4.4 The Solver Module

The Solver Module includes tools for formulating an equation set from user algorithms, and solving the equation set. The general equation set formulation process includes defining (1) variables to be iterated, (2) calculated quantities to be constrained, (3) a Figure-of-Merit (FOM) - if optimization is used- and (4) routines to be called which relate the variables, FOM and constraints. These four quantities are implemented in the Solver Module via a set of classes. Once the equation set is constructed, it may be solved with a choice of tools : (1) a non-linear equation solver with no optimization, (2) a constrained non-linear optimizer (calculus based) or (3) a Genetic Algorithm Optimizer. The equation set formulation is generally the same, for each of these three solver tools. In fact, you can switch from one solver tool to another for many problems.

4.4.1 Equation Set Setup

This section describes member functions from the Solver module, used in the construction of an equation set. These functions are also listed, and described in the file Solver.mod. Many of the procedures described below are implemented through the use of Variable, ObjectiveFunc, Constraint and Calculator classes (see the file Solver.cc). For convenience, we list descriptions of the primary Solver module equation set manipulation functions in Table 4.4.1 .

There are four main steps in the optimization problem formulation/setup:

- declaring the input quantities to be used as variables
- declaring the calculated quantities to be used in constraints
- declaring the quantity to be used as a figure of merit
- declaring the routines to be called as function evaluators

We'll refer to the collection of these tasks as constructing the Equation Set. In the rest of this section we will go through each of these steps for an "example" optimization problem (and discuss ways to reformulate the existing problem setup). The input file "opt.sc" which sets up this problem, is shown in Table 4.4.2. The first part of this script file simply defines some real quantities, and some routines which perform calculations with them. We will perform an optimization problem using these, but the procedures discussed here apply equally well to quantities and routines from user written modules utilizing the SUPERCODE shell.

Variable quantities:

Real Variables

Looking at the input quantities, we can pick certain quantities that could possibly be varied in the optimization problem. For example `x1` and `x2`. It is especially important to only set independent quantities which are not calculated to be variables[1]. The function we call to cast these quantities into variables is "addVariable". This function returns an Integer, so the form of the call is:

```
iv1 = addVariable("x1", x1, !Bounded, !Bounded, 0.,0., On);
```

where `iv1` is an Integer. After the call, `iv1` is assigned a value which will be used to subsequently reformulate the status/properties of the variables. AddVariable has 8 arguments:

1 - String identifier for the variable

2- the Real quantity that will be varied.

3- a Real scale factor to scale the variable with. Choose this value so that the product of it times the variable will be ~ 1.

4- an Integer switch to use a lower bound on the variable (1=yes, 0=no)

5- an Integer switch to use an upper bound on the variable (1=yes, 0=no)

6- the Real lower bound to use (ignored if lower bound switch ==0)

7- the Real upper bound to use (ignored if upper bound switch ==0)

8- an Integer switch setting as to whether the variable is active(1=yes, 0=no)

Several Integer quantities are predefined in the startup "SuperCode.sc" file for use in the addVariable calls:

```
Bounded == 1
On == 1
Off == 0
```

The beginning of the opt.sc calls sets up two potential variable quantities. These are all set with the active-flag set to `On` (i.e. activated). Note that we've not put bounds on one of the variable quantities. This is the place to do it - not in the constraint section below.

There is an important distinction between the Real quantity we want to vary (`x1`) and the Integer variable identifier for it (`iv1`). The former is the Real quantity which is used in the calculation. The latter is an Integer identifier to the Variable "object" which the Solver module uses for the equation set manipulation. If we want to display the value of the quantity `x1` we would type

[1] Setting a calculated quantitiy to be a variable, will cause the optimizer to override the calculated quantity in its solution search - resulting in total chaos.

“`x1;`”[2], or if we wanted to set the value of `x1` radius to 6, we would type “`x1 = 6;`”. If we wanted to switch the” `x1` variable” `iv1` to a de-active status, we would type “`deactivate(iv1)`”.

Discrete Variables

There is another class of variables (discreteVariable) which can be used with the Genetic Algorithm optimizer described below. The discrete variables are allow only discrete, uniformly spaced values (of which Integers are a sub-set). Discrete variables are assigned with the addDiscreteVariable function (see Table 4.4.1). The calculus based optimizers will not work if there is an active discreteVariable however. The form of adding a discrete Variable is

```
Integer discreteVar1 = addDisceteVariable(“x1”,x1,1., 1., 100., 0.5, On);
```

The arguments this functions takes are: (1) a String descriptor, (2) the Real quantity being varied, (2) a const Real scale factor, (4) the const Real lower limit, (5) the const Real upper limit, (6) the const Real increment value, and (7) a const Integer active status flag. In this case the allowed values the variable quantity `x1` may take are in the set (1., 1.5, 2., 2.5, …. 100).

Integer List Variables:

There is also an Integer List Variable class. This class allows you to have a uniques list of integers as a variable set. These integers are stored in a user defined Integer Vector. With this type of variable, Traveling Salesperson Problems (TSP) can be done with the GA optimizer. The way to add an integer List Variable by calling the routine

```
Void addIntListVariable(const String &name, Vector(Integer) &IntList,
                        const Integer &IntLen)
```

The first argument is the name of the integer list variable, the second argument is the user defined Integer Vector (which will hold the results), and the third argument is the length of the Integer List (currently limited to 100). An example application of this variable type is given in the GATSP.sc file.

Constraints:

The next step is to set up the potential calculated quantities to be constrained in the equation set. First we need to understand the form in-which the equation solvers expect to see the constraints. For an Equality constraint, the equation solver will be pointed to a Real quantity, and will try to force this quantity to 0. For an inequality constraint, the equation solver will be pointed to a Real quantity, and will try to force the quantity to be ≥ 0. The actual form of adding a constraint to the equation set is:

```
   ic1 = addConstraint(“c1”, c1, 1., Equality, On);
```

[2] Typing “`iv1;`” would echo a strange looking Integer value.

where `ic1` has type Integer - i.e. the return type of the addConstraint function. AddConstraint takes 5 arguments:

1 - a String name for the constraint

2 - the Real quantity we want to constrain

3 - a Real scale factor used to keep the constrained quantity near 1

4 - an Integer switch for Equality (1) or InEquality (0) constraint

5 - an Integer switch setting as to whether the variable is active (1=yes, 0=no)

Note that `On, Off, Equality` and `InEquality` are set to appropriate Integer values in the startup SuperCode.sc script file.

