \text{Var2} = 200$$

$$500 \leq \text{Var1} - \text{Var2} + 3 * \text{Var3} \leq 600$$

$$\text{Var1} + \text{Var2}/10 \geq 5$$

The following are examples of valid string expressions defining a non-linear constraint:

$$\text{Var1} * 3 * \text{Var2} \geq 600$$

An equation string may use the mathematical operators of +, -, *, and / and the functions described in the table below. You can use parentheses to enforce the order of evaluation.

The variable names in the constraint equation must match the names of variable objects added to the COptQuestOptimization object. The constraint expression is not evaluated until optimization begins, so a constraint string can reference a variable that has not yet been added to the problem. Any characters enclosed in square brackets or curly braces are treated as a variable name. For example, [Machine 1] is treated as a variable name and the OptQuest Engine will locate a variable with the name "Machine 1".

A COptQuestStringConstraint defines a linear constraint if the mathematical expression of the string constraint is a linear expression of decision variables. A COptQuestStringConstraint defines a non-linear constraint if the mathematical expression of the string constraint is a non-linear expression of decision variables or contains user-controlled variables. To use a string constraint to define a non-linear constraint, instantiate a COptQuestStringConstraint object and set the string expression using the COptQuestStringConstraint::SetEquation() method. Use the COptQuestOptimization::AddConstraint() method to add the non-linear constraint to the optimization.

Checking string expressions

The OptQuest Engine will validate all string expressions when the COptQuestOptimization::Optimize() method is called. If any errors are found a COptQuestException will be thrown. All named decision variables and named user-controlled variables that are used in a COptQuestStringConstraint or COptQuestStringObjective must be defined before the string expressions can be validated.

You can check string expressions for syntax errors before starting the optimization by calling the COptQuestStringConstraint::CheckExpression() or COptQuestStringObjective::CheckExpression(). To use the CheckExpression() method, you must add the decision variables and the COptQuestStringConstraint or COptQuestStringObjective to an instance of the COptQuestOptimization class. If the expression contains errors, a COptQuestException is thrown. The exception will give detailed information on the error.

The OptQuest Engine also allows you to check a constraint string expression or an objective string expression for linearity. COptQuestStringConstraint objects and COptQuestStringObjective objects are always checked for linearity when the COptQuestOptimization::Optimize() method is called. The IsLinear() method provides a way to check expressions and report linearity before the optimization begins.

You can use the following functions when defining string expressions.

Function	Syntax	Description
min	min(x,y)	Returns the smaller of two numbers.
max	max(x,y)	Returns the larger of two numbers.
sqrt	sqrt(x)	Returns the square root of a number.
log	log(x)	Returns the logarithm of a specified number.
log10	log10(x)	Returns the base 10 logarithm of a specified number.
pow	pow(x,y)	Returns a specified number raised to the specified power.
exp	exp(x)	Returns e raised to the specified power
abs	abs(x)	Returns the absolute value of a specified number.
Mathematical constant pi	pi	Mathematical constant pi, approximately equal to 3.14159.
Mathematical constant e	e	Mathematical constant e, approximately equal to 2.718.
rand	rand()	Returns a random number between 0 and 1, inclusive.
fmod	fmod(x,y)	Returns the remainder of x / y.
floor	floor(x)	Returns the largest whole number less than or equal to the specified number.
ceil	ceil(x)	Returns the smallest whole number greater than or equal to the specified number.
sin	sin(x)	Returns the sine of x, where x is an angle in radians.
cos	cos(x)	Returns the cosine of x, where x is an angle in radians.
tan	tan(x)	Returns the tangent of x, where x is an angle in radians.
sinh	sinh(x)	Returns the hyperbolic sine of x, where x is an angle in radians.
cosh	cosh(x)	Returns the hyperbolic cosine of x, where x is an angle in radians.
tanh	tanh(x)	Returns the hyperbolic tangent of x, an angle in radians.
asin	asin(x)	Returns the arcsine of x in the range $-\pi/2$ to $\pi/2$ radians. x is between -1 and 1.
acos	acos(x)	Returns the arccosine of x in the range 0 to π radians. x is between -1 and 1.
atan	atan(x)	Returns the arctangent of x in the range of $-\pi/2$ to $\pi/2$ radians. If x is 0, atan returns 0.
atan2	atan2(x,y)	Returns the arctangent of y/x in the range $-\pi$ to π radians. If both parameters of atan2 are 0, the function returns 0.
DtoR	DtoR(x)	Converts degrees to radians.
RtoD	RtoD(x)	Converts radians to degrees.

Defining stop criteria

You control how long an optimization will run by setting stop criteria. If more than one stop criteria are specified, the optimization stops when any one of the criteria is satisfied. At least one stop option must be specified but you can always stop an optimization before the stop criteria is satisfied by calling the `COptQuestOptimization::StopOptimization()` method.

Iteration count	You specify the maximum number of iterations to run. The optimization stops when the number of iterations exceeds this value. Use the <code>COptQuestOptimization::SetMaximumIterations()</code> method to set the maximum number of iterations to be run.
Seconds to run	You specify the maximum time to run in seconds. Optimization stops when the time exceeds this value. Use the <code>COptQuestOptimization::SetMaximumTime()</code> method to set the maximum number of seconds the optimization should run. When the specified time is reached, the current iteration is completed before the optimization stops. If the evaluation performed in the <code>Evaluate()</code> method is a lengthy operation, (such as running a simulation) the actual time before the optimization stops may be significantly longer.
Automatic Stop	When automatic stop is set, the optimization stops when OptQuest has determined that new solutions are not likely to produce a better objective value. Use the <code>COptQuestOptimization::SetAutoStop()</code> method to turn on automatic stop.
User-controlled Stop	In this case, the optimization runs until you tell it to stop by calling the <code>COptQuestOptimization::StopOptimization()</code> method. Use the <code>COptQuestOptimization::SetUserControlledStop()</code> to tell the OptQuest Engine you will stop the optimization.

Running the optimization

The optimization is run by calling the `COptQuestOptimization::Optimize()` method.

The `Optimize()` method performs the following functions:

- Checks all string expressions, making sure the variable names in the expression refer to a named variable that was added to the problem and the expression is syntactically correct. If any errors are detected, a `COptQuestException` is thrown.
- Checks the set of linear constraints for constraint feasibility. If no feasible solution exists, a `COptQuestException` is thrown
- The OptQuest Engine checks that at least one stop option has been specified.
- Initializes the solutions database
- If your optimization problem contains a custom calculation for the objective function, requirements or contains user-controlled variables, you would override the `Evaluate()` method and set the values in the `COptQuestSolution` object. The OptQuest Engine will call the `Evaluate()` method for each solution it generates.

Linearity and String Expressions

The OptQuest Engine will analyze a problem and pick the best method for solving the problem. If a problem has linear constraints and a linear objective, OptQuest will use linear programming to generate the first solution. This solution will have the best objective value. OptQuest will continue to generate

solutions using metaheuristics. Although the first solution will give the best objective value, other solutions are available for solution analysis.

When the optimization completes

If the optimization terminates because of an error, a COptQuestException will be thrown. You can use COptQuestException methods to determine the error condition. Be sure to put a try/catch block around your call to the COptQuestOptimization::Optimize() method. Any error in the problem setup will throw a COptQuestException.

The method COptQuestOptimization::GetTerminationReason() will return an integer value representing the termination reason. The method COptQuestOptimization::GetTerminationReasonString() will return a text string describing the reason the optimization terminated.

To retrieve the best solution, call the COptQuestOptimization::GetBestSolution() method. You can use COptQuestSolution methods to retrieve information about the best solution such as the value of each decision variable and the objective. Use the COptQuestSolution::IsFeasible() method to check the best solution for feasibility. A feasible solution satisfies all linear and non-linear constraints.

The following table describes the reasons an optimization terminates.

TERM_NOTSTARTED = 0	"The optimization has not been started." Returned if you call GetTerminationReason() before calling Optimize();
TERM_RUNNING = 1	"The optimization is still running." Returned if you call GetTerminationReason() while the optimization is in progress.
TERM_LP = 3	"The optimization was solved using a Linear/Integer/Mixed Integer Program." Returned when the optimization problem has been identified as a linear problem and the linear solver was used to solve the problem.
TERM_AUTOSTOP = 4	"The optimization stopped due to the Auto Stop feature." Returned when the stop criteria specify using the Auto Stop feature and the optimization stopped because the Auto Stop criteria was satisfied.
TERM_OPTIMALFOUND = 5	"The optimal solution was found." Returned when OptQuest was able to enumerate and evaluate all possible solutions.
TERM_MAXITERATIONS = 6	"The optimization stopped when the maximum number of iterations was reached." Returned when the stop criteria specify a maximum number of iterations and the optimization stopped because the number of iterations completed.
TERM_MAXTIME = 7	"The optimization stopped when the maximum time was reached." Returned when the stop criteria specify a maximum number of seconds and the optimization stopped because the time criterion was met.
TERM_USERSTOPPED = 8	"The optimization was stopped by the user." Returned when you call COptQuestOptimization::StopOptimization() to end the optimization.
TERM_BEST_FOUND = 9	"Rank And Selection completed." Returned when the user has called COptQuestOptimization::RankAndSelect() to refine a set of solutions using rank and selection algorithms and the rank and selection algorithm has completed.
TERM_EXCEPTION = 10	"The optimization stopped due to an exception." The OptQuest Engine detected an error and threw a COptQuestException. The COptQuestException::Description() method will return a text string describing the error condition.
TERM_INFEASIBLE = 12	"There are no solutions that satisfy the constraints." Returned when there are no possible solutions because of the problem's constraints.

TERM_CANNOTGENERATE = 13	"New (different) solutions cannot be generated." This termination reason will only be seen if you have called <code>COptQuestOptimization.SetCannotGenerateAsTermReason(true)</code> . The default OptQuest behavior is to throw an <code>EXCEPT_CANNOTGENERATE</code> exception.
-----------------------------	---

Running additional iterations

The OptQuest Engine supports running additional iterations after the initial optimization has completed. The OptQuest Engine will suggest a new solution for each additional iteration continuing the search begun in the initial optimization.

To run additional iterations, call the `COptQuestOptimization::SetAdditionalIterations()` method specifying the number of additional iterations to be run. Call the `COptQuestOptimization::OptimizeAdditional()` method to continue the optimization for the additional iterations. The `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method will be called for each additional iteration.

You can also specify an additional number of seconds to run by calling the `COptQuestOptimization::SetAdditionalTime()` and specifying the number of seconds to run. Call the `COptQuestOptimization::OptimizeAdditional()` method to continue the optimization for the additional time. The `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method will be called for each additional iteration.

This process can be repeated any number of time.

Pause and continue

This `COptQuestOptimization.ContinueOptimize()` allows you to continue an optimization that was stopped before all iterations were completed or while there was time remaining. The method will resume the optimization where it left off.

If you set additional iterations after stopping the optimization, the additional iterations are added to the remaining iterations. If you set additional time, the additional time is added to the remaining time.

Analyzing results

Solutions

The COptQuestSolution object contains the information about a particular solution. When the OptQuest Engine creates a solution, it sets the values for the decision variables. When a solution is evaluated, the objective value, user controlled variable values and requirement values are set.

If you are calculating the objective value or setting the values of user controlled variables, you will need to override the COptQuestOptimization::Evaluate(COptQuestSolution solution) method. Use the COptQuestSolution methods to retrieve the values for decision variables and set the values of the objective function, user controlled variables and requirements.

Before a solution is evaluated, it contains:

- The value for each decision variable
- The iteration that created the solution

To retrieve information after a solution has completed evaluation, override the COptQuestOptimization::MonitorStatus(COptQuestSolution currentSolution) method. The currentSolution parameter is the solution that just completed evaluation. If you are tracking the progress of the optimization, MonitorStatus() is the place you should retrieve the best solution found thus far. After a solution has completed evaluation, it will contain:

- The value for each decision variable
- The iteration that created the solution
- The feasibility of the solution
- The value of the objective
- The value of user controlled variables and requirements
- Replication information if replication processing is being used.

Feasible and infeasible solutions

A feasible solution is one that satisfies all linear and non-linear constraints. Each time the OptQuest Engine generates a new set of values for the decision variables it creates feasible solutions for linear constraints.

If a linear constraint is defined using only decision variables, the OptQuest Engine can determine feasibility when it generates a solution, because it has all the information it needs to calculate a value and ensure its feasibility. For example, if the constraint specifies the sum of the decision variables must be ≤ 100 , the OptQuest Engine can evaluate the constraint when it picks a solution. In this case, the OptQuest Engine will never return an infeasible solution. If the OptQuest Engine cannot find any solution that will satisfy this type of constraint, the problem is constraint infeasible and the OptQuest Engine throws a

`COptQuestException.EXCEPT_CONSTRAINTINFEASIBLE` when the `COptQuestOptimization::Optimize()` method is called.

If a constraint contains user-controlled variables or is a non-linear expression of decision variables, the value of the constraint is calculated after the OptQuest Engine has generated a possible solution. If the constraint has user-controlled variables, the caller must calculate the value of the variable, and then communicate the result of the calculation to the OptQuest Engine. For example, if the constraint is $\text{Var1} * \text{Result1} \geq 500$, where `Result1` is a user-controlled variable, the caller must calculate the value of `Result1` and tell the OptQuest Engine the value. The OptQuest Engine can then calculate the value of the constraint and determine the feasibility. If the constraint is a non-linear string equation of only decision variables, the OptQuest Engine does not need any additional information to calculate the value of the constraint.

The OptQuest Engine notes whether a solution is feasible or infeasible, and a penalty is assigned to infeasible solutions. A feasible solution is always superior to an infeasible solution.

The objective function and requirements are evaluated at the end of each iteration of the optimization. The OptQuest Engine uses the results of the non-linear constraints (requirements) to determine if a solution is requirement-feasible. If the result of a requirement is within the bounds of the requirement, the result is requirement-feasible. If the result is outside the bounds of the requirement, the solution is requirement-infeasible.

The OptQuest Engine makes finding a feasible solution its highest priority. Once it has found a feasible solution, it concentrates on finding better solutions.

The fact that a particular solution may be infeasible does not imply that the problem itself is infeasible. However, infeasible problems do exist. For example, suppose that in a Job Shop problem a foreman insists on finding an optimal configuration with the following constraints:

$$\text{drills} + \text{grinders} \leq 4$$

$$\text{drills} + \text{grinders} \geq 5$$

Clearly, there is no combination that will satisfy both of these constraints.

You can make infeasible problems feasible by fixing the inconsistencies of the relationships modeled by the constraints. The OptQuest Engine detects optimization models that are linear-constraint-infeasible and throws a `COptQuestException`. If a model is linear-constraint-feasible, the OptQuest Engine will always find a feasible solution and search for the optimal solution (i.e., the best solution that satisfies all linear constraints).

The OptQuest Engine cannot detect requirement-infeasible solutions. An example of a requirement-infeasible problem is:

$$\text{drills} * \text{grinders} \leq 19$$

$$\text{drills} * \text{grinders} \geq 20$$

If the OptQuest Engine cannot find any feasible solutions, you should check the requirements of the problem for feasibility.

Non-linear constraint and infeasible solutions

The OptQuest Engine may have trouble finding feasible solutions if the bounds on your requirement are tight. For example, if a risk calculation needs to be between .09 and .10, the OptQuest Engine may have trouble finding feasible solutions. If OptQuest does not find any feasible solutions, consider loosening the restrictions on your non-linear constraints.

It is important that the value of a requirement provide a measure that can be used by the OptQuest Engine to determine a distance from feasibility. **A requirement value that indicates a pass/fail condition will not help the OptQuest Engine in its search for solutions since every solution that fails is equally bad.** It is better to assign a range of values to the requirement where a solution that is closer to the bound of the requirement can be treated as superior to a solution whose value is farther away from the bound.

Constraints as goals

In some cases an optimization problem may have goals for a constraint bound. For example, a project manager may have a target value for a performance factor but won't mind if the target is exceeded by 3% or less. In this case the project manager would not want solutions that violate the constraint marked as infeasible but he would want solutions that are close to the constraint bound to be considered better than solutions that are far away from the bound.

The OptQuest Engine allows you to mark `COptQuestStringConstraint` objects and `COptQuestRequirement` objects as goals using the `SetGoal()` method for each object. Constraints and requirements that are goals will never be marked as infeasible.

Goal constraints can be paired with hard constraints to set a limit on the target. For example, you may want to target risk to be 5% or less but not accept risk over 8%. You could define a constraint goal of "Risk <= .05" and a hard constraint of "Risk <= .08". In this case, OptQuest would search for solutions that resulted in a risk less than .05, however, solutions with a risk <= .08 would be considered feasible. Any solution that created a risk > .08 would be marked infeasible.

Simulation optimization

Overview

In order to add optimization to your simulation package, you must be able to:

- Identify the simulation objects that you want OptQuest to manipulate. These simulation objects will be defined as decision variables in OptQuest. An OptQuest solution is a set of values for the decision variables.
- Set the values of the simulation objects in the simulation model and run a simulation using these values. Optimization is an iterative process, so you must be able to repeat this set/run process for many iterations.
- You must be able to identify simulation outputs that are affected by the decision variables. The outputs may be used to define the objective of the optimization or to define non-linear constraints on the optimization. For example, your objective may be to maximize Profit where Profit is an output of the simulation model. Simulation outputs used in non-linear constraints will be defined as requirement objects or as COptQuestUserControlledVariable objects.
- You must be able to determine when a simulation has completed and you must be able to retrieve the values of the simulation outputs. You need to tell OptQuest the values of the simulation outputs/user controlled variables so OptQuest can determine how to adjust the search for the best solution. OptQuest analyzes the results of the simulation and uses its intelligent search procedures to generate a new potential solution.

Advanced Features

The features of the general replication algorithm are available to you if your application can get control after each replication of a simulation. You will need to tell OptQuest the values of the simulation outputs for the replication just completed. (Not the averaged values for all replications run thus far.) OptQuest will keep track of the results for each replication and decide whether more replications are needed. See General replication algorithm.

If your simulation package supports the ability to go back and run more replications for a particular solution, then your simulation optimization can support running additional replications after an optimization has completed. It can also support Rank and Selection. For most simulation packages, you will need the ability to set one or more random seeds such that running a replication with a solution produces new results, not the same results seen in the original replications. See Running additional replications and Ranking and selection algorithm.

Decision variables

You will probably want to provide a user interface that will allow the user to define decision variables. You may want to create a list of all simulation objects that can be used as decision variables and allow the user to select from the list. You will need to define the type of the decision variable (continuous, discrete, integer, etc.) The variable type may be determined by the usage in the simulation model or it may be the choice of the user. You will need to specify the range of values for the decision variable and you will want to make this a meaningful range so that OptQuest doesn't waste time looking at solutions that don't make sense.

Decision variable names can include blanks and other special characters. If the decision variable names are used in string expressions, enclose the name in square brackets or curly braces in the expression.

Square brackets can also be nested so array elements can be named using nested square brackets ([MyArray[2]].)

User controlled variables

In most cases, your optimization definition will include outputs of the simulation. For example, your objective may be to maximize Profit or Throughput where Profit and Throughput are outputs of the simulation model. You may want to create a list of simulation outputs and let the user select from the list or you may infer which outputs are being used based on the definition of the objective function, or constraints the user defines.

Simulation outputs are defined as user controlled variables. OptQuest does not change user controlled variables but it will need to know the value of user controlled variables in order to evaluate a constraint or an objective function that has been defined as a string expression using user-controlled variables.

User controlled variable names can include blanks and other special characters. If the decision variable names are used in string expressions, enclose the name in square brackets or curly braces in the expression. Square brackets can also be nested so array elements can be named using nested square brackets ([Machine Output[2]].)

Constraints

Constraints restrict the values of the decision variables. For example, a budget constraint may dictate that the sum of the decision variables be below a certain value.

At the lowest level, OptQuest differentiates between linear and non-linear constraints. A linear constraint defines a linear relationship among decision variables. A constraint is non-linear if the expression is non-linear or the expression contains an output of the simulation (user controlled variable.) OptQuest can check a solution against a linear constraint and determine if the solution violates the constraint without running a simulation. If the solution violates the constraint, the solution is not sent to the simulation to be evaluated. Non-linear constraints require a simulation to be run.

String constraints allow you to combine decision variables and user controlled variables in an expression.

The distinction OptQuest makes between linear and non-linear constraints is hidden from you. It is a very powerful tool that allows you to create mathematical expressions and define relationships among decision variables and user controlled variables. You may want to provide an interface that allows your users to create constraint expressions.

The COptQuestSolution class provides methods to evaluate the left hand side and right hand side of a constraint string using the values in the solution. It also allows you to determine if the constraint was feasible for that solution. Let's say you have defined the following constraint:

$$100*(var1 + var2) \leq 50 * output1$$

You can evaluate the expression "100*(var1 + var2)" and then evaluate the expression "50 * output1" for a particular solution. (See Solutions for information on retrieving solutions.) The two values and their relationship can be displayed to the user. This can be very useful information for the user. Consider the case where the solution that produced the best objective value cost more than another solution. The cost savings of the inferior solution may be important to the user.

Objective

The objective is the goal of the optimization. The objective may be an output of the simulation or it can be a mathematical expression that uses decision variables and/or user-controlled variables representing outputs of the simulation.

If the objective is calculated by you or is the output from the simulation, instantiate a `COptQuestUserControlledObjective` and set its value in the `COptQuestOptimization.Evaluate(COptQuestSolution solution)` method. If the objective is a mathematical expression, instantiate a `COptQuestStringObjective` and set the expression using the `SetEquation()` method.

If the expression contains user-controlled variables, you will need to set their value in the `Evaluate()` method. You set the value for a user controlled variable using the `COptQuestSolution.SetVariableValue()` method. The OptQuest Engine will handle calculating a value for the objective given the string expression, the values of decision variables and the value of user-controlled variables used in the expression.

Optimization Options

You will want to allow your users to select the stop criteria. The OptQuest engine allows multiple stop options to be set. Optimization stops when the first stop criteria is satisfied. See [Defining stop criteria](#).

If your simulation package can support general replication, you may want to allow your users to set the parameters available for general replication. See [General replication algorithm](#).

Running the optimization

You will need to extend the `COptQuestOptimization` class and override the `Evaluate(COptQuestSolution solution)` method. If you want to provide feedback to the user while the optimization is in progress, override the `MonitorStatus(COptQuestSolution currentSolution)` method as well.

To start the optimization, call the `COptQuestOptimization::Optimize()` method.

When the `Evaluate()` method is called, OptQuest has a new solution that needs to be evaluated. Your `Evaluate()` method needs to do the following:

- For each decision variable, use the `COptQuestSolution.GetVariableValue()` method and set the value of the corresponding simulation object to the new value.
- Run the simulation.
- After the simulation has completed, retrieve the results from the simulation outputs. For each user controlled variable, call `COptQuestSolution.SetVariableValue()`.

If you are using general replication, the `Evaluate()` method will be called for each replication. If you need to do simulation initialization before running the first replication of a new simulation, the `COptQuestSolution .GetReplication()` method will return the replication number.

The `MonitorStatus()` method is called after OptQuest has evaluated the results that were set in the `Evaluate()` method to determine if there is a new best solution. If you are using general replication, `MonitorStatus()` will be called after each replication. The `IsLastReplication()` method of `COptQuestSolution` will return true if this is the last replication of the simulation. Note the `IsLastReplication()` is not valid in the `Evaluate()` method because the current solution has not completed evaluation.

From `MonitorStatus()` you can:

- Update a display of simulation number and replication number. The method `COptQuestSolution.GetIteration()` returns the simulation/iteration being run. If you are using general replication, the method `COptQuestSolution .GetReplication()` returns the replication number.
- If you are tracking the changes in the best solution, you can update a display from `MonitorStatus()`. We suggest you track the changes in the best objective function value. The `COptQuestOptimization.GetBestSolution()` method will return the best solution found thus far. If you are using general replication, you should update the display when `IsLastReplication()` is true.
- When `IsLastReplication()` is true, you may want to tell the user why replication processing terminated. See Replication termination

Displaying results

While the optimization is running (override the `COptQuestOptimization::MonitorStatus()` method)

- If you are using replications you may want to display the current iteration/simulation and replication. For example: "Iteration 10, Replication 3 of 6"
- A graph displaying the best objective function value at each simulation/iteration. We suggest you distinguish between feasible and infeasible solutions. A solution may produce an objective value that looks great, but it may violate one or more constraints.
- The values of the decision variables for the best solution and/or the current solution.
- You can display the feasibility of each constraint for the best solution and/or the current solution. Users may want to take a closer look at constraints that are infeasible. Perhaps the bounds of the constraint are too tight. See the Solutions section for information on retrieving a particular solution. You can use `COptQuestSolution` methods to retrieve the value of the left hand side and the right hand side to display information about individual constraints.

When the optimization completes

When the optimization completes, you may want to display the best N solutions.

- You could allow the user to specify how many solutions to be displayed. Use the `COptQuestOptimization::GetNthBestSolution()` method to retrieve each solution. The input parameter identifies the solution where the best solution is 1, second best is 2 and so on.
- You may want to evaluate the constraint expressions using the `COptQuestSolution` methods to retrieve the value of the left hand side and the right hand side.
- You may want to display the value of the user controlled variables at the Nth solution.

- If your simulation package can support running more replications of a particular solution, you may want to allow the user to select a set of solutions and run additional replications to refine the results or to perform Rank and Selection.

Solution replication information

The COptQuestSolution object contains detailed replication information. To retrieve a solution, use the COptQuestOptimization methods GetCurrentSolution(), GetBestSolution(), GetNthBestSolution() or GetIterationSolution(). From the solution, you can get the following information.