Figure-Of-Merit

For optimization problems[3], a figure-of-merit (FOM) must be specified. In the example.sc file we set up two possible FOM candidates with the addFOM call, e.g.:

```
i = addFOM(“fom”, f,  1., On);
```

The addFOM function returns an Integer type, and takes 4 arguments:

1 - a String name for the FOM

2 - the Real quantity to minimize[4]

3 - the scale factor to multiply the FOM by (see below)

4 - the active setting for the FOM

Note that the VMCON optimizer always minimizes the value of the FOM. To maximize a quantity, use a negative scale factor. Regarding the magnitude of the scale factor, the calculus based optimizer tends to proceed best, when $|\text{quantity} \times \text{scale factor}|$ is in the range 1-10. Also, only use one active FOM at a time. (When using the Genetic Algorithm optimizer, use a scale factor near one, and use the GAMinimize switch to control minimization/maximization). When in doubt as to how many active FOM’s there are, do a “`showEqnSet(0);`”.

Calculators

[3] If the active equation set has the same number of variables as constraints, and all of the constraints are equality constraints, the problem may be solved without optimization. You can set use_vmcon=0 to use another equation solver, Hybrid which may be a bit faster, but the variable bounds will not be enforced.

[4] In the addFOM example here the quantity to be minimized is refered to by its full scope - “Example::totalCost”. This is because the name totalCost was also (inadvertently) used in the TPXCost module, so to avoid ambiguity the full name has to be used. The Shell will tell you if there is any name-overlap. Even though you can use duplicate names in different modules, it’s not a good idea.

During the course of an optimization, the optimizer does many “function evaluations” in order to determine the status of the solution, and to construct “gradients”, etc.. To tell the optimizer to include a particular routine call in the function evaluation, use the addCalculator function, eg. :

```
icl1 = addCalculator(“constraint1”, con1, 1, On);
```

This function returns an Integer type, and has four arguments:

1 - a String name for the calculator

2 - the type Void routine to call

3 - the order to do the call in

4 - the active status of the calculator

Note that the routine to call must be of type Void (i.e. does not return an Integer or Real value). Also of particular importance is the call order (calculators are called in order of ascending magnitude). This determines when the routine will be called relative to the other active “calculators”. Be sure any input a calculation needs, will have already been done before the present calculation is called[5]. When in doubt as to the order of the active calculators of a particular problem, do a “showEqnSet(0);” on it before guessing on which order to add a new calculator.

General Equation Set Manipulation

Often it is desirable to modify the equation set. Sometimes this is done interactively in order to better understand why a particular solution ended up somewhere. Other times it is convenient to read in an existing input file that is close to what to want and simply modify certain parts of it. The functions described here are useful in these circumstances.

- `activate(const Integer &) / deactivate(const Integer &)`

All of the equation set quantities (variables, constraints, calculators and FOMs) can be in either an “active” or “deactive” state. Sometimes it is useful to deactivate a variable and/or constraint and rerun a problem, to better understand why a particular solution was reached. The active status of any variable, constraint, FOM or calculator can be toggled with these routines. These routines accept the Integer (variable, constraint, FOM or calculator) identifier as the argument `(e.g. deactivate(ic2)` in Table 4.4.2). These routines can be used to change the FOM being applied in an optimization as well. The entire equation set can be activated or deactivated with the routines `activateAll()` / `deactivateAll()` respectively. A particular equation set member active status can be queried with the `isActive(const Integer &)` function.

- `reScale(const Integer &, const Real &)`

The variables, constraints and FOM all have scale factors supplied in the argument lists of the defining functions. These scale factors can be modified with routine, which takes two arguments:

[5] Having circular references within a set of calculators will lead to numerical noise, and is a sure guarantee to poor optimizer behavior and frustration.

(1) the type Integer identifier of the variable (constraint or FOM), and (2) the new scale factor to use. An example for the quantities in Table 4.4.2 would be

```
reScale(f,-1.0);
```

which would re-scale the figure of merit from the original scaling vale of 1 to -1 (note this would result in maximizing `f` instead of minimizing it.

- `reType(const Integer &, const Integer &)`

Constraints can be switched to and from the Equality/InEquality status with the reType function. This function expects 2 arguments: (1) the type Integer constraint identifier, and (2) a constant Integer flag (1=Equality, 0=InEquality). See Table 4.4.1 for an example application. . (Note that the const Integers "`Equality`" and "`InEquality`" are predefined in the startup SuperCode.sc file.).

- `reBound(const Integer &, const Integer &, const Integer &, const Real &, const Real&)`

Variable bounds can be modified with the reBound routine. This routine takes 5 arguments, (1) the type Integer variable identifier, (2) A flag indicating whether to apply a lower bound (1=yes, 0=no), (3) A flag indicating whether to apply an upper bound (1=yes, 0=no), (4) the type Real lower bound (if needed) and (5) the type Real upper bound (if needed) . Examples of reBounding the variable iv1 in Table 4.4.1 would be:

```
reBound(iv1, Bounded, Bounded, 1., 10.); // restrict 1 ≤ x1 ≤ 10
reBound(iv1, !Bounded, Bounded, 0., 10.); // restrict x1 ≤ 10
reBound(iv1, Bounded, !Bounded, 1., 10.); // restrict 1 ≤ x1
reBound(iv1, Bounded, Bounded, -10., 10.); // restrict -10 ≤ x1 ≤ 10
```

This function does not yet work for discreteVariables. (Note that the const Integer Bounded is predefined in the startup SuperCode.sc file.).

- `reOrder(const Integer &, const Real&)`

The order inwhich a calculator is called can be changed with the reOrder function., This function accepts 2 arguments: (1) the type Integer calculator identifier, and (2) the type Integer new order number.

- `RemoveAll()`

This routine removes all variables, constraints, FOMs and calculators from the equation set.

- `noDepend( const Integer &, const Integer&)`

This routine specifies that a calculator (argument 1) is independent of a variable (argument 2). This can be used to speed up gradient evaluations in the calculus based solvers (see below) - but one should exercise caution using this feature. The example in Table 4.4.2,

```
noDepend(icl4, iv1);
```

indicates that calculator `icl4` is independent of variable `iv1`. A related routine can specify a range of variables that do not affect a calculator:

```
noDependRange(icl4, iv1, iv2);
```

indicates that calculator `icl4` is independent of all variable defined between iv1 and iv2.

- `runCalcs()`

This routine causes the active calculators to be each called one time in the specified order. This feature is useful for debugging, and benchmarking (e.g. - getting calculated quantities after manually overriding a variable value).

Displaying the Equation Set

The shell will catch quite a few improper equation setups, but not all. Be sure there is at least one active variable which affects each active constraint. For our simple example problem in Table 4.4.2, the optimizer doesn't have any problem. You can display the equation set though, with the routine:

```
ShowEqnSetBase(Ostream &, const Integer &)
```

where the first argument is the Ostream to dump the output to, and the second argument is a switch where if =0 displays only active equation set quantities, and if != 0 displays all equation set quantities. So, for example, the call

```
ShowEqnSetBase(cout,0);
```

will display the active equation set to the screen. An additional function (`ShowEqnSet(const Integer &)`) is provided in the startup script file SuperCode.sc, which always dumps data to the screen.

4.4.2 Problem Solving Methods

Once an equation set is defined, several tools are included in the Solver module to solve it. These tools are:

1) Hybrid [4.4.1] - a nonlinear equation solver
2) VMCON [4.4.2] - a nonlinear constrained optimization solver
3) Some GALIB [4.4.3] implementations (genetic algorithm optimization).

Hybrd

To use the non-linear equation solver Hybrd, the equation set should consist of an equal number of variables and constraints, and no InEquality constraints (any FOMs will be ignored). Set the switch `use_hybrd = 1`; With this tool, the bounds on the variables are not enforced.

VMCON

To use nonlinear constrained optimization package VMCON, set the switch `use_vmcon = 1`; (be sure that `useGA` is not =1). This package uses a calculus based optimization method (utilizing Powell's algorithm) to find an optimum. It can also be used to solve a set of equations with equal number of variables and constraints, and no InEquality constraints by the use of an automatically generated dummy FOM. Variable bounds are enforced with this package.

VMCON and HYBRID inputs

Some controlling inputs for both VMCON and HYBRD are:

maxfev = maximum number of major iterations to take [Integer]

tol = convergence absolute tolerance used to stop calculation (typically 10^{-3}-10^{-6}) [Real]

gdel = increment used to form gradients (should be less than tol) [Real]

useCentralDiff = Integer switch controlling central difference / forward difference gradient scheme. If the major iteration is < useCentralDiff, a central difference gradient scheme is used. Otherwise a forward difference scheme is used. This allows you to use a faster forward gradient scheme at the initial stages of the solution and switch to a more accurate central difference scheme in the final stages.

VMCON and HYBRID return status flag

The return status flag is called "info", and its possible values from VMCON are:

0 - Optimizer hasn't been called, or improper equation setup.

1 - Succesful optimization.

2 - Optimizer hit the maximum number of allowed iterations (maxfev). Try calling the optimizer again.

3 - Noise in function evaluation. Check for proper scale factors on the variables. (try setting tol higher)

4 - Singular matrix in solution process. Is there an constraint activated, which no variables affect?

5 - No feasible solution possible. Do a showEqnSet(0) and see what variables are at the bounds, and what inequality constraints are at the limits. Try opening up the solution domain.

Calling VMCON and HYBRID

Once a problem is formulated and set up as desired (via an input file, interactive commands, or some combination) the solver tools can be called with either:

solve(1)	- calls the solver once

doCase(n)	- calls the solver up to n times in succession until a solution is reached.

Genetic Algorithm Solver

Note: This feature is only available if SUPERCODE has been built with the environment variable GALIB set to the directory containing the installed class library Galib [4.4.3]. This requires previous installation of the GALIB on the machine you are building SuperCode on. GALIB should defined in the ".mak" file in the Config directory.

It is also possible to use a genetic algorithm (GA) optimization technique to solve the equation set. In this case a FOM must be supplied. Capability to include the effect of constraints is also included. It is also possible to use the discreteVariable types with the GA solver. This optimization feature is an implementation from the GALIB class library [4.4.3]. Presently only limited GA and genome types are provided. The genome type is the 1D Real Array genome. To use this solver, set "`useGA=1;`" . The equation set manipulation and solver calling commands are the same as discussed above for the calculus based solvers. Example input files for using the GA solver are given in Tables

GA input

crossoverFrac	- fraction of the population to be crossed-over during mating
mutationFrac	- fraction of the population to be mutated
popSize	- population size
alphaGACon	- Factor used in constraint implementation in fittness function (see below)
betaGACon	- Factor used in constraint implementation in fittness function (see below)
gammaGACon	- Factor used in constraint implementation in fittness function (see below)
GAMinimize	- set =0 (default) to maximize, =1 to minimize.
GAElite	- Switch to force initial population to include variable set initial guess as individual 1 (if GAElite ==1) . Also, if additionally whichGAType ==1, the best individual is always carried to the next generation.
nConverge	- number of generations to look back for convergence check (with whichTerminator = 1) - see also tol.
nGenerations	- Number of generations to run with whichTerminator = 2.
whichTerminator	- if == 2, stop after nGenerations generations. If ==1, stop after solution is within "tol" relative difference from solution nConverge generations before.
whichGAType	- if = 1, a simple GA is used, if = 2 a steady-state GA is used.
whichGAScaling	- if =1 a linear scaling is used, if =2 a truncation scaling is used.

whichGASelector - if =1 a roulette wheel selector scheme is used, if =2 a tournament selector scheme is used.