Is this the last replication for this iteration?	COptQuestSolution::IsLastReplication() - returns true if no more replications will be run for the current solution. This method should be called from MonitorStatus() or when the optimization is complete.
Number of replications evaluated	COptQuestSolution::GetReplication() - returns the number of the replication. If the method is called from COptQuestOptimization.Evaluate() it is the replication that is about to be evaluated. If it is called from COptQuestOptimization.MonitorStatus() it is the number of the replication that just completed evaluation.
The standard of deviation for the replications of a solution	COptQuestSolution::GetReplicationStandardDeviation()
The reason replications terminated for a solution	COptQuestSolution::GetCurrentReplicationTerminationReason() or COptQuestSolution::GetCurrentReplicationTerminationReasonString() - returns the reason replications were stopped 1 = REPLTERM_MAXREPLICATIONS = maximum replications were run 2 = REPLTERM_METCONFIDENCE = the confidence interval was met 3 = REPLTERM_NOTNEARBEST = the objective value was not near the best objective 4 = REPLTERM_REMOVEBYRAS = removed by rank and selection

General replication algorithm

The OptQuest Engine supports objective values that are based on experimentation through the General Replication Algorithm. This feature allows the user to provide the OptQuest Engine with results of multiple replications per simulation/iteration. The OptQuest Engine allows you to run a fixed number of replications per simulation or a varying number of replications per simulation.

Running a fixed number of replications

To run a fixed number of replications:

- Call COptQuestOptimization::SetUseReplications(true) to turn on replication processing
- Set the minimum number of replications and the maximum number of replications to the same value. Use the COptQuestOptimization::SetMinimumReplications() and COptQuestOptimization::SetMaximumReplications() methods.

- Call the `COptQuestOptimization::SetReplicationConfidence()` setting the confidence type to 0. The remaining parameters are ignored and can be any value.

Running a varying number of replications

The varying replications option allows OptQuest to test for the statistical significance between the mean of the objective in the current iteration/simulation (the "current mean value") and the best value found in previous iterations/simulations (the "best value"). The purpose of this test is to weed out inferior solutions without wasting too much time on them. The varying replications option can significantly speed up optimizations by eliminating inferior solutions before running the maximum number of replications.

When running a varying number of replications, you will specify a minimum number of replications to be run and a maximum number to be run. The OptQuest Engine will always run the minimum number of replications for a solution. OptQuest then determines if more replications are needed. The OptQuest Engines stops evaluating a solution when one of the following occurs:

- The maximum number of replications has been run.
- The true objective value is within a given percentage of the mean of the replications to date
- The current replication objective value is not converging

To run a varying number of replications:

- Call `COptQuestOptimization::SetUseReplications(true)` to turn on replication processing
- Set the minimum number of replications by calling the `COptQuestOptimization::SetMinimumReplications()` method. Set the maximum number of replications by calling `COptQuestOptimization::SetMaximumReplications()` method. The minimum and maximum replication counts should not be equal.
- For type 1 confidence testing, call the `COptQuestOptimization::SetReplicationConfidence()` setting the confidence type to 1. The 2nd parameter indicates the confidence level. The remaining parameter is ignored and can be any value.
- For type 2 confidence testing, call the `COptQuestOptimization::SetReplicationConfidence()` setting the confidence type to 2. The 2nd parameter indicates the confidence level and the 3rd parameter is the error percentage expressed as a value between 0 and 1.

The evaluations for the General Replication Algorithm include a replication number that provides a unique number from the iteration number. The replication number can be used to ensure that the trial run with this solution is unique from other trials run with the same solution.

The following methods are used for to define the parameters of general replication processing. All methods are on the `COptQuestOptimization` class.

<code>SetUseReplications()</code>	Used to turn on replication processing. If true, multiple replications of the same solution will be evaluated.
<code>GetUseReplications()</code>	Returns the <code>SetUseReplications()</code> value.
<code>SetMinimumReplications()</code>	Sets the minimum number of replications. The minimum is used to determine the minimum replications for each solution (default is 1).
<code>GetMinimumReplications()</code>	Returns the <code>SetMinimumReplications()</code> value.

SetMaximumReplications()	Sets the maximum number of replications. The maximum is used to ensure the algorithm never exceeds a certain number of replications (the default is no maximum).
GetMaximumReplications	Returns the SetMaximumReplications() value.
SetReplicationConfidence()	<p>Sets the confidence parameters:</p> <p>type 0 – No confidence level is used (default). A fixed number of replications will be run. 1 – Stop replications after minimum replications when confidence level is reached 2 – Stop replications after minimum replications when confidence level is reached, or when best solution does not fall within current solutions confidence level.</p> <p>level 1 = 80% 2 = 90% 3 = 95% 4 = 98% 5 = 99% 6 = 99.9%</p> <p>errPercent Percent of mean for which the confidence level is determined.</p>
IsSolutionMetConfidence()	Given an iteration number which identifies a solution, returns whether or not a solution has met the confidence level. Only valid for confidence type 1 and 2.
GetReplicationConfidenceType()	Gets the confidence parameter “type” described above.
GetReplicationConfidenceLevel()	Gets the confidence parameter “level” described above.
GetReplicationConfidencePercent()	Gets the confidence parameter “errPercent” described above.

Running additional replications

The OptQuest Engine supports running additional replications on a set of solutions after an initial optimization has completed. To support this feature, you must be able to setup the simulation such that you can go back and run more replications of a particular solution. For most simulation packages, you will need the ability to set one or more random seeds such that running a replication with a solution produces new results, not the same results seen in the original replications.

This feature can shorten the total optimization time by allowing the user to run a minimal set of replications on the initial optimization and then go back and refine a set of solutions. For example, the user may run 3 replications per simulation on the initial optimization and then run an additional 10 replications on the top 25 solutions.

To use this feature, you identify the top N solutions that should have additional replications by calling the `COptQuestOptimization::SetAdditionalReplicationsSolutions()` method. You can also specify specific solutions by calling `COptQuestOptimization::AddAdditionalReplicationsSolution()`. The solution is identified by the iteration number that generated the solution.

The `COptQuestOptimization::SetAdditionalReplications()` method identifies how many additional replications should be run. The `COptQuestOptimization::OptimizeAdditional()` method runs the additional replications.

The `Evaluate()` method will be called for each additional replication. Note that all the replications for a particular solution will be evaluated before the next solution is evaluated. For example, if the user wants 5 more replications run on solution A and solution B, the 5 additional replications will be run on solution A before any additional replications are run on solution B.

<p><code>SetAdditionalReplications()</code></p>	<p>This method can only be used if the original optimization used general replication. See <code>COptQuestOptimization::SetUseReplication()</code> method.</p> <p>This method specifies the additional replications you want run. The minimum number of replications that was specified in the <code>COptQuestOptimization::SetMinimumReplications()</code> method is increased by the number specified. No new solutions are generated. Additional replications are run until the new minimum is satisfied.</p>
<p><code>IsAdditionalReplicationsMode()</code></p>	<p>This method can be used in the <code>Evaluate()</code> or <code>MonitorStatus()</code> methods to determine if <code>Evaluate()</code> was called to evaluate additional replications. <code>COptQuestSolution.GetIteration()</code> and <code>COptQuestSolution.GetReplication()</code> identify the iteration and replication of a solution.</p>
<p><code>SetAdditionalReplicationsSolutions()</code></p>	<p>Specifies the top "n" solutions that you want additional replications to be run on. The default is to run additional replications on all solutions.</p>
<p><code>AddAdditionalReplicationsSolution()</code></p>	<p>This method allows you specify a specific solution. The solution is identified by the iteration that generated the solution.</p>
<p><code>OptimizeAdditional()</code></p>	<p>Runs the additional optimization. No new solutions are generated.</p>

Ranking and selection algorithm

The Rank and Selection algorithm can be run on a set of solutions after an initial optimization with replications has completed. This feature refines the list of best solutions using algorithms defined in the paper "Selection-of-the-Best Procedures for Optimization via Simulation" by Jutta Pichitlamken and Barry L. Nelson.

To support this feature, you must be able to setup the simulation such that you can go back and run an additional replication of a particular solution. For most simulation packages, you will need the ability to set one or more random seeds such that running an additional replication for a solution produces new result, not a result seen in the original replications.

The Ranking and Selection Algorithm eliminates solutions from a candidate list until there is a defined percentage chance that the remaining candidates are at most the indifference zone value from the true best in the candidate list.

To use this feature, you identify the top N solutions that should have additional replications by calling the `COptQuestOptimization::SeRankAndSelectionSolutions()` method. You can also specify specific solutions by calling `COptQuestOptimization::AddRankAndSelectionSolution()`. The solution is identified by the iteration number that generated the solution.

You will need to specify the maximum number of replication to be run. Call the `COptQuestOptimization::SetMaximumReplications()` to define the maximum. Rank and selection will stop

The `COptQuestOptimization::SetRankAndSelectionIndifferenceZone()` method allows you to set the indifference zone. The `COptQuestOptimization::SetRankAndSelectCorrectProbability()` allows you to set the probability. The `COptQuestOptimization::RankAndSelect()` runs the rank and select algorithms.

The `Evaluate()` method will be called for each additional replication. Note that one additional replication is run for each solution. The set of solutions is then checked to see which solutions need additional replications and one additional replication is run on those solutions. This cycle continues until no solutions are "in play" or the maximum number of replications has been run.

The Rank and Select algorithm can be repeated to continuing refining the solutions.

<code>SetRankAndSelectionSolutions</code>	Selects the best "numSolutions" as candidates for the Ranking and selection algorithm. If you do not specify any solutions, the OptQuest Engine will select the best 1% (minimum of 20) of the solutions for the Ranking and Selection Algorithm.
<code>SetRankAndSelectionIndifferenceZone</code>	Set the Indifference Zone used in the Ranking and Selection Algorithm. The Indifference Zone is the difference between two solutions that you want to detect. The default is calculated as follows until you have a non-zero Indifference Zone: <ul style="list-style-type: none"> • 10% of the difference between the best 2 solutions • 1% of the objective value • .00001
<code>SetRankAndSelectionCorrectProbability</code>	Sets the probability used in the Ranking and Selection Algorithm. The value is the probability that you selected the correct solutions within the Indifference Zone. The default is 0.95.
<code>AddRankAndSelectionSolution</code>	Adds additional solutions to the Ranking and Selection Algorithm. Allows you to add solutions

	that may not be in the top "numSolutions".
RankAndSelect	Runs the rank and selection algorithms on the selected set of solutions. No new solutions are generated.
IsRankAndSelectionMode	In Evaluate(), this returns true if you are in the Ranking and Selection Algorithm.

Portfolio optimization

Overview

An OptFolio® optimization consists of projects, one or more objectives, and optional constraints. Projects have a value (or set of values) associated with the project. The objective may be to maximize some measure of portfolio return, minimize some measure of portfolio risk, or to maximize a score (which may reflect a mix of quantitative and qualitative criteria) associated with the projects. Constraints come in the form of resource, project interaction, and balancing or averaging constraints.

Projects can be evaluated with a single value derived from a model or several values using Monte Carlo simulation. The simulation uses distribution definitions, along with cancellation points, to create many observations of NPV values. The simulation can also use correlations among the projects when creating the observations. Using simulation allows statistics on the measure (NPV, IRR, Payback Period, etc.), such as mean, standard deviation, percentiles, etc. to be used as objectives. Several statistics can be used as objectives, such as maximizing the mean, while the standard deviation is below a given threshold. This simulation technology could be adapted to decision tree analyses.

Resource constraints can be placed on the projects. Multiple constraints per resource can be used to reflect resource availability for each time period. In addition, multiple resource constraints can be used to reflect different resources (e.g. labor, cash).

Several scenarios for a project can be defined and placed in a mutually exclusive group to have the optimizer pick at most one (or exactly one) from this group. The mutually exclusive feature can also be used to choose between different starting times to better match resource availabilities.

Project dependencies can be enforced by using dependency constraints. A constraint is placed on any portfolio, such that if one project (A) is selected, then another project (B) must also be selected.

However, the second project (B) may be selected without the first (A). If two projects must be selected together, a “super” project should be created, which includes both projects.

Balance constraints can be used to balance a portfolio. You can use a balance constraint that ensures that x% of your portfolio must be selected from projects that have a given property. Balance constraints can also be used to ensure that no more than y% of your portfolio is selected from projects that have a specific property.

Average constraints can be used to make sure that the average value of some property of the projects in your portfolio is above or below a certain value. For example, projects can be assigned a certain score, based on a mix of qualitative and quantitative criteria not captured by NPV or other traditional financial or risk measures. Then, a constraint can be used to reduce risk by choosing a portfolio that has an average score above a certain target value.

OptQuest's portfolio optimization supports the following measures.

- NPV (Net Present Value): The Net Present Value is the cash flow from each period discounted by the discount rate per period.
- IRR (Internal Rate of Return): The Internal Rate of Return is the percentage return on the investment(s). It reflects the point at which the rate reflects a zero NPV.
- PBP (Payback Period): The Payback Period is the time in which it takes to pay back the original investment(s). The calculation for the Payback Period uses the discount rate. The Payback Period assumes a linear Cash Flow during the period to interpolate the payback period that occurs within the period.