GA Fitness Function

As the GA bases the value of a particular variable set guess of a single "fitness" value, we construct a fitness function based on the FM and constraint residuals. We use a fitness function which degrades the FOM by an increasing amount as the constraints become less satisfied. For maximization cases, the fitness function is defined as

$$f = \frac{FOM^{g}}{e^{(aR^{b})}},$$

where

FOM - is the optimization figure-of-merit

R - sum over all scaled constraint residual values R_i, where

$$R_i = \begin{cases} 0, \; \textit{for satisfied inequaity constra}\,\mathrm{int} \\ \quad Min(10, |sf_i * r_i|), \textit{otherwise} \end{cases}$$

where sf_i is the scale factor and r_i is the residual for constraint i.

α - is alphaGACon

β - is betaGACon

γ - is gammaGACon

For minimization, the fitness function is:

$$f = \frac{1}{(1 + FOM)^{g}\, e^{(aR^{b})}} \; .$$

Thus, you can prescribe more (less) importance to the constraints by increasing (decreasing) alphaGACon, and/or betaGACon. Also individual constraints can be given more or less weighting by the relative value of their scale factor.

Note: when using the GA solver, do not expect to see extreme convergence as with a calculus based optimizer (e.g. many constraints resting on bounds, near machine precision). You should be satisfied with somewhat "fuzzy" results.

GA output

The normal equation set output is available with the `showEqnSetBase` routine described above. Additionally, GA specific information is available with the routine

```
showGABase(Ostream&)
```

which dumps the GA information to the prescribed Ostream.

Examples:

Two example script applications of the GA solver are given in Tables 4.4.3 (does a "Mexican Hat" problem with local maxima) and 4.4.4 (does a simple Traveling Salesperson Problem with an Integer List Variable).

4.4.3 PVM Implementation of Solver tools.

Note: to use this feature, the SUPERCODE must have been built with the environment variable PVM_ROOT defined to be the location of the installed PVM library. This requires previous installation of PVM on the system you are building SUPERCODE. . PVM_ROOT should be defined in the ".mak" file in the Config directory.

It is possible to use the optimization tools described above to perform optimizations using external C or FORTRAN codes as the function evaluators. Many copies of the external code can be simultaneously run on different machines, greatly reducing run time[6]. This is done using the Parallel Virtual Machine (PVM) [4.4.4] software. In this case, the "calculators" are not called to perform function evaluators, but rather an external executable is called. Variable values determined by the optimizer are sent to the external code via message passing, and FOM / constraint residuals calculated in the external code are sent back to SUPERCODE by message passing. Some work instituting message passing in the external code is necessary, but this should be minimal for a well written code. The PVM routines used by SUPERCODE are described in the PVM Module. In this section we concentrate on describing the implementation of PVM in the user "external code" to perform optimization. There are two main tasks needed to use to drive an external code, namely (1) setting up the virtual machine on which the external code will run, and (2) instituting message passing in the external code. Regarding SUPERCODE, no modifications are needed. The PVM implementation is the same for both VMCON and GALIB solvers.

Note, calling parallel versions of SUPERCODE itself is not yet implemented.

Setting up the Virtual Machine

[6] The number of other machines is limited by: (1) number of machines you have accounts on, (2) the external code must run on the other machines, (3) PVM must be installed on the machines, (4) the machines must have Internet IP addresses.

This step is described in detail in Ref. 4.4.4. Basically it consists of constructing a host file which lists the IP addresses of all the machines you want to be included as a node in the virtual machine. You can also include additional information, such as the directory containing the executable on each node. Then to start the virtual machine up, you simply type "`pvm hostFile`", where `hostFile` is the filename of the file containing the virtual machine. You can type "`conf`" at the resulting prompt to check on the virtual machine configuration. Or, you can type "`quit`" to stop the PVM console - but the virtual machine is really still running in the background. You'd type "`halt`" at the PVM console prompt to shut down the entire virtual machine[7].

Message Passing in the External Code.

The idea here is to pass in variable values picked by SUPERCODE optimizer to be used in the calculation in the external code. After the calculation is done, results from the external code are passed back to SUPERCODE. Doing this requires adding some message passing calls in the external code. Examples of implementing the message passing are shown in the following example files, distributed with SuperCode:

pvm.cc	- C++ code to do the first calculation in the case 1 of file Opt.sc (Table 4.4.2)
pvm.f	- FORTRAN code to do the first calculation in the case 1 of file Opt.sc (Table 4.4.2)
pvmGA.cc	- C++ source to do GA optimization calculation from file GATest2.sc
pvmGA.f	- FORTRAN source to do GA optimization calculation from file GATest2.sc
Makefile	- Makefile indicating how to build the external executable for pvmGA.cc and pvmGA.f on several platforms.

Note: the PVM FORTRAN string passing routines do not seem to work as advertised on some platforms, so a stub routine (c_stub.c) is supplied that uses the C versions of these calls. You must compile this routine and link with it, to use these stubs to call the C message passing routines from a FORTRAN code.

The message passing to receive the variable values should be put after the last possible assignment for the "variable" quantities (e.g. a namelist read in a FORTRAN code), and before any place these quantities are used in a calculation. In the message passing, the first quantity passed is the number of variables (Integer). Then a list of variable names and values are passed. The variable name is that assigned in the addVariable call (see above). This list of names and values should be parsed (see the examples) to properly assign the values to the right variable. Next the calculation is performed in the external code. Then the results are passed back to SUPERCODE. In the message passing from the external code to SUPERCODE, the FOM name and value are passed first. If any constraints included in the calculation, they are passed next (first the string name, then the value). The order of the constraints doesn't matter, since on the receiving end in SUPERCODE , the

[7] On some PVM installations, it is necessary to always have PVM running in order to start SuperCode. This is due to a bug in the PVM routine pvm_mytid.

constraint name is parsed to match with the appropriate constraint. Be sure the variable and constraint names you pass in the external code are exactly the same as those supplied in the addConstraint / addVariable calls in the equation set setup. Also, if you add or remove a variable / constraint in the equation setup, you must be sure to modify the message passing in the external code accordingly.