You define a portfolio optimization by defining

- The objective for the portfolio optimization which is defined by a measure and a statistic on the measure.
- Portfolio project decision variables
- Portfolio project observations
- Optionally, you can add statistical requirements. The statistical requirement defines a measure and a statistic on the measure and a bound.

In the simplest case, the optimization evaluation can be done entirely within the OptQuest Engine. However, if the portfolio problem includes user-controlled variables, or the objective or requirements require special evaluation, you must extend the `COptQuestOptimization` class and provide an override to the `Evaluate(COptQuestSolution solution)` method. The `Evaluate()` method would provide the custom calculations specific to your application while the OptQuest Engine would provide the portfolio search methods and the statistical evaluations of portfolio objective and requirements.

The projects are defined as decision variables which include the observations for that project's measure. Additional decision variables can be defined which are used along with the project variables and constraints to provide certain project characteristics. For example, mutually exclusive projects require additional decision variables and constraints using those variables.

The objective and requirements can be defined as statistics on the dot product of the solution and observations. For example, you can maximize the mean of the measure while putting an upper requirement on the standard deviation.

Portfolio project decision variables

A portfolio project decision variable is defined for each project in the portfolio. Typically, the portfolio project decision variable represents the percentage of investment in the project. For each portfolio project, you must also provide the observations for that project's measure. The observations are used to calculate statistical measures and in searching for possible solutions.

Typically, a project decision variable is defined with a lower bound of 0 and an upper bound of 1 and its value represents the percentage of investment in that project. Portfolio optimization supports 3 types of project investment.

Case 1: The project can be funded at any level between the lower bound and the upper bound.

Case 2: The project must be funded above a certain percentage or not at all.

Case 3: The project must be funded 100% or not at all.

To define a portfolio project decision variable, instantiate a `COptQuestProjectVariable` and add it to the optimization using the `COptQuestOptimization.AddVariable()` method.

For case 1 the lower bound and the upper bound of the project variable define the range of investment.

For case 2, the lower bound is set to 0 and the `COptQuestProjectVariable::SetExclusiveRange()` method is used to define the range of variables that must be excluded from the solution. For example, if a project's funding can be 0 or any value greater than .25 (25%), you would define the lower bound as 0, the upper bound as 1 and define an exclusive range between 0 and .25. Values greater than 0 and less than .25 would not be allowed.

For case 3, the project is defined with a low bound of 0, a high bound of 1 and an exclusive range of 0 to 1.

Portfolio project observations

Each portfolio project decision variable has a set of observations that are either simulated values or sampled values. If there is only one observation, it represents a single known value for the project. Note that some statistics, such as standard deviation, require more than one observation.

The observation values sent to the OptQuest Engine can be either:

- Net Present Values (NPV)
- Cash flow by period

If the measure used is NPV, the project's NPV for each observation can be calculated and the set of NPV observations given to the OptQuest Engine. Other measures such as Internal Rate of Return (IRR) and Pay Back Period (PBP), require the cash flow for each period. If you are using Internal Rate of Return or Pay Back Period, the cash flow observations for each period are given to the OptQuest Engine.

Portfolio analysis of Net Present Value can use either NPV observations or cash flow by period observations. If the portfolio analysis only uses Net Present Value, it is more efficient to use the NPV observation. However, if different measures are used in the portfolio analysis, the Cash Flow by period per observation can be used for all measures, including NPV.

To add observations to a project variable, use the `COptQuestProjectVariable.SetObservations()` method. You can also define observations by period. In this case, call the `COptQuestProjectVariable.SetNumberOfPeriods()` method and the `COptQuestProjectVariable.SetPeriodObservations()` method to define the observations for each period.

Discount rate

The discount rate is used to calculate the Net Present Value (NPV) or Payback Period (PBP) from the cash flow per period observations.

The discount rate can be set as a constant discount rate for all periods/observations or it can be set per period/observation. Setting the discount rate per period/observation allows the user to use simulation to simulate the discount rate when it is unknown. If the discount rate does not change per observation, but does change per period, set all of the observations for that period to the same discount rate.

The discount rate is defined as a `COptQuestDiscountRateVariable` and added to the optimization by the `COptQuestOptimization::AddVariable()` method. If the discount rate is a fixed value, use the `COptQuestDiscountRateVariable::SetCurrentValue()` method to set the rate. To set the discount rate per period, use the `COptQuestDiscountRateVariable::SetNumberOfPeriods()` method to set the number of periods and use the `COptQuestDiscountRateVariable::SetPeriodObservations()` to set the observations by period.

Constraints

Portfolio project variables can be used to define constraints in the same manner as other decision variables, such as continuous variables. A constraint can contain both portfolio project decision variables and non-portfolio decision variables.

Budget and resource constraints

You can define multiple budget and resource constraints. Budget constraints limit the total money spent for a period of time. If your portfolio optimization problem deals with multiple time periods, you could define multiple budget constraints. Resource constraints may define the number of people needed for a particular project. Like budget constraints, your portfolio problem may have multiple resource constraints.

For example, the following constraint sets a budget of \$25,000 for projects costing \$30,000 and \$50,000:

$$30000 * \text{Project1} + 50000 * \text{Project2} \leq 25000$$

If the constraint values are per period, the following constraints are used (where P1P1Expense is defined as the expense for Project 1 Period 1, etc.)

$$P1P1\text{Expense} * \text{Project1} + P2P1\text{Expense} * \text{Project2} \leq \text{Period1Budget}$$

$$P1P2\text{Expense} * \text{Project1} + P2P2\text{Expense} * \text{Project2} \leq \text{Period 2Budget}$$

$$P1P3\text{Expense} * \text{Project1} + P2P3\text{Expense} * \text{Project2} \leq \text{Period 3Budget}$$

If you had period budgets that could be carried over, you would create the constraints as follows:

$$P1P1\text{Expense} * \text{Project1} + P2P1\text{Expense} * P2 + P3P1\text{Expense} * P3 \leq \text{Period1Budget}$$

$$(P1P1\text{Expense} + P1P2\text{Expense}) * \text{Project1} + (P2P1\text{Expense} + P2P2\text{Expense}) * \text{Project2} \leq \text{Period1Budget} + \text{Period2Budget}$$

$$(P1P1\text{Expense} + P1P2\text{Expense} + P1P3\text{Expense}) * \text{Project1} + (P2P1\text{Expense} + P2P2\text{Expense} + P2P3\text{Expense}) * \text{Project2} \leq \text{Period1Budget} + \text{Period2Budget} + \text{Period3Budget}$$

Quality measure constraint

By assigning a quality or satisfaction measure to portfolio projects, you define a constraint that ensure a given level of quality is met for the selection of projects. For example, a project of level 5 (higher is better) and a project of level 3 might have the following constraint:

$$5 * \text{Project1} + 3 * \text{Project2} \geq 5$$

Mutually exclusive projects

A group of projects is mutually exclusive if one and only one project can be included in the portfolio. If the projects can only be 0 or 1 with no values in between the mutually exclusive constraint is defined as:

$$\text{Project1} + \text{Project2} + \dots + \text{ProjectN} \leq 1$$

If your projects are continuous (i.e. can take on values between 0 and 1), add a binary decision variable for each project in the mutually exclusive group of projects. Then define the following constraints:

$$\text{Project1} \leq b1$$

$$\text{Project2} \leq b2$$

...

$$\text{ProjectN} \leq bN$$

$$b_1 + b_2 + \dots + b_N \leq 1$$

This set of constraints will allow only one project from the group of projects to be included in the portfolio

If you are using COptQuestStringConstraints, you can enter the constraint expression as they appears above. If you are using COptQuestLEConstraint objects, transpose the expressions to the form:

$$\text{ProjectN} - b_N \leq 0$$

Delayed Start of Project

If a project can have different start dates to take into account resource availability or other constraints, a delayed start of the project can be implemented. To implement a delayed start of a project, a project for each potential start date is defined with the appropriate additions to objective and requirement calculations and additions to the constraints defined. Then all of the projects representing the different potential start dates are placed in a mutually exclusive group as defined above

Project dependencies

A project dependency defines a set of projects that must be included if the primary project is included.

For example, if Project1 and Project2 depend on Project3, then if Project 3 is included, Project1 and Project2 will also be included.

$$\text{Project1} \geq \text{Project3}$$

$$\text{Project2} \geq \text{Project3}$$

If you are using COptQuestStringConstraints, you can enter the constraint expression as they appears above. If you are using COptQuestGEConstraint objects, transpose the expressions to the form:

$$\text{Project1} - \text{Project3} \geq 0$$

$$\text{Project2} - \text{Project3} \geq 0$$

Average Score Constraint

By assigning a score to portfolio projects, you define a constraint that ensures a given average score is met for the selection of projects. For example, if we want the average score to be above a certain level "C" (which is a constant value), we want the following equation to be true for scores of 7, 5, and 3 for the three projects:

$$(7 * \text{Project1} + 5 * \text{Project2} + 3 * \text{Project3}) / (\text{Project1} + \text{Project2} + \text{Project3}) \geq C$$

Since the above equation is not linear, we can make it linear with the knowledge that the project values are always between 0 and 1 and that the coefficients are positive. The following transformations yield a linear constraint that can be used for an average constraint.

$$7 * \text{Project1} + 5 * \text{Project2} + 3 * \text{Project3} \geq C * (\text{Project1} + \text{Project2} + \text{Project3})$$

$$7 * \text{Project1} + 5 * \text{Project2} + 3 * \text{Project3} \geq C * \text{Project1} + C * \text{Project2} + C * \text{Project3}$$

$$(7-C) * \text{Project1} + (5-C) * \text{Project2} + (3-C) * \text{Project3} \geq 0$$

Balance Constraint

By assigning a type to portfolio projects, you define a constraint that ensures a given balance of types is met for the selection of projects. For example, if we want the type of project to be above a certain level "P" (which is a constant value), we want the following equation to be true if projects "Project1" and "Project2" have that property and the NPV for the three projects are 700, 500, and 200 for the three projects:

$$(700*Project1+500*Project2)/(700*Project1+500*Project2+300*Project3) \geq P$$

Since the above equation is not linear, we can make it linear with the knowledge that the project values are always between 0 and 1 and that the coefficients are positive. The following transformations yield a linear constraint that can be used for an average constraint.

$$700*Project1 + 500*Project2 \geq (700*P)*Project1+(500*P)*Project2+(300*P)*Project3$$

$$(700-700*P)*PROJECT1+(500-500*P)*p2+(300*P)*Project3 \geq 0$$

Portfolio requirement

Often a portfolio optimization problem seeks solutions that keep risk below a threshold where risk is defined by a statistic. A portfolio requirement allows the user to define a threshold, a measure and a statistic on the measure. The statistic is calculated on the dot product of the portfolio project variables and the project observations. The result is compared against the threshold. For example, you may define a risk requirement that says the semi-standard of deviation must be below a particular value.

The value for a portfolio requirement is calculated by the OptQuest Engine and compared against the portfolio requirement's threshold. If the value satisfies the threshold, the solution is feasible. If it does not satisfy the threshold, the solution is infeasible.

The supported statistics are described in Portfolio measures and statistics.

To define a portfolio requirement, define a `COptQuestPortfolioLowerRequirement` if the result must be \geq a specified bound or a `COptQuestPortfolioUpperRequirement` if the result must be \leq a specified bound.

The portfolio requirements allow you to specify a measure and a statistic on the measure. To add the portfolio requirement to the optimization use the `COptQuestOptimization.AddRequirement()` method.

Portfolio objective

The OptQuest Engine allows for calculation of certain measures within the Engine and allows for the calculation of user specific measures outside of the OptQuest Engine.

For measures that are supported by the OptQuest Engine, the evaluation of the objective can be done entirely within the OptQuest Engine without any additional code by the user. The portfolio optimization currently supports maximizing/minimizing one of the following measures.

- NPV (Net Present Value): The Net Present Value is the cash flow from each period (discounted by the discount rate per period if provided).
- IRR (Internal Rate of Return): The Internal Rate of Return is the percentage return on the investment(s). It reflects the point at which the rate reflects a zero NPV.
- PBP (Payback Period): The Payback Period is the time in which it takes to pay back the original investment(s). The calculation for the Payback Period uses the discount rate if provided. The Payback

Period assumes a linear Cash Flow during the period to interpolate the payback period that occurs within the period.

For measures that are not supported by the OptQuest Engine, or if the user prefers, the evaluation is done by the user. The set of projects being evaluated is provided to the user and the user provides the calculation of the measure and returns that objective value to the OptQuest Engine.

The statistics supported by OptQuest are described in Portfolio measures and statistics.

To define a portfolio objective, instantiate a `COptQuestPortfolioObjective` and add it to the optimization using the `COptQuestOptimization.AddObjective()` method.

Portfolio measures and statistics

Portfolio optimization supports the following statistics:

- **Mean** – The sum of the values divided by the number of values.
- **Median** - The midpoint of the values after they have been ordered from the smallest to the largest. There are as many values above the median as below the median.
- **“nth” Percentile** – The “nth” percent of the values after they have been ordered from the smallest to the largest. There are nth percent values below the median.
- **Standard Deviation** – The square root of the variance.
- **Semi-Standard Deviation**
 - Calculation 1: The square root of the semi-variance using calculation 1.
 - Calculation 2: The square root of the semi-variance using calculation 2.
- **Variance** – The arithmetic mean of the squared deviations from the mean.
- **Semi-Variance**
 - Calculation 1: The arithmetic mean of the squared deviations from the mean, for all values less than the mean. The mean is determined by dividing by the number of values that meet the criteria
 - Calculation 2: The arithmetic mean of the squared deviations from the mean, for all values less than a provided value. The mean is determined by dividing by the total number of values.
- **Coefficient of Variation** – The ratio of standard deviation to the mean.
- **Coefficient of Semi-Variation** –
 - Calculation 1: The ratio of semi-standard deviation to the mean using calculation 1 for semi-standard deviation.
 - Calculation 2: The ratio of semi-standard deviation to the mean using calculation 2 for semi-standard deviation.
- **Probability** – The probability (0 to 1) that the value is less than some target value.

If you are defining a portfolio optimization problem, you need to identify the statistic on a measure to be used to calculate the objective value. The solution and the observations for each project are used to create a dot product. The dot product is used to calculate the value for the statistic. The following

methods are valid for COptQuestPortfolioObjective, COptQuestPortfolioLowerRequirement and COptQuestPortfolioUpperRequirement.

SetPortfolioMeasure	Identifies the measure on the observations. Net Present Value (NPV) = 1 Internal Rate of Return (IRR) = 2 Payback Period (PBP) = 3
GetPortfolioMeasure	Returns the number identifying the measure.
SetPortfolioStatistic	Identifies the statistic associated with the objective. The target value for probability, percentile and semi standard deviation are set in the second parameter. The second parameter is ignored for all other statistics. Mean = 1 Median = 2 Percentile = 3 Standard deviation = 4 Semi standard deviation (calculation 1) = 5 Variance = 6 Semi variance (calculation 1) = 7 Coefficient of variation = 8 Coefficient of semi variance (calculation 1) = 9 Probability = 10 Semi standard deviation (calculation 2) = 11 Semi variance (calculation 2) = 12 Coefficient of semi variance (calculation 2) = 13
GetPortfolioStatistic	Returns the number identifying the statistic
GetPortfolioStatisticValue	For probability, percentile and semi standard deviation, returns the target value for the statistic. All other statistics return 0.
GetBestStatistic	Using the best solution, the engine calculates the value of the statistic on the specified measure.
GetNthBestStatistic	Using the nth best solution, the engine calculates the value of the statistic on the specified measure. The nth best solution is identified by the SetNthBest() method
GetCurrentStatistic	Using the current solution, the engine calculates the value of the statistic on the specified measure.
GetSolutionStatistic	Using the solution at iteration " n", the engine calculates the value of the statistic on the specified measure.

Running the optimization

The optimization is run for a given number of iterations. This number should be a function of the number of projects in the optimization. If there are any calculations that must be done by the user, each set of projects is given to the user to evaluate and return the objective or requirement from the user. The OptQuest Engine with OptFolio uses several heuristics to determine which set of projects to evaluate in its search for the best portfolio.

To start the portfolio optimization, call the Optimize() method of the COptQuestOptimization object. You do not need to provide an override for the Evaluate() method unless your portfolio optimization requires specialized evaluation not handled by the OptQuest Engine. For example, if your objective function calculates a statistic not supported by the engine, you will need to do the calculation in the Evaluate() method. If you want to provide feedback to the user while the optimization is in progress, override the MonitorStatus() method. Refer to the topic Solutions.

Parallel Implementation

Overview

Parallel implementation can be used to decrease the total optimization time by distributing the evaluation across multiple machines or processors. For example, if the evaluation requires a lengthy simulation, the total optimization time can be decreased by performing parallel simulations on multiple machines or processors.

Non-parallel optimization is a serial process where:

- The OptQuest Engine calculates possible solutions for the decision variables.
- A solution is evaluated and results are calculated for the objective function, non-linear constraints and user controlled variables. The results are given back to the OptQuest Engine.
- The results are analyzed by the OptQuest Engine and a new set of possible solutions is calculated.

In a parallel optimization, multiple solutions are evaluated in parallel and the results are reported to the OptQuest Engine as the evaluations complete. As a machine or processor finishes its evaluation, the evaluation of another solution can begin.

To run parallel evaluations, you must:

- Tell the OptQuest Engine how many evaluations can be done in parallel by calling `COptQuestOptimization.SetEvaluate()` or add an evaluator for each parallel evaluation by calling `COptQuestOptimization::AddEvaluate()`.
- Extend the `COptQuestOptimization` class and override the `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method. You are responsible for managing the parallel evaluation and returning results to the OptQuest Engine. If the solution of the `Evaluate()` method is null, OptQuest will not give another solution until it has received the results for an outstanding solution. This occurs when the maximum number of solutions is being evaluated or OptQuest needs all outstanding solutions in order to generate a new solution.

If the solution is null, you must wait for one of the processors to finish their evaluation, and return the result of the evaluation. If you return from the `Evaluate()` method without waiting for an outstanding evaluation to complete, the OptQuest Engine will immediately call the `Evaluate()` method with a null solution.

- When an evaluation has completed call the `COptQuestOptimization::EvaluateComplete(COptQuestSolution solution)` method to indicate the requirement, objective and user controlled values have been set.
- Override the `COptQuestOptimization::MonitorStatus(COptQuestSolution currentSolution)` method to track the progress of the optimization. You can use the `COptQuestOptimization::GetNumberOfCompletedIterations()` method to track the progress of the optimization. If `GetNumberOfCompletedIterations()` returns a value greater than 0, you can retrieve the best solution using the `COptQuestOptimization.GetBestSolution()` method.

COptQuestSolution objects and Parallel ID's

During evaluation, each `COptQuestSolution` object contains a parallel ID. If you called `COptQuestOptimization.SetEvaluate()` to define the number of parallel evaluators, the parallel ID will be between 1 and the number of parallel processes you defined. If you called

`COptQuestOptimization.AddEvaluate()` to add an evaluator, the parallel ID will be one of the ID's you specified. You can use the parallel ID to associate a solution with an evaluator.

Parallel Optimization Behavior

An optimization problem run in parallel mode will behave differently than the same optimization problem run in serial (non-parallel) mode. In fact, the same optimization problem run in parallel mode may produce different results each time it is run. To understand why this happens, you need some background information on the OptQuest Engine's solution generation.

Overview of solution generation

The OptQuest Engine uses information from evaluated solutions to create a new set of solutions.

Depending on the solution generation algorithm, OptQuest may generate one new solution or many new solutions. In a serial optimization, the OptQuest Engine waits until all the solutions in the previous list have been evaluated before using the solution results to generate a new set of solutions.

In a parallel optimization, you set the maximum number of solutions that the OptQuest Engine needs to provide. When all solutions are out for evaluation, the OptQuest Engine waits for the user to return a solution before it provides another solution.

In a parallel optimization, OptQuest takes advantage of the parallel evaluators by generating new solutions when needed even though it is still waiting for some solutions to complete evaluation. Since the OptQuest Engine uses the results from evaluated solutions to create new solutions, the order of completion and the number of completed solutions will impact solution generation. This means the number of evaluators and the processor speed of the evaluators affect the solution generation.

Typical implementation of a parallel optimization

In this implementation, the user takes full advantage of the parallel evaluators to increase the impact of the solution generation techniques of the OptQuest Engine. The results from this implementation are not necessarily reproducible.

1. Set the maximum number of parallel solutions to the number of evaluators you have available. Optionally, you can ask for one extra and keep it in a queue to make sure you can immediately start an evaluator when it has completed.
2. Create two queues:
 - The "to be evaluated" queue will be used to distribute solutions to evaluators.
 - The "completed evaluation" queue will hold the solutions that have been evaluated.
3. During the optimization:
 - If the maximum number of parallel solutions has been provided by the OptQuest Engine, wait for a solution to be put on the "completed evaluation" queue.
 - Put the current solution on a "to be evaluated" queue.
 - As solutions complete evaluation, place them on the "completed evaluation" queue.
 - Return solutions from the "completed evaluation" queue.

Implementation of a reproducible parallel optimization

You can implement parallel optimization in a manner that will give reproducible results each time the problem is run. The goals in this implementation are to return solutions to OptQuest in the same order you received them from OptQuest and to keep all evaluators busy. This implementation will also

accommodate some evaluations being done on fast processors and some being done on slower processors.

1. Define the maximum number of parallel solutions to be larger than the number of evaluators you have available.
2. Create two queues
 - The “to be evaluated” queue will be used to distribute solutions to evaluators.
 - The “completed evaluation” queue has solutions that have been evaluated.
3. During optimization:
 - When beginning the optimization, request all available solutions from OptQuest and put them on the “to be evaluated” queue.
 - As solutions complete evaluation, place them on the “completed evaluation” queue.
 - Return the solutions from the “completed evaluation” queue to OptQuest in the order you received them from OptQuest. OptQuest solutions have an “iteration” number that can be used to determine the order of the solutions.
 - Upon returning a solution, immediately get another solution and place it on the “to be evaluated” queue.
 -

To keep all evaluators busy, the number of parallel solutions must be larger than the number of evaluators. If this number is too small, there may be times when faster evaluators are waiting on a slower one to complete its evaluation since it would be next in line to be returned. A good rule of thumb is if the faster evaluator is “n” times faster than the slower, then multiply the number of evaluators by “n”.

It is important to note that the larger the number of parallel evaluators that are defined, the later the OptQuest Engine can start using the information from returned solutions in its generation algorithms. This may cause the number of total iterations needed to get to the same quality of solution to be more than in the serial mode.

Recommendation

To get the best result with the most efficient use of the parallel evaluators, the typical implementation is recommended. However, since it is not reproducible, the reproducible implementation is available for users that require reproducible results.

Parallel evaluation methods

OptQuest can provide the calling class with more than one solution to evaluate in parallel. The following methods are used for parallel evaluations.

SetEvaluate(int numParallelEvals)	SetEvaluate() is called once and indicates the number of solutions that can be evaluated in parallel. OptQuest will provide a solution for each parallel process. Once the maximum is reached, OptQuest will wait until one of the evaluations is complete before retrieving any more solutions. Calling SetEvaluate() is equivalent to calling AddEvaluate() n times with values 1 to n.
AddEvaluate(int pID)	AddEvaluate() adds one evaluator each time it is called. The input parameter assigns a parallel ID (pID) to the evaluator.
RemoveEvaluate(int pID)	RemoveEvaluate() removes the evaluator with the specified parallel ID.
Evaluate(COptQuestSolution solution)	An overloaded virtual method. The Evaluate() method should perform the following tasks:

	<ol style="list-style-type: none"> 1. Send the solution to a parallel evaluator. 2. Check for any returned evaluations from the parallel evaluators. If any are available, the method should set the value for the solution's objective, user controlled variables and requirements and call <code>EvaluateComplete(COptQuestSolution solution)</code> 3. Return execution to the <code>Optimize()</code> method. <p>The parallel evaluator (or the <code>Evaluate</code> method)) can access the values of the variables and set the requirements for a particular solution at any time. Use the <code>COptQuestSolution</code> methods to get and set the solution values.</p> <p>If the solution parameter of the <code>Evaluate()</code> method is null, there is no solution to evaluate because the maximum number of parallel solutions has been reached. The <code>Evaluate()</code> method should then wait until a evaluation is returned from the parallel evaluators. If execution is returned to <code>Optimize()</code>, it will simply call <code>Evaluate()</code> again with null solution. This could result in a tight loop using CPU resources.</p>
<code>EvaluateComplete(COptQuestSolution solution)</code>	The <code>EvaluateComplete()</code> method is used to indicate that the evaluation for one of the solutions is complete, the requirements have been set for each requirement object, and the objective value has been set.
<code>GetNumberOfCompletedIterations()</code>	Returns the number of iterations completed. Parallel evaluations that have not completed are not included in the count.

Parallel Implementation with Replications

The parallel implementation of the OptQuest Engine supports replications. This feature is used if your evaluation requires multiple replications or trials and you want to use the replication features of the OptQuest Engine. These features include the general replication algorithm and the ranking and selection algorithm.

When you are using replications, the OptQuest Engine forces the replications to be evaluated serially. If a solution is being evaluated, the OptQuest Engine does not return it for another replication until the results of the previous replication have been returned. This allows features such as Confidence Testing and Rank and Selection to work in the parallel implementation without running unneeded evaluations.

To run parallel evaluations with replications, you must:

- Call `COptQuestOptimization::SetUseReplications(True)` to indicate you are using replications
- Call `COptQuestOptimization::SetMinimumReplications()` and `COptQuestOptimization::SetMaximumReplications()` to set the minimum and maximum number of replications.
- You can turn on replication confidence testing by calling `COptQuestOptimization::SetReplicationConfidence()` to set the confidence testing parameters.