Doing the parallel run

Once the virtual machine is set up and running, and message passing has been built into the external code, running an optimization case is similar as before. The equation setup is the same, except instead of declaring calculators, you set

```
use_pvm = yes;
```

and tell SuperCode what the name of the external code with:

```
pvmCommand = “codeName”;
```

where `codeName` is the name of the executable. To do an optimization type

```
solve(1);
```

as before. Be sure the external executable is in the directory you specified when you configured the virtual machine in the hostFile. You can also run the external executable a single time on a single node of the virtual machine with the “SUPERCODE “variable values” (good idea for setup and debugging) with the command:

```
computeOneRemotely();
```

It is possible to display the status of the virtual machine from the SUPERCODE console with the PVM Module routine:

```
showPVMBase(cout);
```

(or the `showPVM()` routine defined in the SuperCode.sc startup file).

Table 4.4.1 . Equation Set Manipulation Functions . This represents a subset of the Solver Module equation set manipulators. Details on the argument lists for these functions can be found in the Solver.cc, and Solver.mod.

Function	**Return type**	**Description**
Variables:		
addVariable(…)	Integer	adds a quantity to the variable list the optimizer can vary in the solution process
addDiscreteVariable(…)	Integer	adds a quantity to the discrete variable list the optimizer can vary in the solution process
activate(Integer Variable)	Void	sets the active status of a predefined variable to On
deactivate(Integer Variable)	Void	sets the active status of a predefined variable to Off
reBound(…)	Void	changes to Bounds settings for a predefined variable.
Constraints		
addConstraint(…)	Integer	adds a constraint to the optimizer equation set
reType (Integer constraint, Integer Equality/InEquality)	Void	switches the type of a constraint to/from an Equality/InEquality.
activate(Integer Constraint)	Void	sets the active status of a predefined constraint to On
deactivate(Integer Constraint)	Void	sets the active status of a predefined constraint to Off
Figure-Of-Merit		
addFOM(…)	Integer	adds a FOM to the equation set
reScale(Integer FOM, Real scale-factor)	Void	changes the scale factor used by a particular FOM
activate(Integer FOM)	Void	sets the active status of a predefined FOM to On
deactivate(Integer FOM)	Void	sets the active status of a predefined FOM to Off

Function	**Return type**	**Description**
Calculators		
addCalculator(…)	Integer	adds a calculator to the equation set
reOrder(Integer Calculator, Integer Order)	Void	assigns a new calculation order to a Calculator
activate(Integer Calculator)	Void	sets the active status of a predefined calculator to On
deactivate(Integer Calculator)	Void	sets the active status of a predefined calculator to Off
Misc. Commands		
showEqnSet(0)	Void	display the active equation set to the screen
showEqnSet(1)	Void	display the entire equation set to the screen
runCalcs()	Void	run through the active calculators one time
doCase(n)	Void	try calling the optimizer up to “n” times in succession, until a solution is found.
solve(1)	Void	call the solver once

Table 4.4.2 Example script file for setup of an optimization run using VMCON.

```
/////////////////////////////////// -*- C++ -*- //////////////////////////////
//
// FILE NAME
//    opt.sc $Revision: 2.1 $
//
// AUTHOR
//    Scott Haney, LLNL, (510) 423-6308
//
// COPYRIGHT
//    Copyright (C) 1990-1996, The Regents of the University of California
//    and Lawrence Livermore National Laboratory.
//
// CREATED
//    August 21, 1991
//
// DESCRIPTION
//    Exercises the Optimize module by solving the example problems
//    from the VMCON manual.  Also shows how compiled routines
//    can call interpreted routines transparently.
//
///////////////////////////////////////////////////////////////////////////////

// Define and set initial guess for variables.

  Real x1 = 2.0, x2 = 2.0, c1, c2, c3, f;
  Integer i, ic1, ic2, ic3, icl1, icl2, icl3, icl4, iv1, iv2;

// Define the constraints.

Void con1()
{
  c1 = 1.0 - (2.0 * x2 -x1);
}

Void con2()
{
  c2 = 1 - (Sqr(x1) / 4.0 + Sqr(x2));
}

Void con3()
{
  c3 = x1 + x2 - 3.0;
}

Void con4()
{
}

// Define the figure of merit.

Void fom()
{
  f = Sqr(x1 - 2.0) + Sqr(x2 - 1.0);
}

// Set up the equation set.

iv1 = addVariable("x1", x1, 1.0, !Bounded, !Bounded, 0.0, 0.0, On);
iv2 = addVariable("x2", x2, 1.0, !Bounded, !Bounded, 0.0, 0.0, On);

ic1 = addConstraint("c1", c1, 1.0, Equality, On);
ic2 = addConstraint("c2", c2, 1.0, InEquality, On);
ic3 = addConstraint("c3", c3, 1.0, Equality, Off);
```

```
i = addFOM("fom", f, 1.0, On);

icl1 = addCalculator("constraint 1", con1, 1, On);
icl2 = addCalculator("constraint 2", con2, 2, On);
icl3 = addCalculator("constraint 3", con3, 3, Off);
icl4 = addCalculator("constraint 4", con4, 0, On);
i = addCalculator("fom calculator", fom, 4, On);

// Show example of 'noDepend' call.  We are saying that constraint
// calculator con4 does not depend on x1 or x2.

noDepend(icl4, iv1);
noDepend(icl4, iv2);

// Now go through the three examples in the VMCON manual:

cout << "Case 1:\n";

tol = 1.0e-7;
solve(1);
showEqnSet(0);

cout << "\nCase 2:\n";

x1 = x2 = 2.0;
reType(ic1, InEquality);
solve(1);
showEqnSet(0);

cout << "\nCase 3:\n";

x1 = x2 = 2.0;
deactivate(ic2);
deactivate(icl2);
activate(ic3);
activate(icl3);
solve(0);
showEqnSet(0);
```