- Override the `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method. If your evaluation requires each replication to be performed on the same machine or processor, it is your responsibility to make sure all the replications for a solution are evaluated on the same machine or processor. The `COptQuestSolution` object contains a `pid` that may be helpful in guaranteeing all replications are evaluated on the same machine or processor.
- Return the results of the evaluation.

See General replication algorithm.

Stopping parallel evaluation

Optimizations with parallel evaluations stop when the stop criteria has been satisfied and all outstanding parallel evaluations have completed. If the stop criteria is 1000 iterations, no new solutions will be evaluated when iteration 1000 is complete, but the optimization will not terminate until all outstanding solutions have completed evaluation.

If the optimization is stopped by the `COptQuestOptimization.StopOptimization()` method, the optimization is stopped immediately. All outstanding parallel evaluations are ignored.

Neural Net

Overview

The OptQuest Engine provides a capability for defining and training a Multilayer Feedforward Neural Network (MFNN). The OptQuest Neural Net is different from other MFNN software in that the training of the network is done via optimization techniques unique to the OptQuest Engine. This allows for a very flexible network structure and activation functions.

The OptQuest Neural Network is constructed in the traditional way by creating nodes and connecting them via arcs. Nodes are defined as input, bias, hidden, or output nodes and are assigned a level in the network. Input and bias nodes are not restricted to the first level. Hidden nodes cannot be in the first or last level of the network. Output nodes must be in the last level of the network.

Arcs are defined as connecting two nodes. The network does not need to be fully connected and any arc can skip one or more layers to connect nodes that are not in consecutive layers.. Arcs cannot connect nodes on the same layer.

Each node in the OptQuest Neural Network has an input value.

Input node	The input to an Input node is a data value.
Bias node	The input to a Bias node is the value 1.0.
Hidden node	Hidden nodes can have one of two input values 1. The sum of the products of the predecessor node values and the weights of the connecting arcs. 2. The product of the products of the predecessor node values and the weights of the connecting arcs.
Output node	The sum of the products of the predecessor node values and the weights of the connecting arcs.

Each node in the OptQuest Neural Network has an activation function.

Input node	Input nodes have the identity function as their activation function.
Bias node	Bias nodes have the identity function as their activation function.
Hidden node	Hidden nodes can have one of two activation functions: 1. $[0,1]$ Sigmoid : $1.0/(1.0+e^{-(\text{slope}*(-\text{input value}))})$ 2. $[-1,1]$ Sigmoid: $2.0/(1.0+e^{-(\text{slope}*(-\text{input value}))}) - 1.0$ The "slope" of the sigmoid function is a user-defined parameter
Output node	Output nodes have the identity function as their activation function

Training of the weights is not done using traditional MFNN methods, but rather, the weights are adjusted within an OptQuest Engine optimization problem:

- The weights of the arcs are treated as the decision variables.
- The mean absolute deviation between the predicted values and the target values (raised to a user-defined power) in the training set is treated as the objective value to be minimized
- The weight of the arcs that connect to the output layer may be removed from the optimization problem and determined through linear regression. (The user can turn this option on and off.)

- An additional decision variable can be used to optimize the slope of the sigmoid function.

When dealing with large-scale neural networks, it may be more practical to optimize the weights by taking one subset at a time instead of considering the entire set of weights. The following options are available for selecting weights during the optimization process:

- All-weight training — Optimization of the entire set of weights
- Layer-by-layer training — Optimization of a weights associated to arcs in a layer, taking one layer at a time
- Node-by-node training — Optimization of weights associated with a node, taking each node at a time
- Random-subset training — Optimization of a subset of weights, where each subset of a user-specified size is sampled randomly from the entire set of weights

Creating nodes

Each node in the network is a neural net node object that is created with the following properties:

- Node number – a unique number in the network
- Type – the type of node
 - Input node: outgoing value equal to the inputs value.
 - Bias node: a 1.0 as the outgoing value.
 - Sum node: takes the sum of input values and goes through the activation function to provide the outgoing value.
 - Product node: takes the product of input values and goes through the activation function to provide the outgoing value.
- Level – the level the node is in the network. The network is traversed level by level.

You create a neural net node by instantiating the `COptNeuralNetNode` object, setting the node's properties and adding the node to the `COptNeuralNet` object.

Input node	used as one of the inputs to the network. Call the <code>COptNeuralNet::AddInputNode()</code> method to add the node to the neural net.
Bias node	used to create a bias value of 1.0 Call the <code>COptNeuralNet::AddBiasNode()</code> method to add the node to the neural net.
Hidden node	used as a hidden node in the network, not related to an input or output. Call the <code>COptNeuralNet::AddHiddenNode()</code> method to add the node to the neural net.
Output node	used as one of the outputs to the network. <code>COptNeuralNet::AddOutputNode()</code> method to add the node to the neural net.

Creating arcs

Each arc in the network goes from one node to another node and is a neural net arc object that is created with the following properties:

- Arc number – a unique number in the network.

- From Node – the node the arc comes from.
- To Node – the node the arc goes to.
- Weight – a weight on the arc. The outgoing value from the From Node is multiplied by the weight before it arrives at the To Node.

You create an arc by instantiating the `COptNeuralNetArc` object and adding the arc to the `COptNeuralNet` object using the `COptNeuralNet::AddArc()` method.

Training the neural net

After the nodes and arcs have been defined and added to the neural net, you can set weights on arcs or train the neural network on a training set.

The objective of the training is to minimize an average error. The default is the average squared error. The power for difference between the expected and predicted values can be modified using `COptNeuralNet::SetMeanErrorPower()`.

The training set is defined using the `COptNeuralNet::SetTrainingSetMemory()` method. You can specify the optimization method using one of the following `COptNeuralNet` methods:

<code>SetTrainingTypeAll()</code>	The entire set of arcs
<code>SetTrainingTypeLevel()</code>	The set of arcs coming into each level
<code>SetTrainingTypeNode()</code>	The set of arcs coming into each node
<code>SetTrainingTypeRandom()</code>	A randomly selected set of a user defined number of arcs

The number of iterations to use in training is set by the `COptNeuralNet::SetStoppingRuleIterations()` method. In addition, the training can stop when a certain error level has been reached by using the `COptNeuralNet::SetStoppingRuleError()` method.

Once the training set, training type and stopping rule have been set, you train the neural net by calling the `COptNeuralNet::Train()` method. During training, the status of the training can be monitored by overriding the `COptNeuralNet::MonitorStatus()` method which is called after each iteration of the training. You can track the progress by calling the `COptNeuralNet::GetBestObjectiveValue()` method.

Considerations in training

- The training methods can be used in combination, one after another, to refine the training.
- Linear regression can be used to set the weights going into the output node(s) (`COptNeuralNet::SetRegressionForLastLevel()`).
- Arc weights can be fixed by the user and therefore, not optimized by the training (`COptNeuralNetArc::SetFixed()`).
- After training, the structure can be saved in a file using `COptNeuralNet::WriteNetworkDefinition()`.

Using a neural net

A trained Neural Network that has been saved in a file can be read into a Neural Network Structure using the `COptNeuralNet::ReadNetworkDefinition()` method. Predictions for new input sets can be made using the `COptNeuralNet::Predict()` method. The Neural Network can also be further trained, or retrained, on a new training set.

Advanced topics

Exception Handling

When the OptQuest Engine encounters an error condition, it will throw a `COptQuestException`. A `COptQuestException` contains the following information:

- An exception error code.
- A reason code that gives additional information about the error. For example, the exception code `EXCEPT_BADCONEQUATION` which indicates an error in a constraint expression, may have the reason code `REASON_EQMISSINGCLOSEPAREN` which indicates a missing closing parentheses.
- An error token which is a string. This is used in string expressions for constraints or objectives and identifies the token that caused the error.

The `COptQuestException.Description()` method will return a text string describing the error.

Throwing a `COptQuestException`

You may want to throw a `COptQuestException` when there is a failure in the `Evaluate()` method. For example, if you are doing simulation optimization and there is an unrecoverable error in the simulation, you may want to throw a `COptQuestException` to stop the optimization.

To throw a `COptQuestException`, create an instance of the `COptQuestException` object. The parameters are an error number and a text string describing the error. You will get the error number by calling the `COptQuestOptimization::GetEvaluateException()` method. The text string can be any descriptive text.

For example:

```
throw new COptQuestException(this.GetEvaluateException(), "Error running simulation.");
```

Efficient Frontier

In some cases you may wish to find the best solutions for a range of constraint bounds. The graph of the bound versus the best objective function is called the efficient frontier. A typical example occurs in portfolio optimization where the constraint is a limitation on risk and the objective is the best return. OptQuest can automatically search for solutions on the efficient frontier by quickly adapting to changing constraint bounds. It can also be useful to know what effect a constraint has on the optimal solution, and what would happen if the constraint were relaxed or tightened.

A linear or non-linear constraint with variable bounds defines an Efficient Frontier optimization. The following is an example of a linear constraint with variable bounds.

$$\text{Var1} + 5 * \text{Var2} < (1000, 5000, 20000)$$

The left hand side will be evaluated against the value 1000, then 5000 and finally 20000. You define a constraint with variable bounds by instantiating a `COptQuestStringConstraint` and setting the expression using the `COptQuestStringConstraint.SetEquation()` method. If the left hand side or right hand side contains a comma separated list of values, the constraint defines an efficient frontier.

The OptQuest Engine will automatically decide when to advance to the next bound in the constraint. If the optimization stop criterion is a number of iterations, an equal number of iterations will be run for each

bound. If the optimization stop criteria is the number of seconds to run, an equal amount of time will be spent on each bound.

The OptQuest Engine also allows you to control when the optimization should advance to the next bound.

To use this feature, call the `COptQuestOptimization.SetUserControlledVariableConstraint(true)` before calling the `COptQuestOptimization.Optimize()` method. This tells the OptQuest Engine you will be controlling the Efficient Frontier. When you want the bound to be changed call the `COptQuestOptimization.ChangeVariableConstraint()`. This tells OptQuest to advance to the next bound.

If you control the `EfficientFrontier()` using the `COptQuestOptimization.SetUserControlledVariableConstraint(true)` your code will need to handle exceptions and terminations reasons that may occur at a particular bound. For example, the problem may be constraint infeasible at one of the bounds, all solutions for a bound have been enumerated, or OptQuest may not be able to generate new solutions for a particular bound. The OptQuest Engine automatically handles these conditions when the `SetUserControlledVariableConstraint()` feature is turned off (the default setting).

When the OptQuest Engine advances to the next bound of an efficient frontier it rechecks its database of solutions, looking to see if previous solutions are now feasible or infeasible, given the new bounds.

To capture the best solution at each bound, override the `EfficientFrontier()` method of the `COptQuestOptimization` class. This method is called when the final evaluation for a particular bound has completed.

Note that while the optimization is in progress, OptQuest may find a better solution for a previously evaluated bound. If your user interface is displaying the Efficient Frontier graph while the optimization is in progress, you may want to update previously evaluated bounds. Use the `COptQuestOptimization.GetNthEfficientFrontier(int boundIndex)` to retrieve the best solution for previously evaluated bounds. Use the method `COptQuestOptimization.IsNthEfficientFrontierFeasible(int boundIndex)` to determine if the best solution at the Efficient Frontier bound (`boundIndex`) is feasible or infeasible.

Modifying an optimization problem

The OptQuest Engine allows you to change an optimization problem "on the fly." For example, the optimization could run for some number of iterations, you could modify the problem and then continue the optimization. You could also run a complete optimization, make modifications and then run additional iterations. When an optimization problem is modified, all evaluated solutions are rechecked. It is possible that the changes will make previously feasible solutions infeasible or vice versa. It is also possible the changes will make the optimization problem constraint infeasible. If this occurs, the OptQuest Engine will throw a `COptQuestException`.

When changes are made while an optimization is in progress, the changes are applied immediately and all evaluated solutions are rechecked. If the changes are made after an optimization completes, the `COptQuestOptimization::Recalculate()` method will apply the changes and recheck all evaluated solutions but not generate any new solutions. The `COptQuestOptimization::OptimizeAdditional()` method will apply the changes, recheck all evaluated solutions and then continue the solution search using the new information. The additional iterations will utilize the information from the previously evaluated solutions to generate new solutions.

If you want to analyze the effect of a change within a set of solutions, use the `Recalculate()` method. If you want to see if the change can drive the search to a better solution, use the `OptimizeAdditional()` method.

For example, you may want to know what impact a constraint has on a solution. You could:

- Setup the optimization problem with the constraint
- Run the optimization and retrieve the best solution
- Remove the constraint and call `OptimizeAdditional()` to run additional iterations, allowing OptQuest to search for better solutions with the new problem definition.
- Retrieve the best solution.

Alternatively, you could:

- Setup the optimization problem without the constraint
- Run the optimization and retrieve the best solution
- Add the constraint and call `Recalculate()` which will reevaluate all existing solutions.
- Retrieve the best solution

The following tables summarize the changes you can make for each optimization object.

Modifying decision variables: `COptQuestContinuousVariable`, `COptQuestDesignVariable`, `COptQuestDiscreteVariable`, `COptQuestIntegerVariable`

Note: The OptQuest Engine will throw an `EXCEPT_CONSTRAINTINFEASIBLE` exception if changes to the variable bounds make the problem constraint infeasible.

Modification	Comments
Add a variable	You are not allowed to add a variable when an optimization is in progress.
Remove a variable	You are not allowed to remove a variable. However, you can fix the value of a decision variable by setting the lower bound and upper bound to the same value.
Change the bounds	Use the <code>COptQuestVariable::SetLowerBound()</code> , <code>COptQuestVariable::SetUpperBound()</code> methods. If the changes make the problem constraint infeasible, the OptQuest Engine will throw an <code>EXCEPT_CONSTRAINTINFEASIBLE</code> exception.

Modifying constraints: `COptQuestEQConstraint`, `COptQuestGEConstraint`, `COptQuestLEConstraint`

Note: The OptQuest Engine will throw an `EXCEPT_CONSTRAINTINFEASIBLE` exception if changes to the constraint make the problem constraint infeasible.

Modification	Comments
Add a constraint	Use the <code>COptQuestOptimization::AddConstraint()</code> method.
Remove a constraint	Use the <code>COptQuestOptimization::RemoveConstraint()</code> method.
Change the coefficient of a decision variable	Use the <code>COptQuestConstraint::ChangeVariableCoefficient()</code> method.
Add a variable to a constraint	Use the <code>COptQuestConstraint::AddVariable()</code> method.
Remove a variable from the constraint	Use the <code>COptQuestConstraint::RemoveVariable()</code> method.
Change the right hand side	Use the <code>COptQuestConstraint::SetRHS()</code> method.

Modifying requirements: COptQuestDualRequirement, COptQuestLowerRequirement, COptQuestUpperRequirement

Modification	Comments
Add a requirement	You are not allowed to add a requirement when an optimization is in progress.
Remove a requirement	Use the COptQuestOptimization::RemoveRequirement() method.
Change the requirement bounds	Use the COptQuestRequirement::SetLowerBound(), COptQuestRequirement::SetUpperBound() method.

Modifying portfolio requirements: COptQuestPortfolioLowerRequirement, COptQuestPortfolioUpperRequirement

Modification	Comments
Add a requirement	You are not allowed to add a requirement when an optimization is in progress.
Remove a requirement	Use the COptQuestOptimization::RemoveRequirement() method.
Change the requirement bounds	Use the COptQuestRequirement::SetLowerBound(), COptQuestRequirement::SetUpperBound() method.
Change statistic	Use the SetPortfolioStatistic() method of the portfolio requirement.

Modifying string constraints: COptQuestStringConstraint:

Modification	Comments
Add a linear string constraint	Use the COptQuestOptimization::AddConstraint() method
Add a non-linear string constraint	You are not allowed to add a non-linear string constraint during an optimization or after an optimization has completed.
Remove a string constraint	Use the COptQuestOptimization::RemoveConstraint() method
Change a linear string expression to another linear string expression	Use the COptQuestStringConstraint::SetEquation() method
Change a non-linear string expression to another non-linear string expression	Use the COptQuestStringConstraint::SetEquation() method
Change a non-linear string expression to a linear string expression or change a linear string expression to a non-linear string expression	You are not allowed to change the linearity of a string constraint during an optimization or after an optimization has completed.

Modifying the objective

Objective class	Comments
COptQuestUserControlledObjective	No changes allowed.
COptQuestStringObjective	Use the COptQuestStringObjective::SetEquation() method to change the objective expression

COptQuestPortfolioObjective	Use the COptQuestPortfolioObjective::SetPortfolioStatistic() method to change the statistic. The portfolio measure cannot be modified.
COptQuestMultipleObjective	To modify an objectives coefficient, use the COptQuestMultipleObjective::ChangeObjectiveCoefficient() method. Note: If the coefficient is zero, the objective has no affect.

Additional optimization

When an optimization has completed (the Optimize() method has returned), you can analyze the solutions and determine if additional optimization is needed. You can continue the optimization by doing one of the following:

- **Run more replications** - If the original optimization used general replication, you can run additional replications on all solutions or a selected set of solutions. No new solutions are created. Running additional replications
- **Run Rank and Select** - If the original optimization used general replication, then you can run Rank and Select to refine the top " n" solutions. Ranking and selection algorithm
- **Run more iterations** - You can run additional iterations which will generate new solutions. Running additional iterations
- **Suggest solutions to the OptQuest Engine** - You can suggest specific solutions to be tried. To suggest a solution, get a new, empty solution by calling the COptQuestOptimization::CreateSolution() method. You set the suggested values by calling the COptQuestSolution::SetVariableValue() method. Use the COptQuestOptimization::AddSuggestedSolution() to add the suggested solution to the set of solutions that will be evaluated. Call the COptQuestOptimization::SetAdditionalIterations() methods giving it the number of suggested solutions you added. Call the COptQuestOptimization.OptimizeAdditional() method is to evaluate the suggested solutions. Suggesting possible solutions

Permutation groups

If your optimization problem involves multiple sets of permutation variables, you can group permutation variables by using permutation groups.

Saving and using solutions

The OptQuest Engine can save the results of the optimization and then use those results to save time in calculating the objective value and the requirements. This method is useful when an optimization is to be run again and the time to calculate the objective and requirement values is considerable. For example, if a simulation is run to calculate the objective function, it may take minutes for each solution to be evaluated.

When using saved results, the OptQuest Engine will read the saved results into memory and then search those results for an identical solution. If the solution is found in the saved solutions, its objective value and requirements are used. If the solution is not found, the solution will be returned for evaluation.

SaveSolutions()	Saves the solution information in the specified file
UseSolutions()	Uses the solution information from the specified file.

Tuning OptQuest for a particular problem

The OptQuest Engine is designed to be an all purpose optimizer that can be used to solve a wide variety of optimization problems. The OptQuest Engine combines Tabu search, scatter search, integer programming, and neural networks into a single, composite search algorithm that provides maximum efficiency in identifying new scenarios. In some cases you may want to tune the OptQuest Engine to run efficiently against a particular type of problem. The OptQuest Engine uses a set of search parameters to guide the search for new solutions. By modifying the search parameters, you can tune OptQuest to run your problem efficiently.

When should this feature be used?

If you repeatedly run a particular optimization problem or a class of problems that are similar in nature, you may want to tune the OptQuest search parameters. If your application is providing optimization to a wide variety of problems, you should use the default search parameters shipped with the OptQuest Engine. The default parameters have been tuned to run against a wide variety of optimization problems.

How it works

You will be setting up two optimization problems where one optimization runs an outer model and a second optimization runs an inner model. The outer optimization model will manipulate the search parameters and track the results. The inner model is the optimization model you are tuning. The inner model will be run once for each iteration of the outer model. The result from the outer optimization is the settings for the search parameters that provide the best results for the inner model.

The COptQuestTuner class is used to run the outer optimization model. It will generate a new set of search parameters for each iteration. Each iteration of the COptQuestTuner class will run your inner model using those search parameters. To set up the outer model:

- Create an instance of the COptQuestTuner class.
- Create an instance of a COptQuestUserControlledObjective. The objective value reflects a “goodness” measure of the inner model using the set of search parameters. Note that if your inner model runs optimizations on multiple problems, you will need to design an objective value that combines the objective value from each problem into one objective value that is given to the outer model. This combined objective could simply be a sum of the objectives (perhaps normalized to take into account any difference in the magnitude of the objectives). In this case, the direction of the outer objective (minimize/maximize) should match the inner models. Or the objective could reflect the distance from an ideal solution. In this case, the direction of the outer model would be to minimize the distance.
- Override the COptQuestTuner.Evaluate() method. In the Evaluate() method:
 - Use the COptQuestTuner.GetCurrentSearchParameters() to get a COptQuestSearchParameters object. This object defines the search parameters to be used for the inner optimization model.
 - Create the inner model. Create a new instance of the COptQuestOptimization class that runs the model you are tuning. Pass the COptQuestSearchParameters object to the COptQuestOptimization constructor.

- When the inner optimization model completes, retrieve the best objective value (InnerOptimization.GetBestSolution().GetObjectiveValue().) This will be the current objective value for the outer model. (OuterOptimization.GetCurrentSolution().SetObjectiveValue().)

When the outer model has completed, use the COptQuestTuner.GetBestSearchParameters() method to retrieve the COptQuestSearchParameters object that gave the best results for the inner model. The COptQuestSearchParameters.Write() method can be used to write the settings to a file.

To incorporate the search parameters into your standard product:

- Create an instance of the COptQuestSearchParameters object.
- Set the search parameters to the values from the tuner. You can use the COptQuestSearchParameters.Read() method to read the values from the file produced by COptQuestSearchParameters.Write() or you can cut/paste the values from the file into your source code.
- When you create an instance of the COptQuestOptimization class, pass the COptQuestSearchParameters to the constructor.

Solution Search Algorithms

The OptQuest Engine uses many different heuristics and meta-heuristics to generate new solutions. The OptQuest Engine's solution search algorithm has been designed to be a "black-box," general purpose, global optimization algorithm. It has been designed to use many heuristics to find a global optimal solution quickly and it is guaranteed to work with all types of problems.

For most users, OptQuest's standard search algorithms are the best choice. Some optimization problems are better suited to use a specific solution generation algorithm. It is suggested that the user contact OptTek Systems, Inc. if they would like to explore how other options may improve the search for particular types of problems.

If you choose to use a specialized search algorithm, you will set up your optimization problem and then call the COptQuestOptimization.SetMetaHeruistic(int meta) method to identify the solution search algorithm. The integer constants that identify each meta-heuristic are defined in the COptQuestSearchParameters class. OptQuest will throw an exception if the option is not applicable to the type of optimization problem you have defined. The meta-heuristic is set once before the optimization begins and cannot be changed once the optimization has started. If you want to try different meta-heuristics, you must define a new optimization problem for each one.

Many of the search algorithms are documented in white papers that can be found on the internet. Many of the algorithms that have a stopping criterion or converge to a single solution have been modified to restart the algorithms so that the search can continue.

OptQuest supports the following meta-heuristics:

Default

The OptQuest Engine will choose the meta-heuristic that testing has shown works best for the types of variables and constraints that are part of the optimization problem.

COptQuestSearchParameters.META_STANDARD

The "Standard" option runs the meta-heuristic that combines many heuristics that have been incorporated in the OptQuest Engine over its lifetime. It uses Scatter Search as its primary search method and uses many other heuristics to improve the search.

COptQuestSearchParameters.META_DIVERSITY:

The “Diversity” option runs a simple heuristic that randomly finds diverse solutions. It is not searching for the best solution, but is generating diverse solutions for the user.

COptQuestSearchParameters.META_ENUMERATION:

The “Enumeration” option is used to enumerate all possible solutions. The OptQuest engine must be able to enumerate all possible values for the decision variables. (e.g. Discrete, Permutation) and the total number of solutions must be less than 100K. Many of the other options enumerate once the search cannot generate unique solutions, but this options provides a faster enumeration of the solutions.

COptQuestSearchParameters.META_GENETICALGORITHM:

The “GA” option implements a genetic algorithm with the typical techniques such as inheritance, mutation, selection, and crossover.

COptQuestSearchParameters.META_SCATTER:

The “Scatter Search” option uses combination techniques on the reference set of solutions. The search includes methods to diversify, update the reference set, select reference solutions to combine, and to combine solutions.

COptQuestSearchParameters.META_SPSA:

Simultaneous Perturbation Stochastic Approximation (SPSA): The “SPSA” option samples changes in two opposing directions (randomly chosen for all directions) to approximate a gradient and then moves in the best direction.

COptQuestSearchParameters.META_PARTICLESWARMOPTIMIZATION:

The “PSO” option implements a swarm optimization algorithm. Random solutions are generated and then they “swarm” toward both the global best and their own best.

COptQuestSearchParameters.META_DESIGNOFEXPERIMENTS:

The “DOE” option runs a set of experiments at the lower and upper bounds of each of the variables.

COptQuestSearchParameters.META_NELDERANDMEAD:

The “Nelder and Mead” option implements the simplex problem for finding a local minimum, where the each simplex is replaced with a new one to find the local minimum.

COptQuestSearchParameters.META_ADAPTIVE:

The “Adaptive” option uses most of the available solution generation heuristics to create solutions. It uses extreme diversification, outside diversification, inside intensification, and single variable mutation to provide a complete search.

COptQuestSearchParameters.META_ELECTROMAGNETISM:

The “ElectroMagnetism” option uses a technique that simulates a charge being put on a solution to move it or repel it from other solutions. A Nelder and Mead algorithm is used as a local search technique on the solutions generated by this option.

COptQuestSearchParameters.META_VARIABLE:

The “Variable” option using a binary search on each variable to attempt to improve the best solution.

OptQuest Solution Generation

Overview

The OptQuest Engine search algorithm is based on metaheuristics. A metaheuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality. The heuristics used within the OptQuest Engine are many and are guided by such a meta-strategy. The individual heuristics may use the best solution, a reference set of solutions, a defined subset of evaluated solutions, or the entire set of evaluated solutions to create new solutions in the search for global optimality.