Table 4.4.3 Example script file for setup of an optimization run using the GA solver.

```
//////////////////////////////// -*- C++ -*- //////////////////////////////
//
// FILE NAME
//    GAPOpt.sc
//
// AUTHOR
//    J. Galambos
//
// COPYRIGHT
//    None
//
// CREATED
//    12/20/1996
//
// DESCRIPTION
//    Exercises the Genetice Algorithm (GA) optimizer.
//    Does the Mexican hat (concentric circle) multimodal function
//    Optimization.
//
///////////////////////////////////////////////////////////////////////////


// Define and set initial guess for variables.

  Real x1 = 0.0, x2 = 0.0, c1, c2, c3, f;
  Integer i, ic1, ic2, ic3, icl1, icl2, icl3, icl4, iv1, iv2;
Real z;


// Define the calculations  (i.e. constraints + FOM)

Void Calc1()
{
    z = x1*x1 + x2*x2;

    f = 0.5 - ( Sqr(sin(sqrt(z)) ) - 0.5)/Sqr(1. + 0.001*(z));

}

// Set up the equation set.

// First the Variables. Both a "regular" continuous variable,
// bounded between -10 and 10.

iv1 = addVariable("x1", x1, 1.0, Bounded, Bounded, -10., 10., On);
iv2 = addVariable("x2", x2, 1.0, Bounded, Bounded,-10., 10., On);


// Add the constraints. (None)

// Add the Figure-of-Merit (FOM)

i = addFOM("fom", f, 1.0, On);

// Add a calculator for the solver to call, which relates everything:

icl1 = addCalculator("calculator 1", Calc1, 4, On);


// Some switches for the optimization:

useGA = 1;          // Don't use VMCON (calulus based optimizer). Will use GA.
popSize = 100;      // population size for GA
mutationFrac=0.001; // Mutation fraction for GA (should be ~ 1/pop-size)
```

```
nGeneration = 13;    // # of generations to run to check for convergence
whichTerminator = 1; // Use terminate upon convergence
nConverge = 20;    // # of generations to run to check for convergence
tol = 1.0e-2;      // convergence criteria
crossoverFrac=0.6; // cross-over fraction to use in reproduction
GAElite = 0;       // Elite option on, always uses best of previous generation.
iPrint=1;          // Verbose output
alphaGACon = 0.2;   // coefficient used in constraint degradation of FOM
betaGACon = 1.;    // coefficient used in constraint degradation of FOM
gammaGACon = 1.0;  // coefficient used in constraint degradation of FOM
GAMinimize=0;      // Maximize, don't minimize
whichGAType = 2;   // Use simple GA
whichGAScaling = 1;// Use uniform scaling
whichGASelector = 1; // Use roulette wheel selecto


x1 = 5.; x2 = 7.; // Initial guess:

// Let's do it and see what happens:

solve(1);
showEqnSet(0);
```

Table 4.4.3 Example script file for setup of an optimization run using the GA solver to do a Traveling Salesperson Problem.

```
////////////////////////////////// -*- C++ -*- //////////////////////////////
//
// FILE NAME
//    GAPTSP.sc
//
// AUTHOR
//    J. Galambos
//
// COPYRIGHT
//    None
//
// CREATED
//    8/97
//
// DESCRIPTION
//    Exercises the Genetice Algorithm (GA) optimizer.
//    Does the Traveling Sales Person optimization for a
//    simple square box route. Also exercises the Integer List Variable class.
//
///////////////////////////////////////////////////////////////////////////


// Define and set initial guess for variables.

Real distance;
Integer nTowns = 8;
RealVector rx(nTowns), ry(nTowns);
RealMatrix dist(nTowns, nTowns);
IntegerVector order;


Integer i,j;

// Set up points as a square box:

rx(1) =  0.; ry(1) =  0.;
rx(2) =  0.; ry(2) =  1.;
rx(3) =  0.; ry(3) =  2.;
rx(4) =  1.; ry(4) =  2.;
rx(5) =  2.; ry(5) =  2.;
rx(6) =  2.; ry(6) =  1.;
rx(7) =  2.; ry(7) =  0.;
rx(8) =  1.; ry(8) =  0.;

Real dx,dy;

for (i=1; i<= nTowns; i++)
    for(j=i; j<= nTowns; j++)
    {
      dx = rx(i) - rx(j);
      dy = ry(i) - ry(j);
      dist(j,i) = dist(i,j) = sqrt(dx*dx + dy*dy);
    }

// Calculator to find path length for a guess:

Void Calc1()
{
    Integer i;
    distance=0.;
    for (i=1; i< nTowns; i++)
      distance += dist(order(i), order(i+1));

    distance += dist(order(nTowns),order(1));  // bit to go back home

}


// Set up the equation set.

// First the Variable, an Integer List.
useIntList = True;
addIntListVariable("towns",order, nTowns);

// Add the constraints. (None)

// Add the Figure-of-Merit (FOM)

Integer FOM = addFOM("path", distance, 1.0, On);

// Add a calculator for the solver to call, which relates everything:

Integer calc = addCalculator("calculator", Calc1, 10, On);


// Some switches for the optimization:

useGA = 1;          // Don't use VMCON (calulus based optimizer). Will use GA.
popSize = 100;      // population size for GA
mutationFrac=0.1; // Mutation fraction for GA (should be ~ 1/pop-size)
crossoverFrac  = 1.0;
pReplaceFrac = 0.95;  // Fraction of population to replace (for SS type GA)

whichTerminator=2; // 1 = Use Terminate upon Convergence
                   // 2 = Use Terminate upon Generation
nConverge = 20;    // # of generations to run to check for convergence
                   // with whichTerminator=1;
tol = 1.0e-2;      // convergence criteria with whichTerminator=1;
nGeneration = 5;  // # of generations to run  with whichTerminator=2
GAElite = 0;       // Elite option on, always uses best of previous generation.
// For siple GA
iPrint=1;          // Verbose output
alphaGACon = 0.2;   // coefficient used in constraint degradation of FOM
betaGACon = 1.;    // coefficient used in constraint degradation of FOM
gammaGACon = 1.0;  // coefficient used in constraint degradation of FOM
GAMinimize=1;      // Minimize, don't maximize
whichGAType = 2;   // Use steady-state GA
whichGAScaling = 1;// Use uniform scaling
whichGASelector = 1; // Use roulette wheel selector


// Let's do it and see what happens:

Real et;
timerOn();

solve(1);
showEqnSet(0);

et = elapsedTime();
cout << "CPU Time (Sec) = " << et << "\n";
```

4.5 Sampler Module

This module can be used to perform “uncertainty” or “risk” analysis. The idea is to prescribe distribution functions to certain quantities. These quantities are typically inputs to a calculation which have some degree of uncertainty to them, and are better described by a distribution function rather than a fixed value. These presribed distribution functions can be statistically sampled from, in a set of calculations (typically involving 100’s to 1000’s of samples) , and results for specified calculated quantities are stored in new “output” distributions. This allows one to calculate the distribution of calculated quantities based on a set of input distributions for the dependent assumptions. The methods are implemented with a set of Classes for input and output probability distributions (see Sampler.cc).

Many of the features described here are used in the Sampler1.sc example case shown in Table 4.5.1.

4.5.1 Specifying Input Distributions

A probability distribution function (PDF) can be assigned to a Real variable with the following routines:

Uniform distribution:

```
Integer addUniformPDF( const String &name , Real &var, const Real &min, const
      Real &max,  Integer &seed, constInteger &onnoff),
```

where

name = name for the PDF

var = Real quantity to assign the PDF to

min = minimum value for the PDF

max = maximum value for the PDF

seed = seed for the random number generator

onoff = on / off status

Triangular distribution:

```
Integer addUniformPDF( const String &name , Real &var, const Real &min, const
      Real &peak, const Real &max,  Integer &seed, constInteger &onnoff),
```

where

name = name for the PDF

var = Real quantity to assign the PDF to

min = lower value for the PDF

peak = most likely value where the triangle peaks

max = upper value for the PDF

seed = seed for the random number generator

onoff = on / off status

Normal (Gaussian) distribution:

```
Integer addNormalPDF( const String &name , Real &var, const Real &mean, const
      Real &sigma,  const Integer &lowerTrunc, const Integer &upperTrunc,
      const Real &xmin, const Real &xmax, Integer &seed, constInteger
      &onnoff),
```

where

name = name for the PDF

var = Real quantity to assign the PDF to

mean = value where gaussian is maximum

sigma = Standard deviation of the gaussian

lowerTrunc = switch to use a lower truncation (if !=0)

upperTrunc = switch to use an upper truncation (if !=0)

min = minimum value for the PDF (if lowerTrunc !=0)

max = maximum value for the PDF (if upperTrunc !=0)

seed = seed for the random number generator

onoff = on / off status

The onoff active status can be set to On with the `turnOn(const Integer &whch)` routine, where whch is the Integer value assigned during the addxxxPDF call.

4.5.2 Specifying Output Distributions

An output distribution function (ODF) can be assigned to a Real variable with the following routines:

```
Integer addODF( const String &name , Real &var, const Real &scale, const
      Integer &onnoff),
```

where

name = name for the PDF

var = Real quantity to assign the PDF to scale = factor to scale var’s value by.

onoff = on / off status

Multiple ODF’s can be created.

4.5.3 Do the Calculation

The sampling is done with the `Void rollDice(const Integer &n)` routine. A call to this routine with argument n, results in n different (1) samplings of the PDFs, (2) performing the prescribed calculation, and storing of the ODF values. When this routine is called several times in succession, the previously collected statistics are maintained and added to. Thus you can see the effect of additional samples by calling rollDice multiple times.

The actual calculation performed depends on the setting of the switch `callSolve`. If `callSolve ==1`, the optimizer is called (see Solver Module) “n” times for each set of sampled PDF values. (Warning: this can be a time consuming calculation). In this case it is important to not assign a quantity to be both a variable (controlled by the optimizer) and to be a PDF (controlled by the Sampler Module). This will result in overriding the values picked by the optimizer with values sampled from the prescribed PDF, and render the optimization futile. If `callSolve !=1,` the active calculators declared with the Solver Module “addCalculator” routine are called once for each set of sampled values. No optimization/iterations are performed.

4.5.3 Miscellaneous PDF Manipulations

Calculate Moments

Various moments of the sampled PDFs and calculated ODFs are calculated with the Void doAnalysis() routine. This routine is called automatically with each call to routine rollDice (see above) .

Show Moments

The Void PDFPrint(Ostream &os) routine will print the calculated distribution moments to the Ostream os.

Dump PDF/ODF values

The contents of a PDF or ODF object can be dumped to a RealVector with `the Void dumpVals(const Integer &whch, RealVector &vec)` routine. Here whch is the Integer reference assigned to a PDF/ODF in the `addxxxPDF` or `addODF` call. For a PDF these values are the set of sampled values, and for an ODF these values are the set of calculated values. This feature is useful for processing the values in a PDF/ODF in the shell, for example to print or plot them.

Make Histogram (x,y) values

The routine:

```
Void makeHistogram(const RealVector &vec, const Integer &nBox,  RealVector
      &yp, RealVector &xp, RealVector &ycum, RealVector&xcum)
```

generates a set of (x,y) points for plotting a histogram. Here `vec` is a vector of data points to construct the histogram from (use dumpVals to generate this vector from a PDF/ODF), `nBox` is the number of boxes to use in the histogram, `yp` are the resulting y point values for the histogram, `xp` are the resulting x point values for the histogram, `ycum` are the resulting y point values for the cumulative distribution, and `xcum` are the resulting x point values for the cumulative distribution. The example case in Table 4.5.1 shows an example of generating and plotting (with the PLPLOT Module) a histogram.

Switches

latmon	- Integer switch to toggle from MonteCarlo sampleing (=0) or Latin HyperCube sampling (=1 , default).
quietRoll	- Integer switch to suppress the echoing of the sample number to the console (1=echo, 0=don't)
nRolls	- Integer that keeps track of how many samples have been done.