The OptQuest Engine maintains an internal database of all unique solutions. Each solution contains the value for each decision variable. If the solution has been evaluated, the solution will also contain the values for each user-controlled variable, requirement and the objective.

Whenever OptQuest generates a new solution, it checks the database of solutions to be sure the new solution hasn't already been evaluated. When the database is filled, optimization continues but the oldest solution is removed from the database and the newest solution is kept. This means that the OptQuest Engine may generate a solution that has already been evaluated but is no longer in the database of solutions, so the solution may be evaluated a second time.

The OptQuest Engine allows you to set the size of the database. The minimum size is 3000 solutions. OptQuest does not limit the maximum number of solutions but since the database is kept in memory you should consider your machine configuration when setting the database size.

In some cases, you may want to turn off the duplicate solution checking. For example, if the evaluation of the objective is a fast calculation, it may be faster to evaluate a solution than to check the database to see if it has been evaluated.

SetDataBaseSize()	Sets the size of the database to the value specified.
GetDataBaseSize()	Returns the database size
SetCheckDup()	Turns duplicate solution checking on/off.
GetCheckDup()	Returns true if duplicate solution checking is turned on. Returns false if it is turned off.

Discrete variable enumeration

If an optimization problem uses only discrete variables, the OptQuest engine attempts to enumerate all possible solutions if OptQuest's metaheuristic search algorithms fail to generate new solutions. If the enumeration results in a set of solutions too large for the database of solutions, the OptQuest Engine selects a subset of the enumeration to evaluate.

If a complete enumeration is accomplished, the OptQuest Engine reports that the optimal solution has been found. If the problem is requirement-infeasible, the OptQuest engine reports that the problem is infeasible.

Suggesting possible solutions

If you have a guess about the possible solutions for an optimization problem, you can give the solutions to the OptQuest Engine. Suggested solutions can shorten the time it takes to find an optimal solution. You can suggest a solution before the optimization is started or while the optimization is running. If you add the suggested solutions before the optimization is started, they are evaluated early in the optimization. If you add the suggested solutions after the optimization has started, they are added to the list of solutions that will be evaluated and it may take several iterations before these solutions are evaluated.

If your suggested solution violates one or more linear constraints, the solution is modified to make it constraint feasible before it is returned to you for evaluation. As a result, you may not see your exact suggested values returned for evaluation, since they may have been changed to make your solution constraint feasible.

To add a suggested solution:

1. Use the `COptQuestOptimization.CreateEmptySolution()` method to get a new empty `COptQuestSolution`.
2. Set the value for each decision variable in the suggested solution using the `COptQuestSolution.SetVariableValue()` method.
2. Call the `COptQuestOptimization::AddSuggestedSolution()` method to add the solution to the list of solutions to be evaluated.

You can make multiple suggestions by repeating these steps. Any variables that you do not set will use the previously set suggested value. The default suggested value is the mid-point between the lower and upper bound.

Running only suggested solutions

If you want only your suggested solutions evaluated, do the following:

- Add the suggested solutions before calling the `COptQuestOptimization::Optimize()` method
- Set the number of iterations to be run to the number of solutions you added.
- Call the `COptQuestOptimization::Optimize()` method. No new solutions will be generated and the solutions you added will be evaluated.

Modifying a solution

The OptQuest Engine allows you to modify the current solution suggested by the OptQuest Engine by changing the values of the decision variables. If you use a value other than the value suggested by OptQuest, it is important to change the OptQuest solution to match the values you used to calculate the objective. Use the `COptQuestSolution.SetVariableValue()` method to assign a new value to a decision variable.

You will need to modify solutions in the `COptQuestOptimization::Evaluate(COptQuestSolution solution)` method.

If your modifications violate a linear constraint, the OptQuest Engine will throw a COptQuestException and terminate the optimization.

Constraints and solution generation

The OptQuest Engine can determine feasibility of linear constraints when it generates a solution, because it has all the information it needs to calculate a value and ensure its feasibility. Linear constraints may improve optimization performance as infeasible solutions are eliminated within the OptQuest Engine.

Optimization performance may suffer when you include non-linear constraints in the optimization problem. Non-linear constraints can be time-consuming to evaluate if the constraint requires a lengthy evaluation, such as running an entire simulation. Also, the OptQuest Engine tries to rule out undesirable solutions by examining the results of feasible and infeasible non-linear constraints.

It is important that the value of a non-linear constraint provide a measure that can be used by the OptQuest Engine to determine a distance from feasibility. **A requirement value that indicates a pass/fail condition will not help the OptQuest Engine in its search for solutions since every solution that fails is equally bad.** It is better to assign a range of values to the requirement where a solution that is closer to the bound of the requirement can be treated as superior to a solution whose value is farther away from the bound.

If the optimization is set to run for a specified amount of time, it may be necessary to increase the amount of time to compensate for the additional overhead of non-linear constraints.

Precision properties

When generating new solutions, the OptQuest Engine uses a variable precision value which is used to compare the values for decision variables and determine duplicate solutions. Two decision variables are considered to have the same value if their difference is \leq the variable precision * the range of the variable.

The default starting value for the variable precision is .01. As the OptQuest Engine searches for new solutions, this value is decreased.

The OptQuest Engine allows you to modify the starting value for the variable precision. Call the COptQuestOptimization::SetVarPrecision() method to set a new value.

The OptQuest Engine also uses an objective precision value which is used to compare the objective value of solutions and determine if one solution is superior to another. The default starting value for the objective precision is .01. As the optimization progresses, the value is decreased.

The OptQuest Engine allows you to modify the starting objective precision. Call the COptQuestOptimization::SetObjPrecision() method to set a new value.

SetObjPrecision()	Sets the starting value for the objective precision.
GetObjPrecision	Returns the value of the objective precision.
SetVarPrecision()	Sets the starting value for the variable precision.
GetVarPrecision	Returns the value of the variable precision.

Rejecting Solutions

The OptQuest Engine allows you to mark a solution as rejected. A solution that has been rejected will not be considered as a candidate for the best or Nth best solution. When new solutions are generated, rejected solutions are checked to make sure the new solution is not a duplicate of a rejected solution. To reject a solution, call the `COptQuestSolution::RejectSolution()` method from the `Evaluate()` method.

Non-linear string constraints

If you have defined one or more constraints as `COptQuestStringConstraint` and the constraint equation is non-linear but contains only decision variables, you can ask the OptQuest engine to automatically reject solutions that violate the non-linear constraint. If a solution violates the non-linear constraint, OptQuest can reject the solution as constraint infeasible and the solution will be rejected without evaluation.

<code>SetRejectNonLinear</code>	Directs the OptQuest Engine to reject solutions that violate non-linear constraints without calling the <code>Evaluate()</code> method. The default value is <code>False</code> .
<code>GetRejectNonLinear</code>	Returns the value of <code>SetRejectNonLinear</code>

Version and licensing information

Version information

You can determine the version of the OptQuest Engine by calling the `COptQuestOptimization::GetVersion()` method. This method returns a string identifying the version number of the OptQuest Engine.

Licensing

When you purchase the OptQuest Engine or request a trial version of the OptQuest Engine, you will be assigned a license ID. Use this ID as the input parameter to the `COptQuestOptimization::SetLicenseID()`.

<code>SetLicenseID()</code>	The OptQuest engine requires a valid license ID, which is provided to you at the time the product is purchased. <code>COptQuestOptimization::Optimize()</code> throws an exception if the license has not been set or is invalid.
-----------------------------	--

Glossary

binary variable

A binary variable is a discrete variable with a value of 0 or 1.

linear constraint

Linear constraints describe a linear relationship among decision variables. A linear constraint is a mathematical expression where linear terms (i.e., a coefficient multiplied by a decision variable) are added or subtracted and the resulting expression is forced to be greater-than-or-equal, less-than-or-equal, or exactly equal to a right-hand side value. For example, if the total budget for scanners (\$2000 each) and printers (\$1000 each) is \$5000, you can define this as:

$$2000 * \text{scanners} + 1000 * \text{printers} \leq 5000$$

In another example, if x_1 represents the number of spaces in buffer 1 and x_2 represents the number of spaces in buffer 2, the following constraint restricts the search to solutions for which buffer 2 has at least as many spaces as buffer 1.

$$x_2 - x_1 \geq 0$$

constraint-feasible

A feasible solution is one that satisfies all linear and non-linear constraints.

continuous variable

A continuous variable may take on any value between a user-defined lower bound and upper bound. A precision value identifies the minimum amount continuous variables can change; you can modify the precision value.

For more information, see **Variables Precision** in Optimization properties.

decision variable

A decision variable is a quantity that the decision-maker controls. For example, the number of nurses to employ during the morning shift in an emergency room may be a decision variable in an optimization model for labor scheduling. The OptQuest Engine manipulates decision variables in search of their optimal values.

design variable

Used for decisions where value of the variable represents an alternative, and not a quantity. Design variables are useful in optimization problems where the decision variables consist of choosing the best alternative from a catalog, and a larger number may not imply the commitment of more resources. Therefore, choice #10 may not be a more costly or better choice than choice #1. These variables are defined by a lower bound, an upper bound, and a step size that controls the number of choices available within the specified range.

discrete variable

A decision variable that begins at a lower bound and increments by a step size up to an upper bound.

efficient frontier

A series of optimal solutions based on an optimization with a variable requirement. The graphical representation of the frontier shows how the optimal objective function value relates to the requirement bound.

infeasible

Infeasibility occurs when no combination of values for the decision variables can satisfy a set of constraints. An infeasible solution does not imply that the problem itself is infeasible.

metaheuristics

A family of optimization approaches that includes scatter search, genetic algorithms, simulated annealing, Tabu search, and their hybrids.

objective function

A mathematical expression describing a relationship among decision variables, or the result of an operation (such as simulation), that uses the decision variables as inputs.

optimization

A process that searches for the optimal solution to a model or problem.

permutation variable

Permutation variables are used to solve sequencing problems, for example, the order in which paint should be mixed to minimize cleanup time between color changes. The value of a permutation variable represents the order within the sequence.

non-linear constraints (requirement)

A mathematical expression describing a non-linear relationship among decision variables and/or user-controlled variables. A non-linear constraint can also be a restriction on a simulation output that requires its value to fall within a specified range.

scatter search

A population-based metaheuristic that operates on a collection of reference points with the goal of finding high-quality solutions to an optimization problem.

Tabu search

A metaheuristic that uses search history and memory management to guide the problem-solving process. In its simplest form, memory prohibits the search from reinvestigating solutions that have already been

evaluated. In OptQuest, memory functions to encourage search diversification and intensification. These memory components divert the search from locally optimal solutions to find a globally optimal solution.

Glossary

E

enumeration variable:

Index

A		
Additional iterations	14	
Additional optimization	53	
Additional replications	25	
Auto Stop	13	
automatic stop	11	
B		
binary variables	63	
C		
constraints	6, 8, 9, 15, 22, 31, 32, 58, 63, 64	
defining	6	
constraints limit	32	
continuous variable	63	
continuous variables	3, 63	
COptNeuralNet	47	
COptNeuralNetArc	47	
COptNeuralNetNode	46	
COptQuestException	49, 58	
COptQuestOptimization	9, 58, 59	
COptQuestStringConstraint	20	
COptQuestUserControlledObjective	21	
Creating arcs	46	
Creating nodes	46	
D		
database size	57	
decision variables	3, 22, 30, 58, 63, 64	
design variable	63	
Discount rate	31	
discrete variable	3, 57, 63, 64	
discrete variable enumeration	57	
duplicate solutions	57	
E		
efficient frontier	49, 64	
EfficientFrontier	49	
equation strings	9	
equations	9	
string	9	
errors	12, 13	
Evaluate	21, 57, 58	
experimentation	23	
F		
feasibility	8, 15, 22, 59	
Feasible	8, 15, 22	
G		
General replication algorithm	19, 23	
GetVersion	61	
I		
infeasible	8, 15, 57, 59, 64	
J		
JavaDoc	2	
L		
Linear	6	
linear objective	12	
Linear-constraint-feasible	15	
Linear-constraint-infeasible	15	
M		
Metaheuristic	64	
metaheuristics	64	
MILP	5, 12	
Mutually Exclusive Projects	32	
N		
Neural Network	47	
Non-linear	8	
O		
objective function	5, 9, 15, 64	
objective precision	59	
P		
Parallel Implementation	39, 41, 42	
Pass/fail	59	
Permutation groups	53	
permutation variables	3, 64	
platforms	2	
Portfolio objective	34	
Portfolio requirement	34	
precision properties	59	
Project Dependencies	33	
Q		
Quality measure constraint	32	
R		
Rank And Select	26	
requirement-feasible	15	
requirement-infeasible	15, 57	
requirements	8, 15, 59, 64	
defining	8	
Resource constraints	32	
S		
scatter search	64	
solutions	8, 15, 57, 58, 59, 64	
Statistics	35	
stop criteria	11, 13, 21	
string expressions	9, 15	
suggested solution	58	
T		
Tabu search	64	
U		
user-controlled variables	5	
V		
version information	61	