4.5.4 Minimizing Uncertainty

It is possible to use a combination of the Solver and Sampler module capabilities to minimize uncertainty, i.e. minimize the standard deviation of an ODF. In order to make this problem tracktable in CPU time and to be able to utilize the calculus based optimizer in this optimization, a special type of ODF class is provided - specialODF. Rather than performing statistical sampling to generate this ODF, it is calculated approximately by a series expansion of the PDFs [4.5.1]. With this scheme one can use the optimizer to pick the best set of controllable quantities, in order to minimize uncertainty in a calculated quantity due to uncertainty in other dependent quantities.

To use this feature first assign a calculated quantity to be a specialODF with the routine:

`Integer addSpecialODF(const String &name, Real &var, const Real &scale, const Integer &onoff)`, where

name	- the name of this special ODF
var	- the calculated quantity be considered as this special ODF
scale	- a scale factor to apply to var
onoff	- the On/Off toggle (1=on, 0=Off).

Then create an optimization equation set with the usual variables, constraints, calculators as described in the Solver Module section 4.5. Then add some normalPDFs to quantities that affect the specialODF, and are uncertain[8]. Make sure that none of these quantities are also variables. Next add a calculator which points to the routine `Sampler::findSpecialDist`() - make sure this calculator is called last. This routine will calculate the mean value of the specialODF (`meanSpecialDist`) and the standard deviation of the specialODF (`stDevSpecialDist`). You can then assign the optimization FOM to be the minimum "`stDevSpecialDist`" or you can use "`stDevSpecialDist`" in a constraint. When the optimizer is called, the routine `findSpecialDist`() will be called in the function evaluation - providing information about the magnitude of the uncertainty of the specialODF.

See the specialODF.sc example case.

[8] Presently, only gaussian PDFs are allowed with this feature.

Table 4.5.1 Example script file to exercise the Sample Module.

```
/////////////////////////////// -*- C++ -*- //////////////////////////////
//
// FILE NAME
//    $Id$
//
// AUTHORS
//    John Galambos
//
// CREATED
//    7/18/97
//
// DESCRIPTION
//   Sampler example
//
// REVISION HISTORY
//    $Log$
///////////////////////////////////////////////////////////////////////////


// Set up a simple calculation, where c_i depends on x_i:

Real x1, x2, x3, x4, x5;
Real c1, c2, c3;

Void calc1()
{
    c1 = x1 + x2 + x3 + x4 + x5;
    c2 = x1 * x2 * x3 * x4 * x5;
    c3 = 100.* (x1 + 2. *x2*x3 + sqrt(x4/x5));
}

Integer calculator = addCalculator("calc1", calc1, 1,On);


///////////////////////////////////////////////////////////////////////////
//
// Probability Distribution Function definitions
//
///////////////////////////////////////////////////////////////////////////

// Calculation definition:

 callSolve = False;              // no optimizations needed

// Sampling initializations:

 Integer seed = -1;    // set the random number seed
 latmon = 1;                 // Use LatinHyperCube instead of MonteCarlo

Integer


//     uniform dist. from 1 to 7 $/m3:

PDFx1  = addUniformPDF("x1",x1,1.,7.,seed,On),

// Normal dist, mean = 10, st. dev. = 3, no truncation

 PDFx3 = addNormalPDF("x2", x2, 10., 3., Off, Off, 0., 0.,seed, On),

// Normal dist, mean = 10, st. dev. = 2, lower truncation=1,
//    upper truncation = 30

PDFx3 = addNormalPDF("x3", x3, 10., 2., On, On, 1., 30.,seed, On),
// Triangular dist.: most likely = 10 , lower value = 1, upper value = 25

PDFx4 = addTriangPDF("x4", x4, 10., 25., 35., seed, On),

    PDFx5 = addTriangPDF("x5", x5, 1, 10., 25., seed, On);



// ****************************************
//  *** Set up output quantities to monitor:
// ****************************************

 Integer
 out1 = addODF("c1", c1, 1., On),
 out2 = addODF("c2", c2, 1.e-5, On), // scale by 1.e-5 to store values near 1
 out3 = addODF("c3", c3, 0.001, On); // scale by 0.001 to store values near 1

// Do it:

  rollDice(200);

// Show what happened to screen:

 PDFPrint(cout);

//  Set up plot for cost output distribution:

RealVector yv2,xv2,xv1,yv1;

Void setUpHistO()
{
  RealVector yp(nCount);

  dumpVals(out1,yp);

  Integer nBox = 25;

  makeHistogram(yp,nBox, yv2, xv2, yv1, xv1);
}

// Routine to plot histogram:

Void p1()
{
  xyPlot(yv2, xv2,
         lineColor = cycle,
         xLabel = "c1", yLabel = "Probability",
         title = "Distribution");
}


// Routine to plot cumulative cost probability:

Void p2()
{
  xyPlot(yv1, xv1,
         lineColor = cycle,
         xLabel = "c1", yLabel = "Cumulative Probability",
         title = "Cost Distribution");
}

setUpHistO();
p1();  // Plot c1
p2();  // Plot cumulative distribution of c1
```

References.

4.4.1 - Hybrd - Package from MINPACK library.
4.4.2 - Vmcon Roger L. Crane, K. E. Hillstrom, M. Minkoff, “Solution of the NonLinear Progamming Problem with Subroutine VMCON”, ANL-80-64.
4.4.3 - Galib, “A C++ Genetic Algorithms Library”, Mathew Wall, http://lancet.mit.edu/galib-2.4/ .

4.4.4 - PVM: Parallel Virtual Machine “A Users' Guide and Tutorial for Networked Parallel Computing”, Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jian, Robert Manchek, Vaidy Sunderam, http://www.netlib.org/pvm3/book/pvm-book.html .

4.5.1 - J. Galambos, J. Holmes, “Efficient Treatment of Uncertainty in Numerical Optimization”, Risk Analysis, **17**, 1997, p. 93.